

---

# CuPy Documentation

*Release 9.5.0*

**Preferred Networks, inc. and Preferred Infrastructure, inc.**

**Sep 30, 2021**



# CONTENTS

<b>1</b>	<b>Overview</b>	<b>1</b>
<b>2</b>	<b>Installation</b>	<b>3</b>
2.1	Requirements . . . . .	3
2.2	Installing CuPy . . . . .	4
2.3	Installing CuPy from Conda-Forge . . . . .	5
2.4	Installing CuPy from Source . . . . .	5
2.5	Uninstalling CuPy . . . . .	6
2.6	Upgrading CuPy . . . . .	6
2.7	Reinstalling CuPy . . . . .	7
2.8	Using CuPy inside Docker . . . . .	7
2.9	FAQ . . . . .	7
<b>3</b>	<b>Using CuPy on AMD GPU (experimental)</b>	<b>11</b>
3.1	Requirements . . . . .	11
3.2	Environment Variables . . . . .	11
3.3	Docker . . . . .	11
3.4	Installing Binary Packages . . . . .	12
3.5	Building CuPy for ROCm From Source . . . . .	12
3.6	Limitations . . . . .	12
<b>4</b>	<b>User Guide</b>	<b>15</b>
4.1	Basics of CuPy . . . . .	15
4.2	User-Defined Kernels . . . . .	18
4.3	Accessing CUDA Functionalities . . . . .	27
4.4	Fast Fourier Transform with CuPy . . . . .	29
4.5	Memory Management . . . . .	35
4.6	Performance Best Practices . . . . .	38
4.7	Interoperability . . . . .	39
4.8	Difference between CuPy and NumPy . . . . .	45
4.9	API Compatibility Policy . . . . .	48
<b>5</b>	<b>API Reference</b>	<b>51</b>
5.1	The N-dimensional array (ndarray) . . . . .	51
5.2	Universal functions (cupy.ufunc) . . . . .	65
5.3	Routines (NumPy) . . . . .	86
5.4	Routines (SciPy) . . . . .	272
5.5	CuPy-specific functions . . . . .	413
5.6	Low-level CUDA support . . . . .	419
5.7	Custom kernels . . . . .	475

5.8	Profiling . . . . .	488
5.9	Environment variables . . . . .	491
5.10	Comparison Table . . . . .	494
<b>6</b>	<b>Contribution Guide</b>	<b>521</b>
6.1	Classification of Contributions . . . . .	521
6.2	Development Cycle . . . . .	521
6.3	Issues and Pull Requests . . . . .	523
6.4	Coding Guidelines . . . . .	525
6.5	Unit Testing . . . . .	526
6.6	Documentation . . . . .	529
6.7	Tips for Developers . . . . .	529
<b>7</b>	<b>Upgrade Guide</b>	<b>531</b>
7.1	CuPy v9 . . . . .	531
7.2	CuPy v8 . . . . .	532
7.3	CuPy v7 . . . . .	533
7.4	CuPy v6 . . . . .	533
7.5	CuPy v5 . . . . .	533
7.6	CuPy v4 . . . . .	534
7.7	CuPy v2 . . . . .	535
<b>8</b>	<b>License</b>	<b>537</b>
8.1	NumPy . . . . .	537
8.2	SciPy . . . . .	538
	<b>Python Module Index</b>	<b>539</b>
	<b>Index</b>	<b>541</b>

## OVERVIEW

CuPy is an implementation of NumPy-compatible multi-dimensional array on CUDA. CuPy consists of `cupy.ndarray`, the core multi-dimensional array class, and many functions on it. It supports a subset of `numpy.ndarray` interface.

The following is a brief overview of supported subset of NumPy interface:

- Basic indexing (indexing by ints, slices, newaxes, and Ellipsis)
- Most of Advanced indexing (except for some indexing patterns with boolean masks)
- Data types (dtypes): `bool_`, `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32`, `uint64`, `float16`, `float32`, `float64`, `complex64`, `complex128`
- Most of the array creation routines (`empty`, `ones_like`, `diag`, etc.)
- Most of the array manipulation routines (`reshape`, `rollaxis`, `concatenate`, etc.)
- All operators with broadcasting
- All universal functions for elementwise operations (except those for complex numbers)
- Linear algebra functions, including product (`dot`, `matmul`, etc.) and decomposition (`cholesky`, `svd`, etc.), accelerated by `cuBLAS` and `cuSOLVER`
- Multi-dimensional fast Fourier transform (FFT), accelerated by `cuFFT`
- Reduction along axes (`sum`, `max`, `argmax`, etc.)

CuPy additionally supports a subset of SciPy features:

- Sparse matrices and sparse linear algebra, powered by `cuSPARSE`.
- Multi-dimensional image processing
- Signal processing
- Fast Fourier transform (FFT)
- Linear algebra functions
- Special functions
- Statistical functions

CuPy also includes the following features for performance:

- User-defined elementwise CUDA kernels
- User-defined reduction CUDA kernels
- Just-in-time compiler converting Python functions to CUDA kernels
- Fusing CUDA kernels to optimize user-defined calculation

- [CUB/cuTENSOR](#) backends for reduction and other routines
- Customizable memory allocator and memory pool
- [cuDNN](#) utilities
- Full coverage of [NCCL](#) APIs

CuPy uses on-the-fly kernel synthesis: when a kernel call is required, it compiles a kernel code optimized for the shapes and dtypes of given arguments, sends it to the GPU device, and executes the kernel. The compiled code is cached to `$(HOME)/.cupy/kernel_cache` directory (this cache path can be overwritten by setting the `CUPY_CACHE_DIR` environment variable). It may make things slower at the first kernel call, though this slow down will be resolved at the second execution. CuPy also caches the kernel code sent to GPU device within the process, which reduces the kernel transfer time on further calls.

## INSTALLATION

### 2.1 Requirements

- **NVIDIA CUDA GPU** with the Compute Capability 3.0 or larger.
- **CUDA Toolkit:** v9.2 / v10.0 / v10.1 / v10.2 / v11.0 / v11.1 / v11.2 / v11.3 / v11.4
  - If you have multiple versions of CUDA Toolkit installed, CuPy will automatically choose one of the CUDA installations. See *Working with Custom CUDA Installation* for details.
  - This requirement is optional if you install CuPy from `conda-forge`. However, you still need to have a compatible driver installed for your GPU. See *Installing CuPy from Conda-Forge* for details.
- **Python:** v3.6.0+ / v3.7.0+ / v3.8.0+ / v3.9.0+

---

**Note:** Currently, CuPy is tested against **Ubuntu** 18.04 LTS (x86\_64), **CentOS** 7 (x86\_64) and Windows Server 2016 (x86\_64).

---

#### 2.1.1 Python Dependencies

NumPy/SciPy-compatible API in CuPy v9 is based on NumPy 1.20 and SciPy 1.6, and has been tested against the following versions:

- **NumPy:** v1.17 / v1.18 / v1.19 / v1.20 / v1.21
- **SciPy (optional):** v1.4 / v1.5 / v1.6 / v1.7
  - Required only when using *Routines (SciPy)* (`cupyx.scipy`).
- **Optuna (optional):** v2.x
  - Required only when using *Automatic Kernel Parameters Optimizations* (`cupyx.optimizing`).

---

**Note:** SciPy and Optuna are optional dependencies and will not be installed automatically.

---

---

**Note:** Before installing CuPy, we recommend you to upgrade `setuptools` and `pip`:

```
$ python -m pip install -U setuptools pip
```

---

## 2.1.2 Additional CUDA Libraries

Part of the CUDA features in CuPy will be activated only when the corresponding libraries are installed.

- **cuTENSOR**: v1.2 / v1.3
  - The library to accelerate tensor operations. See [Environment variables](#) for the details.
- **NCCL**: v2.4 / v2.6 / v2.8 / v2.9 / v2.10 / v2.11
  - The library to perform collective multi-GPU / multi-node computations.
- **cuDNN**: v7.6 (CUDA 9.2 & 10.0) / v8.0 (CUDA 10.1) / v8.1 (CUDA 10.2+) / v8.2 (CUDA 10.2+)
  - The library to accelerate deep neural network computations.
- **cuSPARSELt**: v0.0.1 / v0.1.0
  - The library to accelerate sparse matrix-matrix multiplication.

## 2.2 Installing CuPy

Wheels (precompiled binary packages) are available for Linux (x86\_64) and Windows (amd64). Package names are different depending on your CUDA Toolkit version.

CUDA	Command
v9.2	\$ pip install cupy-cuda92
v10.0	\$ pip install cupy-cuda100
v10.1	\$ pip install cupy-cuda101
v10.2	\$ pip install cupy-cuda102
v11.0	\$ pip install cupy-cuda110
v11.1	\$ pip install cupy-cuda111
v11.2	\$ pip install cupy-cuda112
v11.3	\$ pip install cupy-cuda113
v11.4	\$ pip install cupy-cuda114

---

**Note:** To enable features provided by additional CUDA libraries (cuTENSOR / NCCL / cuDNN), you need to install them manually. If you installed CuPy via wheels, you can use the installer command below to setup these libraries in case you don't have a previous installation:

```
$ python -m cupyx.tools.install_library --cuda 11.2 --library cutensor
```

---

---

**Note:** Use `pip install cupy-cudaXXX -f https://pip.cupy.dev/pre` to install pre-release (development) versions.

---

When using wheels, please be careful not to install multiple CuPy packages at the same time. Any of these packages and cupy package (source installation) conflict with each other. Please make sure that only one CuPy package (cupy or cupy-cudaXX where XX is a CUDA version) is installed:

```
$ pip freeze | grep cupy
```

---



## 2.3 Installing CuPy from Conda-Forge

Conda/Anaconda is a cross-platform package management solution widely used in scientific computing and other fields. The above `pip install` instruction is compatible with conda environments. Alternatively, for both Linux and Windows once the CUDA driver is correctly set up, you can also install CuPy from the `conda-forge` channel:

```
$ conda install -c conda-forge cupy
```

and conda will install a pre-built CuPy binary package for you, along with the CUDA runtime libraries (`cuda-toolkit`). It is not necessary to install CUDA Toolkit in advance.

Conda has a built-in mechanism to determine and install the latest version of `cuda-toolkit` supported by your driver. However, if for any reason you need to force-install a particular CUDA version (say 10.0), you can do:

```
$ conda install -c conda-forge cupy cuda-toolkit=10.0
```

**Note:** `cuDNN`, `cuTENSOR`, and `NCCL` are available on `conda-forge` as optional dependencies. The following command can install them all at once:

```
$ conda install -c conda-forge cupy cuda-toolkit=10.1 cudnn cutensor nccl
```

Each of them can also be installed separately as needed. Note that `cuTENSOR` is available since CUDA 10.1+.

**Note:** If you encounter any problem with CuPy installed from `conda-forge`, please feel free to report to [cupy-feedstock](#), and we will help investigate if it is just a packaging issue in `conda-forge`'s recipe or a real issue in CuPy.

**Note:** If you did not install CUDA Toolkit by yourself, the `nvcc` compiler might not be available, as the `cuda-toolkit` package from `conda-forge` does not include the `nvcc` compiler toolchain. If you would like to use it from a local CUDA installation, you need to make sure the version of CUDA Toolkit matches that of `cuda-toolkit` to avoid surprises.

**Note:** Use `conda install -c conda-forge/label/cupy_rc cupy` if you want to install pre-release (development) versions.

## 2.4 Installing CuPy from Source

Use of wheel packages is recommended whenever possible. However, if wheels cannot meet your requirements (e.g., you are running non-Linux environment or want to use a version of CUDA / `cuDNN` / `NCCL` not supported by wheels), you can also build CuPy from source.

**Note:** CuPy source build requires `g++-6` or later. For Ubuntu 18.04, run `apt-get install g++`. For Ubuntu 16.04, CentOS 6 or 7, follow the instructions [here](#).

---

**Note:** When installing CuPy from source, features provided by additional CUDA libraries will be disabled if these libraries are not available at the build time. See *Installing cuDNN and NCCL* for the instructions.

---

---

**Note:** If you upgrade or downgrade the version of CUDA Toolkit, cuDNN, NCCL or cuTENSOR, you may need to reinstall CuPy. See *Reinstalling CuPy* for details.

---

You can install the latest stable release version of the [CuPy source package](#) via `pip`.

```
$ pip install cupy
```

If you want to install the latest development version of CuPy from a cloned Git repository:

```
$ git clone --recursive https://github.com/cupy/cupy.git
$ cd cupy
$ pip install .
```

---

**Note:** Cython 0.29.22 or later is required to build CuPy from source. It will be automatically installed during the build process if not available.

---

## 2.5 Uninstalling CuPy

Use `pip` to uninstall CuPy:

```
$ pip uninstall cupy
```

---

**Note:** If you are using a wheel, `cupy` shall be replaced with `cupy-cudaXX` (where XX is a CUDA version number).

---

---

**Note:** If CuPy is installed via `conda`, please do `conda uninstall cupy` instead.

---

## 2.6 Upgrading CuPy

Just use `pip install` with `-U` option:

```
$ pip install -U cupy
```

---

**Note:** If you are using a wheel, `cupy` shall be replaced with `cupy-cudaXX` (where XX is a CUDA version number).

---

## 2.7 Reinstalling CuPy

To reinstall CuPy, please uninstall CuPy and then install it. When reinstalling CuPy, we recommend using `--no-cache-dir` option as `pip` caches the previously built binaries:

```
$ pip uninstall cupy
$ pip install cupy --no-cache-dir
```

**Note:** If you are using a wheel, `cupy` shall be replaced with `cupy-cudaXX` (where `XX` is a CUDA version number).

## 2.8 Using CuPy inside Docker

We are providing the [official Docker images](#). Use [NVIDIA Container Toolkit](#) to run CuPy image with GPU. You can login to the environment with `bash`, and run the Python interpreter:

```
$ docker run --gpus all -it cupy/cupy /bin/bash
```

Or run the interpreter directly:

```
$ docker run --gpus all -it cupy/cupy /usr/bin/python3
```

## 2.9 FAQ

### 2.9.1 `pip` fails to install CuPy

Please make sure that you are using the latest `setuptools` and `pip`:

```
$ pip install -U setuptools pip
```

Use `-vvvv` option with `pip` command. This will display all logs of installation:

```
$ pip install cupy -vvvv
```

If you are using `sudo` to install CuPy, note that `sudo` command does not propagate environment variables. If you need to pass environment variable (e.g., `CUDA_PATH`), you need to specify them inside `sudo` like this:

```
$ sudo CUDA_PATH=/opt/nvidia/cuda pip install cupy
```

If you are using certain versions of `conda`, it may fail to build CuPy with error `g++: error: unrecognized command line option '-R'`. This is due to a bug in `conda` (see [conda/conda#6030](#) for details). If you encounter this problem, please upgrade your `conda`.

## 2.9.2 Installing cuDNN and NCCL

We recommend installing cuDNN and NCCL using binary packages (i.e., using `apt` or `yum`) provided by NVIDIA.

If you want to install tar-gz version of cuDNN and NCCL, we recommend installing it under the `CUDA_PATH` directory. For example, if you are using Ubuntu, copy `*.h` files to `include` directory and `*.so*` files to `lib64` directory:

```
$ cp /path/to/cudnn.h $CUDA_PATH/include
$ cp /path/to/libcudnn.so* $CUDA_PATH/lib64
```

The destination directories depend on your environment.

If you want to use cuDNN or NCCL installed in another directory, please use `CFLAGS`, `LD_FLAGS` and `LD_LIBRARY_PATH` environment variables before installing CuPy:

```
$ export CFLAGS=-I/path/to/cudnn/include
$ export LD_FLAGS=-L/path/to/cudnn/lib
$ export LD_LIBRARY_PATH=/path/to/cudnn/lib:$LD_LIBRARY_PATH
```

## 2.9.3 Working with Custom CUDA Installation

If you have installed CUDA on the non-default directory or multiple CUDA versions on the same host, you may need to manually specify the CUDA installation directory to be used by CuPy.

CuPy uses the first CUDA installation directory found by the following order.

1. `CUDA_PATH` environment variable.
2. The parent directory of `nvcc` command. CuPy looks for `nvcc` command from `PATH` environment variable.
3. `/usr/local/cuda`

For example, you can build CuPy using non-default CUDA directory by `CUDA_PATH` environment variable:

```
$ CUDA_PATH=/opt/nvidia/cuda pip install cupy
```

---

**Note:** CUDA installation discovery is also performed at runtime using the rule above. Depending on your system configuration, you may also need to set `LD_LIBRARY_PATH` environment variable to `$CUDA_PATH/lib64` at runtime.

---

## 2.9.4 CuPy always raises `cupy.cuda.compiler.CompileException`

If CuPy raises a `CompileException` for almost everything, it is possible that CuPy cannot detect CUDA installed on your system correctly. The followings are error messages commonly observed in such cases.

- `nVRTC: error: failed to load builtins`
- `catastrophic error: cannot open source file "cuda_fp16.h"`
- `error: cannot overload functions distinguished by return type alone`
- `error: identifier "__half_raw" is undefined`

Please try setting `LD_LIBRARY_PATH` and `CUDA_PATH` environment variable. For example, if you have CUDA installed at `/usr/local/cuda-9.2`:

```
$ export CUDA_PATH=/usr/local/cuda-9.2
$ export LD_LIBRARY_PATH=$CUDA_PATH/lib64:$LD_LIBRARY_PATH
```

Also see *Working with Custom CUDA Installation*.

## 2.9.5 Build fails on Ubuntu 16.04, CentOS 6 or 7

In order to build CuPy from source on systems with legacy GCC (g++-5 or earlier), you need to manually set up g++-6 or later and configure NVCC environment variable.

On Ubuntu 16.04:

```
$ sudo add-apt-repository ppa:ubuntu-toolchain-r/test
$ sudo apt update
$ sudo apt install g++-6
$ export NVCC="nvcc --compiler-bindir gcc-6"
```

On CentOS 6 / 7:

```
$ sudo yum install centos-release-scl
$ sudo yum install devtoolset-7-gcc-c++
$ source /opt/rh/devtoolset-7/enable
$ export NVCC="nvcc --compiler-bidir gcc-7"
```



## USING CUPY ON AMD GPU (EXPERIMENTAL)

CuPy has an experimental support for AMD GPU (ROCm).

### 3.1 Requirements

- AMD GPU supported by ROCm
- **ROCm: v4.0 / v4.2 / v4.3**
  - See the [ROCm Installation Guide](#) for details.

The following ROCm libraries are required:

```
$ sudo apt install hipblas hipsparse rocsparse rocrand rocthrust rocsolver rocfft hipcub.  
↪rocprim rccl
```

---

**Note:** ROCm 4.3 is known to have an issue related to clang/HIP. If you experienced an error like `fatal error: '__clang_hip_runtime_wrapper.h' file not found`, please try setting an environment variable `LLVM_PATH` (e.g., `export LLVM_PATH="${ROCM_HOME}/llvm"`). See [#5592](#) for details.

---

### 3.2 Environment Variables

When building or running CuPy for ROCm, the following environment variables are effective.

- `ROCM_HOME`: directory containing the ROCm software (e.g., `/opt/rocm`).

### 3.3 Docker

You can try running CuPy for ROCm using Docker.

```
$ docker run -it --device=/dev/kfd --device=/dev/dri --group-add video cupy/cupy-rocm
```

## 3.4 Installing Binary Packages

Wheels (precompiled binary packages) are available for Linux (x86\_64). Package names are different depending on your ROCm version.

ROCm	Command
v4.0	\$ pip install cupy-rocm-4-0
v4.2	\$ pip install cupy-rocm-4-2
v4.3	\$ pip install cupy-rocm-4-3

## 3.5 Building CuPy for ROCm From Source

To build CuPy from source, set the CUPY\_INSTALL\_USE\_HIP, ROCM\_HOME, and HCC\_AMDGPU\_TARGET environment variables. (HCC\_AMDGPU\_TARGET is the ISA name supported by your GPU. Run `rocm_info` and use the value displayed in Name: line (e.g., `gfx900`). You can specify a comma-separated list of ISAs if you have multiple GPUs of different architectures.)

```
$ export CUPY_INSTALL_USE_HIP=1
$ export ROCM_HOME=/opt/rocm
$ export HCC_AMDGPU_TARGET=gfx906
$ pip install cupy
```

---

**Note:** If you don't specify the HCC\_AMDGPU\_TARGET environment variable, CuPy will be built for the GPU architectures available on the build host. This behavior is specific to ROCm builds; when building CuPy for NVIDIA CUDA, the build result is not affected by the host configuration.

---

## 3.6 Limitations

The following features are not available due to the limitation of ROCm or because that they are specific to CUDA:

- CUDA Array Interface
- cuTENSOR
- Handling extremely large arrays whose size is around 32-bit boundary (HIP is known to fail with sizes  $2^{32}-1024$ )
- Atomic addition in FP16 (`cupy.ndarray.scatter_add` and `cupyx.scatter_add`)
- Multi-GPU FFT and FFT callback
- Some random number generation algorithms
- Several options in RawKernel/RawModule APIs: Jitify, dynamic parallelism
- Per-thread default stream

The following features are not yet supported:

- Sparse matrices (`cupyx.scipy.sparse`)
- cuDNN (hipDNN)
- Hermitian/symmetric eigenvalue solver (`cupy.linalg.eigh`)



- Polynomial roots (uses Hermitian/symmetric eigenvalue solver)

The following features may not work in edge cases (e.g., some combinations of dtype):

---

**Note:** We are investigating the root causes of the issues. They are not necessarily CuPy's issues, but ROCm may have some potential bugs.

---

- `cupy.ndarray.__getitem__` (#4653)
- `cupy.ix_` (#4654)
- Some polynomial routines (#4758, #4759)
- `cupy.broadcast` (#4662)
- `cupy.convolve` (#4668)
- `cupy.correlate` (#4781)
- Some random sampling routines (`cupy.random`, #4770)
- `cupy.linalg.einsum`
- `cupyx.scipy.ndimage` and `cupyx.scipy.signal` (#4878, #4879, #4880)



## USER GUIDE

This user guide provides an overview of CuPy and explains its important features; details are found in *CuPy API Reference*.

### 4.1 Basics of CuPy

In this section, you will learn about the following things:

- Basics of `cupy.ndarray`
- The concept of *current device*
- host-device and device-device array transfer

#### 4.1.1 Basics of `cupy.ndarray`

CuPy is a GPU array backend that implements a subset of NumPy interface. In the following code, `cp` is an abbreviation of `cupy`, following the standard convention of abbreviating `numpy` as `np`:

```
>>> import numpy as np
>>> import cupy as cp
```

The `cupy.ndarray` class is at the core of CuPy and is a replacement class for NumPy's `numpy.ndarray`.

```
>>> x_gpu = cp.array([1, 2, 3])
```

`x_gpu` above is an instance of `cupy.ndarray`. As one can see, CuPy's syntax here is identical to that of NumPy. The main difference between `cupy.ndarray` and `numpy.ndarray` is that the CuPy arrays are allocated on the *current device*, which we will talk about later.

Most of the array manipulations are also done in the way similar to NumPy. Take the Euclidean norm (a.k.a L2 norm), for example. NumPy has `numpy.linalg.norm()` function that calculates it on CPU.

```
>>> x_cpu = np.array([1, 2, 3])
>>> l2_cpu = np.linalg.norm(x_cpu)
```

Using CuPy, we can perform the same calculations on GPU in a similar way:

```
>>> x_gpu = cp.array([1, 2, 3])
>>> l2_gpu = cp.linalg.norm(x_gpu)
```

CuPy implements many functions on `cupy.ndarray` objects. See the [reference](#) for the supported subset of NumPy API. Knowledge of NumPy will help you utilize most of the CuPy features. We, therefore, recommend you familiarize yourself with the [NumPy documentation](#).

### 4.1.2 Current Device

CuPy has a concept of a *current device*, which is the default GPU device on which the allocation, manipulation, calculation, etc., of arrays take place. Suppose ID of the current device is 0. In such a case, the following code would create an array `x_on_gpu0` on GPU 0.

```
>>> x_on_gpu0 = cp.array([1, 2, 3, 4, 5])
```

The current device can be changed using `cupy.cuda.Device.use()` as follows:

```
>>> x_on_gpu0 = cp.array([1, 2, 3, 4, 5])
>>> cp.cuda.Device(1).use()
>>> x_on_gpu1 = cp.array([1, 2, 3, 4, 5])
```

To temporarily switch to another GPU device, use with context manager:

```
>>> with cp.cuda.Device(1):
...     x_on_gpu1 = cp.array([1, 2, 3, 4, 5])
>>> x_on_gpu0 = cp.array([1, 2, 3, 4, 5])
```

Most CuPy operations are performed on the currently active device and attempts to process an array stored on a different device will result in an error:

```
>>> with cp.cuda.Device(0):
...     x_on_gpu0 = cp.array([1, 2, 3, 4, 5])
>>> with cp.cuda.Device(1):
...     x_on_gpu0 * 2 # raises error
Traceback (most recent call last):
...
ValueError: Array device must be same as the current device: array device = 0 while
↳ current = 1
```

`cupy.ndarray.device` attribute indicates the device on which the array is allocated.

```
>>> with cp.cuda.Device(1):
...     x = cp.array([1, 2, 3, 4, 5])
>>> x.device
<CUDA Device 1>
```

---

**Note:** When only one device is available, explicit device switching is not needed.

---

### 4.1.3 Current Stream

Associated with the concept of current devices are *current streams*, which help avoid explicitly passing streams in every single operation so as to keep the APIs pythonic and user-friendly. In CuPy, all CUDA operations such as data transfer (see the [Data Transfer](#) section) and kernel launches are enqueued onto the current stream, and the queued tasks on the same stream will be executed in serial (but *asynchronously* with respect to the host).

The default current stream in CuPy is CUDA's null stream (i.e., stream 0). It is also known as the *legacy* default stream, which is unique per device. However, it is possible to change the current stream using the `cupy.cuda.Stream` API, please see [Accessing CUDA Functionalities](#) for example. The current stream in CuPy can be retrieved using `cupy.cuda.get_current_stream()`.

It is worth noting that CuPy's current stream is managed on a *per thread* basis, meaning that on different Python threads the current stream (if not the null stream) can be different.

### 4.1.4 Data Transfer

#### Move arrays to a device

`cupy.asarray()` can be used to move a `numpy.ndarray`, a list, or any object that can be passed to `numpy.array()` to the current device:

```
>>> x_cpu = np.array([1, 2, 3])
>>> x_gpu = cp.asarray(x_cpu) # move the data to the current device.
```

`cupy.asarray()` can accept `cupy.ndarray`, which means we can transfer the array between devices with this function.

```
>>> with cp.cuda.Device(0):
...     x_gpu_0 = cp.ndarray([1, 2, 3]) # create an array in GPU 0
>>> with cp.cuda.Device(1):
...     x_gpu_1 = cp.asarray(x_gpu_0) # move the array to GPU 1
```

**Note:** `cupy.asarray()` does not copy the input array if possible. So, if you put an array of the current device, it returns the input object itself.

If we do copy the array in this situation, you can use `cupy.array()` with `copy=True`. Actually `cupy.asarray()` is equivalent to `cupy.array(arr, dtype, copy=False)`.

#### Move array from a device to the host

Moving a device array to the host can be done by `cupy.asnumpy()` as follows:

```
>>> x_gpu = cp.array([1, 2, 3]) # create an array in the current device
>>> x_cpu = cp.asnumpy(x_gpu) # move the array to the host.
```

We can also use `cupy.ndarray.get()`:

```
>>> x_cpu = x_gpu.get()
```

### 4.1.5 Memory management

Check [Memory Management](#) for a detailed description of how memory is managed in CuPy using memory pools.

### 4.1.6 How to write CPU/GPU agnostic code

CuPy's compatibility with NumPy makes it possible to write CPU/GPU agnostic code. For this purpose, CuPy implements the `cupy.get_array_module()` function that returns a reference to `cupy` if any of its arguments resides on a GPU and `numpy` otherwise. Here is an example of a CPU/GPU agnostic function that computes `log1p`:

```
>>> # Stable implementation of log(1 + exp(x))
>>> def softplus(x):
...     xp = cp.get_array_module(x) # 'xp' is a standard usage in the community
...     print("Using:", xp.__name__)
...     return xp.maximum(0, x) + xp.log1p(xp.exp(-abs(x)))
```

When you need to manipulate CPU and GPU arrays, an explicit data transfer may be required to move them to the same location – either CPU or GPU. For this purpose, CuPy implements two sister methods called `cupy.asnumpy()` and `cupy.asarray()`. Here is an example that demonstrates the use of both methods:

```
>>> x_cpu = np.array([1, 2, 3])
>>> y_cpu = np.array([4, 5, 6])
>>> x_cpu + y_cpu
array([5, 7, 9])
>>> x_gpu = cp.asarray(x_cpu)
>>> x_gpu + y_cpu
Traceback (most recent call last):
...
TypeError: Unsupported type <class 'numpy.ndarray'>
>>> cp.asnumpy(x_gpu) + y_cpu
array([5, 7, 9])
>>> cp.asnumpy(x_gpu) + cp.asnumpy(y_cpu)
array([5, 7, 9])
>>> x_gpu + cp.asarray(y_cpu)
array([5, 7, 9])
>>> cp.asarray(x_gpu) + cp.asarray(y_cpu)
array([5, 7, 9])
```

The `cupy.asnumpy()` method returns a NumPy array (array on the host), whereas `cupy.asarray()` method returns a CuPy array (array on the current device). Both methods can accept arbitrary input, meaning that they can be applied to any data that is located on either the host or device and can be converted to an array.

## 4.2 User-Defined Kernels

CuPy provides easy ways to define three types of CUDA kernels: elementwise kernels, reduction kernels and raw kernels. In this documentation, we describe how to define and call each kernels.

### 4.2.1 Basics of elementwise kernels

An elementwise kernel can be defined by the `ElementwiseKernel` class. The instance of this class defines a CUDA kernel which can be invoked by the `__call__` method of this instance.

A definition of an elementwise kernel consists of four parts: an input argument list, an output argument list, a loop body code, and the kernel name. For example, a kernel that computes a squared difference  $f(x, y) = (x - y)^2$  is defined as follows:

```
>>> squared_diff = cp.ElementwiseKernel(
...     'float32 x, float32 y',
...     'float32 z',
...     'z = (x - y) * (x - y)',
...     'squared_diff')
```

The argument lists consist of comma-separated argument definitions. Each argument definition consists of a *type specifier* and an *argument name*. Names of NumPy data types can be used as type specifiers.

---

**Note:** `n`, `i`, and names starting with an underscore `_` are reserved for the internal use.

---

The above kernel can be called on either scalars or arrays with broadcasting:

```
>>> x = cp.arange(10, dtype=np.float32).reshape(2, 5)
>>> y = cp.arange(5, dtype=np.float32)
>>> squared_diff(x, y)
array([[ 0.,  0.,  0.,  0.,  0.],
       [25., 25., 25., 25., 25.]], dtype=float32)
>>> squared_diff(x, 5)
array([[25., 16.,  9.,  4.,  1.],
       [ 0.,  1.,  4.,  9., 16.]], dtype=float32)
```

Output arguments can be explicitly specified (next to the input arguments):

```
>>> z = cp.empty((2, 5), dtype=np.float32)
>>> squared_diff(x, y, z)
array([[ 0.,  0.,  0.,  0.,  0.],
       [25., 25., 25., 25., 25.]], dtype=float32)
```

### 4.2.2 Type-generic kernels

If a type specifier is one character, then it is treated as a **type placeholder**. It can be used to define a type-generic kernels. For example, the above `squared_diff` kernel can be made type-generic as follows:

```
>>> squared_diff_generic = cp.ElementwiseKernel(
...     'T x, T y',
...     'T z',
...     'z = (x - y) * (x - y)',
...     'squared_diff_generic')
```

Type placeholders of a same character in the kernel definition indicate the same type. The actual type of these placeholders is determined by the actual argument type. The `ElementwiseKernel` class first checks the output arguments and then the input arguments to determine the actual type. If no output arguments are given on the kernel invocation, then only the input arguments are used to determine the type.

The type placeholder can be used in the loop body code:

```
>>> squared_diff_generic = cp.ElementwiseKernel(
...     'T x, T y',
...     'T z',
...     """
...         T diff = x - y;
...         z = diff * diff;
...     """,
...     'squared_diff_generic')
```

More than one type placeholder can be used in a kernel definition. For example, the above kernel can be further made generic over multiple arguments:

```
>>> squared_diff_super_generic = cp.ElementwiseKernel(
...     'X x, Y y',
...     'Z z',
...     'z = (x - y) * (x - y)',
...     'squared_diff_super_generic')
```

Note that this kernel requires the output argument explicitly specified, because the type Z cannot be automatically determined from the input arguments.

### 4.2.3 Raw argument specifiers

The `ElementwiseKernel` class does the indexing with broadcasting automatically, which is useful to define most elementwise computations. On the other hand, we sometimes want to write a kernel with manual indexing for some arguments. We can tell the `ElementwiseKernel` class to use manual indexing by adding the `raw` keyword preceding the type specifier.

We can use the special variable `i` and method `_ind.size()` for the manual indexing. `i` indicates the index within the loop. `_ind.size()` indicates total number of elements to apply the elementwise operation. Note that it represents the size **after** broadcast operation.

For example, a kernel that adds two vectors with reversing one of them can be written as follows:

```
>>> add_reverse = cp.ElementwiseKernel(
...     'T x, raw T y', 'T z',
...     'z = x + y[_ind.size() - i - 1]',
...     'add_reverse')
```

(Note that this is an artificial example and you can write such operation just by `z = x + y[::-1]` without defining a new kernel). A raw argument can be used like an array. The indexing operator `y[_ind.size() - i - 1]` involves an indexing computation on `y`, so `y` can be arbitrarily shaped and stride.

Note that raw arguments are not involved in the broadcasting. If you want to mark all arguments as `raw`, you must specify the `size` argument on invocation, which defines the value of `_ind.size()`.



### 4.2.4 Texture memory

Texture objects (*TextureObject*) can be passed to *ElementwiseKernel* with their type marked by a unique type placeholder distinct from any other types used in the same kernel, as its actual datatype is determined when populating the texture memory. The texture coordinates can be computed in the kernel by the per-thread loop index *i*.

### 4.2.5 Reduction kernels

Reduction kernels can be defined by the *ReductionKernel* class. We can use it by defining four parts of the kernel code:

1. Identity value: This value is used for the initial value of reduction.
2. Mapping expression: It is used for the pre-processing of each element to be reduced.
3. Reduction expression: It is an operator to reduce the multiple mapped values. The special variables *a* and *b* are used for its operands.
4. Post mapping expression: It is used to transform the resulting reduced values. The special variable *a* is used as its input. Output should be written to the output parameter.

*ReductionKernel* class automatically inserts other code fragments that are required for an efficient and flexible reduction implementation.

For example, L2 norm along specified axes can be written as follows:

```
>>> l2norm_kernel = cp.ReductionKernel(
...     'T x', # input params
...     'T y', # output params
...     'x * x', # map
...     'a + b', # reduce
...     'y = sqrt(a)', # post-reduction map
...     '0', # identity value
...     'l2norm' # kernel name
... )
>>> x = cp.arange(10, dtype=np.float32).reshape(2, 5)
>>> l2norm_kernel(x, axis=1)
array([ 5.477226 , 15.9687195], dtype=float32)
```

**Note:** *raw* specifier is restricted for usages that the axes to be reduced are put at the head of the shape. It means, if you want to use *raw* specifier for at least one argument, the *axis* argument must be *0* or a contiguous increasing sequence of integers starting from *0*, like *(0, 1)*, *(0, 1, 2)*, etc.

**Note:** Texture memory is not yet supported in *ReductionKernel*.

## 4.2.6 Raw kernels

Raw kernels can be defined by the `RawKernel` class. By using raw kernels, you can define kernels from raw CUDA source.

`RawKernel` object allows you to call the kernel with CUDA's `cuLaunchKernel` interface. In other words, you have control over grid size, block size, shared memory size and stream.

```
>>> add_kernel = cp.RawKernel(r'''
... extern "C" __global__
... void my_add(const float* x1, const float* x2, float* y) {
...     int tid = blockDim.x * blockIdx.x + threadIdx.x;
...     y[tid] = x1[tid] + x2[tid];
... }
... ''', 'my_add')
>>> x1 = cp.arange(25, dtype=cp.float32).reshape(5, 5)
>>> x2 = cp.arange(25, dtype=cp.float32).reshape(5, 5)
>>> y = cp.zeros((5, 5), dtype=cp.float32)
>>> add_kernel((5,), (5,), (x1, x2, y)) # grid, block and arguments
>>> y
array([[ 0.,  2.,  4.,  6.,  8.],
       [10., 12., 14., 16., 18.],
       [20., 22., 24., 26., 28.],
       [30., 32., 34., 36., 38.],
       [40., 42., 44., 46., 48.]], dtype=float32)
```

Raw kernels operating on complex-valued arrays can be created as well:

```
>>> complex_kernel = cp.RawKernel(r'''
... #include <cupy/complex.cuh>
... extern "C" __global__
... void my_func(const complex<float>* x1, const complex<float>* x2,
...               complex<float>* y, float a) {
...     int tid = blockDim.x * blockIdx.x + threadIdx.x;
...     y[tid] = x1[tid] + a * x2[tid];
... }
... ''', 'my_func')
>>> x1 = cupy.arange(25, dtype=cupy.complex64).reshape(5, 5)
>>> x2 = 1j*cupy.arange(25, dtype=cupy.complex64).reshape(5, 5)
>>> y = cupy.zeros((5, 5), dtype=cupy.complex64)
>>> complex_kernel((5,), (5,), (x1, x2, y, cupy.float32(2.0))) # grid, block and
↪arguments
>>> y
array([[ 0. +0.j,  1. +2.j,  2. +4.j,  3. +6.j,  4. +8.j],
       [ 5. +10.j,  6. +12.j,  7. +14.j,  8. +16.j,  9. +18.j],
       [10. +20.j, 11. +22.j, 12. +24.j, 13. +26.j, 14. +28.j],
       [15. +30.j, 16. +32.j, 17. +34.j, 18. +36.j, 19. +38.j],
       [20. +40.j, 21. +42.j, 22. +44.j, 23. +46.j, 24. +48.j]],
      dtype=complex64)
```

Note that while we encourage the usage of `complex<T>` types for complex numbers (available by including `<cupy/complex.cuh>` as shown above), for CUDA codes already written using functions from `cuComplex.h` there is no need to make the conversion yourself: just set the option `translate_cucomplex=True` when creating a `RawKernel` instance.

The CUDA kernel attributes can be retrieved by either accessing the `attributes` dictionary, or by accessing the `RawKernel` object's attributes directly; the latter can also be used to set certain attributes:

```
>>> add_kernel = cp.RawKernel(r'''
... extern "C" __global__
... void my_add(const float* x1, const float* x2, float* y) {
...     int tid = blockDim.x * blockIdx.x + threadIdx.x;
...     y[tid] = x1[tid] + x2[tid];
... }
... ''', 'my_add')
>>> add_kernel.attributes
{'max_threads_per_block': 1024, 'shared_size_bytes': 0, 'const_size_bytes': 0, 'local_
↪size_bytes': 0, 'num_regs': 10, 'ptx_version': 70, 'binary_version': 70, 'cache_mode_ca
↪': 0, 'max_dynamic_shared_size_bytes': 49152, 'preferred_shared_memory_carveout': -1}
>>> add_kernel.max_dynamic_shared_size_bytes
49152
>>> add_kernel.max_dynamic_shared_size_bytes = 50000 # set a new value for the
↪attribute
>>> add_kernel.max_dynamic_shared_size_bytes
50000
```

Dynamical parallelism is supported by `RawKernel`. You just need to provide the linking flag (such as `-dc`) to `RawKernel`'s `options` argument. The static CUDA device runtime library (`cudadevrt`) is automatically discovered by CuPy. For further detail, see [CUDA Toolkit's documentation](#).

Accessing texture (surface) memory in `RawKernel` is supported via CUDA Runtime's Texture (Surface) Object API, see the documentation for [TextureObject](#) ([SurfaceObject](#)) as well as CUDA C Programming Guide. For using the Texture Reference API, which is marked as deprecated as of CUDA Toolkit 10.1, see the introduction to [RawModule](#) below.

If your kernel relies on the C++ std library headers such as `<type_traits>`, it is likely you will encounter compilation errors. In this case, try enabling CuPy's `Jitify` support by setting `jitify=True` when creating the `RawKernel` instance. It provides basic C++ std support to remedy common errors.

---

**Note:** The kernel does not have return values. You need to pass both input arrays and output arrays as arguments.

---



---

**Note:** No validation will be performed by CuPy for arguments passed to the kernel, including types and number of arguments. Especially note that when passing `ndarray`, its `dtype` should match with the type of the argument declared in the method signature of the CUDA source code (unless you are casting arrays intentionally). For example, `cupy.float32` and `cupy.uint64` arrays must be passed to the argument typed as `float*` and `unsigned long long*`. For Python primitive types, `int`, `float`, `complex` and `bool` map to `long long`, `double`, `cuDoubleComplex` and `bool`, respectively.

---



---

**Note:** When using `printf()` in your CUDA kernel, you may need to synchronize the stream to see the output. You can use `cupy.cuda.Stream.null.synchronize()` if you are using the default stream.

---



---

**Note:** In all of the examples above, we declare the kernels in an `extern "C"` block, indicating that the C linkage is used. This is to ensure the kernel names are not mangled so that they can be retrieved by name.

---

## 4.2.7 Raw modules

For dealing a large raw CUDA source or loading an existing CUDA binary, the `RawModule` class can be more handy. It can be initialized either by a CUDA source code, or by a path to the CUDA binary. It accepts most of the arguments as in `RawKernel`. The needed kernels can then be retrieved by calling the `get_function()` method, which returns a `RawKernel` instance that can be invoked as discussed above.

```
>>> loaded_from_source = r'''
... extern "C"{
...
... __global__ void test_sum(const float* x1, const float* x2, float* y, \
...                          unsigned int N)
... {
...     unsigned int tid = blockDim.x * blockIdx.x + threadIdx.x;
...     if (tid < N)
...     {
...         y[tid] = x1[tid] + x2[tid];
...     }
... }
...
... __global__ void test_multiply(const float* x1, const float* x2, float* y, \
...                               unsigned int N)
... {
...     unsigned int tid = blockDim.x * blockIdx.x + threadIdx.x;
...     if (tid < N)
...     {
...         y[tid] = x1[tid] * x2[tid];
...     }
... }
... }'''
>>> module = cp.RawModule(code=loaded_from_source)
>>> ker_sum = module.get_function('test_sum')
>>> ker_times = module.get_function('test_multiply')
>>> N = 10
>>> x1 = cp.arange(N**2, dtype=cp.float32).reshape(N, N)
>>> x2 = cp.ones((N, N), dtype=cp.float32)
>>> y = cp.zeros((N, N), dtype=cp.float32)
>>> ker_sum((N,), (N,), (x1, x2, y, N**2)) # y = x1 + x2
>>> assert cp.allclose(y, x1 + x2)
>>> ker_times((N,), (N,), (x1, x2, y, N**2)) # y = x1 * x2
>>> assert cp.allclose(y, x1 * x2)
```

The instruction above for using complex numbers in `RawKernel` also applies to `RawModule`.

For CUDA kernels that need to access global symbols, such as constant memory, the `get_global()` method can be used, see its documentation for further detail.

CuPy also supports the Texture Reference API. A handle to the texture reference in a module can be retrieved by name via `get_texref()`. Then, you need to pass it to `TextureReference`, along with a resource descriptor and texture descriptor, for binding the reference to the array. (The interface of `TextureReference` is meant to mimic that of `TextureObject` to help users make transition to the latter, since as of CUDA Toolkit 10.1 the former is marked as deprecated.)

To support C++ template kernels, `RawModule` additionally provide a `name_expressions` argument. A list of template specializations should be provided, so that the corresponding kernels can be generated and retrieved by type:

```

>>> code = r'''
... template<typename T>
... __global__ void fx3(T* arr, int N) {
...     unsigned int tid = blockIdx.x * blockDim.x + threadIdx.x;
...     if (tid < N) {
...         arr[tid] = arr[tid] * 3;
...     }
... }
... '''
>>>
>>> name_exp = ['fx3<float>', 'fx3<double>']
>>> mod = cp.RawModule(code=code, options=('-std=c++11',),
...     name_expressions=name_exp)
>>> ker_float = mod.get_function(name_exp[0]) # compilation happens here
>>> N=10
>>> a = cp.arange(N, dtype=cp.float32)
>>> ker_float((1,), (N,), (a, N))
>>> a
array([ 0.,  3.,  6.,  9., 12., 15., 18., 21., 24., 27.], dtype=float32)
>>> ker_double = mod.get_function(name_exp[1])
>>> a = cp.arange(N, dtype=cp.float64)
>>> ker_double((1,), (N,), (a, N))
>>> a
array([ 0.,  3.,  6.,  9., 12., 15., 18., 21., 24., 27.])

```

**Note:** The name expressions used to both initialize a [RawModule](#) instance and retrieve the kernels are the original (*unmangled*) kernel names with all template parameters unambiguously specified. The name mangling and demangling are handled under the hood so that users do not need to worry about it.

## 4.2.8 Kernel fusion

`cupy.fuse()` is a decorator that fuses functions. This decorator can be used to define an elementwise or reduction kernel more easily than [ElementwiseKernel](#) or [ReductionKernel](#).

By using this decorator, we can define the `squared_diff` kernel as follows:

```

>>> @cp.fuse()
... def squared_diff(x, y):
...     return (x - y) * (x - y)

```

The above kernel can be called on either scalars, NumPy arrays or CuPy arrays likes the original function.

```

>>> x_cp = cp.arange(10)
>>> y_cp = cp.arange(10)[::-1]
>>> squared_diff(x_cp, y_cp)
array([81, 49, 25,  9,  1,  1,  9, 25, 49, 81])
>>> x_np = np.arange(10)
>>> y_np = np.arange(10)[::-1]
>>> squared_diff(x_np, y_np)
array([81, 49, 25,  9,  1,  1,  9, 25, 49, 81])

```

At the first function call, the fused function analyzes the original function based on the abstracted information of arguments (e.g. their dtypes and ndims) and creates and caches an actual CUDA kernel. From the second function call with the same input types, the fused function calls the previously cached kernel, so it is highly recommended to reuse the same decorated functions instead of decorating local functions that are defined multiple times.

`cupy.fuse()` also supports simple reduction kernel.

```
>>> @cp.fuse()
... def sum_of_products(x, y):
...     return cp.sum(x * y, axis = -1)
```

You can specify the kernel name by using the `kernel_name` keyword argument as follows:

```
>>> @cp.fuse(kernel_name='squared_diff')
... def squared_diff(x, y):
...     return (x - y) * (x - y)
```

---

**Note:** Currently, `cupy.fuse()` can fuse only simple elementwise and reduction operations. Most other routines (e.g. `cupy.matmul()`, `cupy.reshape()`) are not supported.

---

## 4.2.9 JIT kernel definition

The `cupyx.jit.rawkernel` decorator can create raw CUDA kernels from Python functions.

In this section, a Python function wrapped with the decorator is called a *target function*.

A target function consists of elementary scalar operations, and users have to manage how to parallelize them. CuPy's array operations which automatically parallelize operations (e.g., `add()`, `sum()`) are not supported. If a custom kernel based on such array functions is desired, please refer to the [Kernel fusion](#) section.

### Basic Usage

Here is a short example for how to write a `cupyx.jit.rawkernel` to copy the values from `x` to `y` using a grid-stride loop:

```
>>> from cupyx import jit
>>>
>>> @jit.rawkernel()
... def elementwise_copy(x, y, size):
...     tid = jit.blockIdx.x * jit.blockDim.x + jit.threadIdx.x
...     ntid = jit.gridDim.x * jit.blockDim.x
...     for i in range(tid, size, ntid):
...         y[i] = x[i]

>>> size = cupy.uint32(2 ** 22)
>>> x = cupy.random.normal(size=(size,), dtype=cupy.float32)
>>> y = cupy.empty((size,), dtype=cupy.float32)

>>> elementwise_copy((128,), (1024,), (x, y, size)) # RawKernel style
>>> assert (x == y).all()
```

(continues on next page)

(continued from previous page)

```
>>> elementwise_copy[128, 1024](x, y, size) # Numba style
>>> assert (x == y).all()
```

The above two kinds of styles to launch the kernel are supported, see the documentation of [cupyx.jit.\\_interface.\\_JitRawKernel](#) for details.

The compilation will be deferred until the first function call. CuPy's JIT compiler infers the types of arguments at the call time, and will cache the compiled kernels for speeding up any subsequent calls.

See [Custom kernels](#) for a full list of API.

## Basic Design

CuPy's JIT compiler generates CUDA code via Python AST. We decided not to use Python bytecode to analyze the target function to avoid performance degradation. The CUDA source code generated from the Python bytecode will not effectively be optimized by CUDA compiler, because for-loops and other control statements of the target function are fully transformed to jump instruction when converting the target function to bytecode.

## Typing rule

The types of local variables are inferred at the first assignment in the function. The first assignment must be done at the top-level of the function; in other words, it must *not* be in `if/else` bodies or `for`-loops.

## Limitations

CuPy's JIT compiler uses `inspect.getsource()` to get the source code of the target function, so the compiler does not work in the following situations:

- In Python REPL
- Lambda expressions as target functions

## 4.3 Accessing CUDA Functionalities

### 4.3.1 Streams and Events

In this section we discuss basic usages for CUDA streams and events. For the API reference please see [Streams and events](#). For their roles in the CUDA programming model, please refer to [CUDA Programming Guide](#).

CuPy provides high-level Python APIs [Stream](#) and [Event](#) for creating streams and events, respectively. Data copies and kernel launches are enqueued onto the [Current Stream](#), which can be queried via `get_current_stream()` and changed either by setting up a context manager:

```
>>> import numpy as np
>>>
>>> a_np = np.arange(10)
>>> s = cp.cuda.Stream()
>>> with s:
...     a_cp = cp.asarray(a_np) # H2D transfer on stream s
...     b_cp = cp.sum(a_cp)     # kernel launched on stream s
...     assert s == cp.cuda.get_current_stream()
```

(continues on next page)

(continued from previous page)

```
...
>>> # fall back to the previous stream in use (here the default stream)
>>> # when going out of the scope of s
```

or by using the `use()` method:

```
>>> s = cp.cuda.Stream()
>>> s.use() # any subsequent operations are done on steam s
<Stream ...>
>>> b_np = cp.asnumpy(b_cp)
>>> assert s == cp.cuda.get_current_stream()
>>> cp.cuda.Stream.null.use() # fall back to the default (null) stream
<Stream 0>
>>> assert cp.cuda.Stream.null == cp.cuda.get_current_stream()
```

Events can be created either manually or through the `record()` method. `Event` objects can be used for timing GPU activities (via `get_elapsed_time()`) or setting up inter-stream dependencies:

```
>>> e1 = cp.cuda.Event()
>>> e1.record()
>>> a_cp = b_cp * a_cp + 8
>>> e2 = cp.cuda.get_current_stream().record()
>>>
>>> # set up a stream order
>>> s2 = cp.cuda.Stream()
>>> s2.wait_event(e2)
>>> with s2:
...     # the a_cp is guaranteed updated when this copy (on s2) starts
...     a_np = cp.asnumpy(a_cp)
>>>
>>> # timing
>>> e2.synchronize()
>>> t = cp.cuda.get_elapsed_time(e1, e2) # only include the compute time, not the copy_
↳time
```

Just like the `Device` objects, `Stream` and `Event` objects can also be used for synchronization.

---

**Note:** In CuPy, the `Stream` objects are managed on the per thread basis.

---



---

**Note:** On NVIDIA GPUs, there are two stream singleton objects `null` and `ptds`, referred to as the *legacy* default stream and the *per-thread* default stream, respectively. CuPy uses the former as default when no user-defined stream is in use. To change this behavior, set the environment variable `CUPY_CUDA_PER_THREAD_DEFAULT_STREAM` to 1, see *Environment variables*. This is not applicable to AMD GPUs.

---

To interoperate with streams created in other Python libraries, CuPy provides the `ExternalStream` API to wrap an existing stream pointer (given as a Python `int`). In this case, the stream lifetime is not managed by CuPy. In addition, you need to make sure the `ExternalStream` object is used on the device where the stream was created. But the created `ExternalStream` object can otherwise be used like a `Stream` object.



### 4.3.2 CUDA Driver and Runtime API

Under construction. Please see [Runtime API](#) for the API reference.

## 4.4 Fast Fourier Transform with CuPy

CuPy covers the full Fast Fourier Transform (FFT) functionalities provided in NumPy ([`numpy.fft`](#)) and a subset in SciPy ([`scipy.fft`](#)). In addition to those high-level APIs that can be used as is, CuPy provides additional features to

1. access advanced routines that [cuFFT](#) offers for NVIDIA GPUs,
2. control better the performance and behavior of the FFT routines.

Some of these features are *experimental* (subject to change, deprecation, or removal, see [API Compatibility Policy](#)) or may be absent in [hipFFT/rocFFT](#) targeting AMD GPUs.

### 4.4.1 SciPy FFT backend

Since SciPy v1.4 a backend mechanism is provided so that users can register different FFT backends and use SciPy's API to perform the actual transform with the target backend, such as CuPy's [`cupyx.scipy.fft`](#) module. For a one-time only usage, a context manager [`scipy.fft.set\_backend\(\)`](#) can be used:

```
import cupy as cp
import cupyx.scipy.fft as cufft
import scipy.fft

a = cp.random.random(100).astype(cp.complex64)
with scipy.fft.set_backend(cufft):
    b = scipy.fft.fft(a) # equivalent to cufft.fft(a)
```

However, such usage can be tedious. Alternatively, users can register a backend through [`scipy.fft.register\_backend\(\)`](#) or [`scipy.fft.set\_global\_backend\(\)`](#) to avoid using context managers:

```
import cupy as cp
import cupyx.scipy.fft as cufft
import scipy.fft
scipy.fft.set_global_backend(cufft)

a = cp.random.random(100).astype(cp.complex64)
b = scipy.fft.fft(a) # equivalent to cufft.fft(a)
```

---

**Note:** Please refer to [SciPy FFT documentation](#) for further information.

---



---

**Note:** To use the backend together with an explicit `plan` argument requires SciPy version 1.5.0 or higher. See below for how to create FFT plans.

---

## 4.4.2 User-managed FFT plans

For performance reasons, users may wish to create, reuse, and manage the FFT plans themselves. CuPy provides a high-level *experimental* API `get_fft_plan()` for this need. Users specify the transform to be performed as they would with most of the high-level FFT APIs, and a plan will be generated based on the input.

```
import cupy as cp
from cupyx.scipy.fft import get_fft_plan

a = cp.random.random((4, 64, 64)).astype(cp.complex64)
plan = get_fft_plan(a, axes=(1, 2), value_type='C2C') # for batched, C2C, 2D transform
```

The returned plan can be used either explicitly as an argument with the `cupyx.scipy.fft` APIs:

```
import cupyx.scipy.fft

# the rest of the arguments must match those used when generating the plan
out = cupyx.scipy.fft.fft2(a, axes=(1, 2), plan=plan)
```

or as a context manager for the `cupy.fft` APIs:

```
with plan:
    # the arguments must match those used when generating the plan
    out = cp.fft.fft2(a, axes=(1, 2))
```

## 4.4.3 FFT plan cache

However, there are occasions when users may *not* want to manage the FFT plans by themselves. Moreover, plans could also be reused internally in CuPy's routines, to which user-managed plans would not be applicable. Therefore, starting CuPy v8 we provide a built-in plan cache, enabled by default. The plan cache is done on a *per device, per thread* basis, and can be retrieved by the `get_plan_cache()` API.

```
>>> import cupy as cp
>>>
>>> cache = cp.fft.config.get_plan_cache()
>>> cache.show_info()
----- cuFFT plan cache (device 0) -----
cache enabled? True
current / max size   : 0 / 16 (counts)
current / max memsize: 0 / (unlimited) (bytes)
hits / misses: 0 / 0 (counts)

cached plans (most recently used first):

>>> # perform a transform, which would generate a plan and cache it
>>> a = cp.random.random((4, 64, 64))
>>> out = cp.fft.fftn(a, axes=(1, 2))
>>> cache.show_info() # hit = 0
----- cuFFT plan cache (device 0) -----
cache enabled? True
current / max size   : 1 / 16 (counts)
current / max memsize: 262144 / (unlimited) (bytes)
hits / misses: 0 / 1 (counts)
```

(continues on next page)

(continued from previous page)

```

cached plans (most recently used first):
key: ((64, 64), (64, 64), 1, 4096, (64, 64), 1, 4096, 105, 4, 'C', 2, None), plan type:↳
↳PlanNd, memory usage: 262144

>>> # perform the same transform again, the plan is looked up from cache and reused
>>> out = cp.fft.fftn(a, axes=(1, 2))
>>> cache.show_info() # hit = 1
----- cuFFT plan cache (device 0) -----
cache enabled? True
current / max size   : 1 / 16 (counts)
current / max memsize: 262144 / (unlimited) (bytes)
hits / misses: 1 / 1 (counts)

cached plans (most recently used first):
key: ((64, 64), (64, 64), 1, 4096, (64, 64), 1, 4096, 105, 4, 'C', 2, None), plan type:↳
↳PlanNd, memory usage: 262144

>>> # clear the cache
>>> cache.clear()
>>> cp.fft.config.show_plan_cache_info() # = cache.show_info(), for all devices
===== cuFFT plan cache info (all devices) =====
----- cuFFT plan cache (device 0) -----
cache enabled? True
current / max size   : 0 / 16 (counts)
current / max memsize: 0 / (unlimited) (bytes)
hits / misses: 0 / 0 (counts)

cached plans (most recently used first):

```

The returned PlanCache object has other methods for finer control, such as setting the cache size (either by counts or by memory usage). If the size is set to 0, the cache is disabled. Please refer to its documentation for more detail.

**Note:** As shown above each FFT plan has an associated working area allocated. If an out-of-memory error happens, one may want to inspect, clear, or limit the plan cache.

**Note:** The plans returned by `get_fft_plan()` are not cached.

#### 4.4.4 FFT callbacks

cuFFT provides FFT callbacks for merging pre- and/or post- processing kernels with the FFT routines so as to reduce the access to global memory. This capability is supported *experimentally* by CuPy. Users need to supply custom load and/or store kernels as strings, and set up a context manager via `set_cufft_callbacks()`. Note that the load (store) kernel pointer has to be named as `d_loadCallbackPtr` (`d_storeCallbackPtr`).

```

import cupy as cp

# a load callback that overwrites the input array to 1

```

(continues on next page)

(continued from previous page)

```
code = r'''
__device__ cufftComplex CB_ConvertInputC(
    void *dataIn,
    size_t offset,
    void *callerInfo,
    void *sharedPtr)
{
    cufftComplex x;
    x.x = 1.;
    x.y = 0.;
    return x;
}
__device__ cufftCallbackLoadC d_loadCallbackPtr = CB_ConvertInputC;
'''

a = cp.random.random((64, 128, 128)).astype(cp.complex64)

# this fftn call uses callback
with cp.fft.config.set_cufft_callbacks(cb_load=code):
    b = cp.fft.fftn(a, axes=(1,2))

# this does not use
c = cp.fft.fftn(cp.ones(shape=a.shape, dtype=cp.complex64), axes=(1,2))

# result agrees
assert cp.allclose(b, c)

# "static" plans are also cached, but are distinct from their no-callback counterparts
cp.fft.config.get_plan_cache().show_info()
```

**Note:** Internally, this feature requires recompiling a Python module *for each distinct pair* of load and store kernels. Therefore, the first invocation will be very slow, and this cost is amortized if the callbacks can be reused in the subsequent calculations. The compiled modules are cached on disk, with a default position `$HOME/.cupy/callback_cache` that can be changed by the environment variable `CUPY_CACHE_DIR`.

## 4.4.5 Multi-GPU FFT

CuPy currently provides two kinds of *experimental* support for multi-GPU FFT.

**Warning:** Using multiple GPUs to perform FFT is not guaranteed to be more performant. The rule of thumb is if the transform fits in 1 GPU, you should avoid using multiple.

The first kind of support is with the high-level `fftn()` and `ifftn()` APIs, which requires the input array to reside on one of the participating GPUs. The multi-GPU calculation is done under the hood, and by the end of the calculation the result again resides on the device where it started. Currently only 1D complex-to-complex (C2C) transform is supported; complex-to-real (C2R) or real-to-complex (R2C) transforms (such as `rfftn()` and friends) are not. The transform can be either batched (batch size > 1) or not (batch size = 1).

```
import cupy as cp

cp.fft.config.use_multi_gpus = True
cp.fft.config.set_cufft_gpus([0, 1]) # use GPU 0 & 1

shape = (64, 64) # batch size = 64
dtype = cp.complex64
a = cp.random.random(shape).astype(dtype) # reside on GPU 0

b = cp.fft.fft(a) # computed on GPU 0 & 1, reside on GPU 0
```

If you need to perform 2D/3D transforms (ex: `fftn()`) instead of 1D (ex: `fft()`), it would likely still work, but in this particular use case it loops over the transformed axes under the hood (which is exactly what is done in NumPy too), which could lead to suboptimal performance.

The second kind of usage is to use the low-level, *private* CuPy APIs. You need to construct a `Plan1d` object and use it as if you are programming in C/C++ with `cuFFT`. Using this approach, your input array can reside on the host as a `numpy.ndarray` so that its size can be much larger than what a single GPU can accommodate, which is one of the main reasons to run multi-GPU FFT.

```
import numpy as np
import cupy as cp

# no need to touch cp.fft.config, as we are using low-level API

shape = (64, 64)
dtype = np.complex64
a = np.random.random(shape).astype(dtype) # reside on CPU

if len(shape) == 1:
    batch = 1
    nx = shape[0]
elif len(shape) == 2:
    batch = shape[0]
    nx = shape[1]

# compute via cuFFT
cufft_type = cp.cuda.cufft.CUFFT_C2C # single-precision c2c
plan = cp.cuda.cufft.Plan1d(nx, cufft_type, batch, devices=[0,1])
out_cp = np.empty_like(a) # output on CPU
plan.fft(a, out_cp, cufft.CUFFT_FORWARD)

out_np = numpy.fft.fft(a) # use NumPy's fft
# np.fft.fft always returns np.complex128
if dtype is numpy.complex64:
    out_np = out_np.astype(dtype)

# check result
assert np.allclose(out_cp, out_np, rtol=1e-4, atol=1e-7)
```

For this use case, please consult the `cuFFT` documentation on multi-GPU transform for further detail.

**Note:** The multi-GPU plans are cached if auto-generated via the high-level APIs, but not if manually generated via

the low-level APIs.

---

#### 4.4.6 Half-precision FFT

cuFFT provides `cufftXtMakePlanMany` and `cufftXtExec` routines to support a wide range of FFT needs, including 64-bit indexing and half-precision FFT. CuPy provides an *experimental* support for this capability via the new (though *private*) `XtPlanNd` API. For half-precision FFT, on supported hardware it can be twice as fast than its single-precision counterpart. NumPy does not yet provide the necessary infrastructure for half-precision complex numbers (i.e., `numpy.complex32`), though, so the steps for this feature is currently a bit more involved than common cases.

```
import cupy as cp
import numpy as np

shape = (1024, 256, 256) # input array shape
idtype = odtype = edtype = 'E' # = numpy.complex32 in the future

# store the input/output arrays as fp16 arrays twice as long, as complex32 is not yet
# available
a = cp.random.random((shape[0], shape[1], 2*shape[2])).astype(cp.float16)
out = cp.empty_like(a)

# FFT with cuFFT
plan = cp.cuda.cufft.XtPlanNd(shape[1:],
                               shape[1:], 1, shape[1]*shape[2], idtype,
                               shape[1:], 1, shape[1]*shape[2], odtype,
                               shape[0], edtype,
                               order='C', last_axis=-1, last_size=None)

plan.fft(a, out, cp.cuda.cufft.CUFFT_FORWARD)

# FFT with NumPy
a_np = cp.asnumpy(a).astype(np.float32) # upcast
a_np = a_np.view(np.complex64)
out_np = np.fft.fftn(a_np, axes=(-2,-1))
out_np = np.ascontiguousarray(out_np).astype(np.complex64) # downcast
out_np = out_np.view(np.float32)
out_np = out_np.astype(np.float16)

# don't worry about accuracy for now, as we probably lost a lot during casting
print('ok' if cp.mean(cp.abs(out - cp.asarray(out_np))) < 0.1 else 'not ok')
```

The 64-bit indexing support for all high-level FFT APIs is planned for a future CuPy release.

## 4.5 Memory Management

CuPy uses *memory pool* for memory allocations by default. The memory pool significantly improves the performance by mitigating the overhead of memory allocation and CPU/GPU synchronization.

There are two different memory pools in CuPy:

- Device memory pool (GPU device memory), which is used for GPU memory allocations.
- Pinned memory pool (non-swappable CPU memory), which is used during CPU-to-GPU data transfer.

**Attention:** When you monitor the memory usage (e.g., using `nvidia-smi` for GPU memory or `ps` for CPU memory), you may notice that memory not being freed even after the array instance become out of scope. This is an expected behavior, as the default memory pool “caches” the allocated memory blocks.

See *Low-level CUDA support* for the details of memory management APIs.

For using pinned memory more conveniently, we also provide a few high-level APIs in the `cupyx` namespace, including `cupyx.empty_pinned()`, `cupyx.empty_like_pinned()`, `cupyx.zeros_pinned()`, and `cupyx.zeros_like_pinned()`. They return NumPy arrays backed by pinned memory. If CuPy’s pinned memory pool is in use, the pinned memory is allocated from the pool.

**Note:** CuPy v8 and above provides a *FFT plan cache* that could use a portion of device memory if FFT and related functions are used. The memory taken can be released by shrinking or disabling the cache.

### 4.5.1 Memory Pool Operations

The memory pool instance provides statistics about memory allocation. To access the default memory pool instance, use `cupy.get_default_memory_pool()` and `cupy.get_default_pinned_memory_pool()`. You can also free all unused memory blocks hold in the memory pool. See the example code below for details:

```
import cupy
import numpy

mempool = cupy.get_default_memory_pool()
pinned_mempool = cupy.get_default_pinned_memory_pool()

# Create an array on CPU.
# NumPy allocates 400 bytes in CPU (not managed by CuPy memory pool).
a_cpu = numpy.ndarray(100, dtype=numpy.float32)
print(a_cpu.nbytes)                # 400

# You can access statistics of these memory pools.
print(mempool.used_bytes())         # 0
print(mempool.total_bytes())        # 0
print(pinned_mempool.n_free_blocks()) # 0

# Transfer the array from CPU to GPU.
# This allocates 400 bytes from the device memory pool, and another 400
# bytes from the pinned memory pool. The allocated pinned memory will be
# released just after the transfer is complete. Note that the actual
```

(continues on next page)

(continued from previous page)

```

# allocation size may be rounded to larger value than the requested size
# for performance.
a = cupy.array(a_cpu)
print(a.nbytes)                # 400
print(mempool.used_bytes())     # 512
print(mempool.total_bytes())    # 512
print(pinned_mempool.n_free_blocks()) # 1

# When the array goes out of scope, the allocated device memory is released
# and kept in the pool for future reuse.
a = None # (or `del a`)
print(mempool.used_bytes())     # 0
print(mempool.total_bytes())    # 512
print(pinned_mempool.n_free_blocks()) # 1

# You can clear the memory pool by calling `free_all_blocks`.
mempool.free_all_blocks()
pinned_mempool.free_all_blocks()
print(mempool.used_bytes())     # 0
print(mempool.total_bytes())    # 0
print(pinned_mempool.n_free_blocks()) # 0

```

See `cupy.cuda.MemoryPool` and `cupy.cuda.PinnedMemoryPool` for details.

## 4.5.2 Limiting GPU Memory Usage

You can hard-limit the amount of GPU memory that can be allocated by using `CUPY_GPU_MEMORY_LIMIT` environment variable (see *Environment variables* for details).

```

# Set the hard-limit to 1 GiB:
# $ export CUPY_GPU_MEMORY_LIMIT="1073741824"

# You can also specify the limit in fraction of the total amount of memory
# on the GPU. If you have a GPU with 2 GiB memory, the following is
# equivalent to the above configuration.
# $ export CUPY_GPU_MEMORY_LIMIT="50%"

import cupy
print(cupy.get_default_memory_pool().get_limit()) # 1073741824

```

You can also set the limit (or override the value specified via the environment variable) using `cupy.cuda.MemoryPool.set_limit()`. In this way, you can use a different limit for each GPU device.

```

import cupy

mempool = cupy.get_default_memory_pool()

with cupy.cuda.Device(0):
    mempool.set_limit(size=1024**3) # 1 GiB

with cupy.cuda.Device(1):
    mempool.set_limit(size=2*1024**3) # 2 GiB

```



---

**Note:** CUDA allocates some GPU memory outside of the memory pool (such as CUDA context, library handles, etc.). Depending on the usage, such memory may take one to few hundred MiB. That will not be counted in the limit.

---

### 4.5.3 Changing Memory Pool

You can use your own memory allocator instead of the default memory pool by passing the memory allocation function to `cupy.cuda.set_allocator()` / `cupy.cuda.set_pinned_memory_allocator()`. The memory allocator function should take 1 argument (the requested size in bytes) and return `cupy.cuda.MemoryPointer` / `cupy.cuda.PinnedMemoryPointer`.

CuPy provides two such allocators for using managed memory and stream ordered memory on GPU, see `cupy.cuda.malloc_managed()` and `cupy.cuda.malloc_async()`, respectively, for details. To enable a memory pool backed by managed memory, you can construct a new `MemoryPool` instance with its allocator set to `malloc_managed()` as follows

```
import cupy

# Use managed memory
cupy.cuda.set_allocator(cupy.cuda.MemoryPool(cupy.cuda.malloc_managed).malloc)
```

Note that if you pass `malloc_managed()` directly to `set_allocator()` without constructing a `MemoryPool` instance, when the memory is freed it will be released back to the system immediately, which may or may not be desired.

Stream Ordered Memory Allocator is a new feature added since CUDA 11.2. CuPy provides an *experimental* interface to it. Similar to CuPy's memory pool, Stream Ordered Memory Allocator also allocates/deallocates memory *asynchronously* from/to a memory pool in a stream-ordered fashion. The key difference is that it is a built-in feature implemented in the CUDA driver by NVIDIA. To enable a memory pool that manages stream ordered memory, you can construct a new `MemoryAsyncPool` instance:

```
import cupy

# Use asynchronous stream ordered memory
cupy.cuda.set_allocator(cupy.cuda.MemoryAsyncPool().malloc)

# Create a custom stream
s = cupy.cuda.Stream()

# This would allocate memory asynchronously on stream s
with s:
    a = cupy.empty((100,), dtype=cupy.float64)
```

Note that in this case we do not use the `MemoryPool` class. The `MemoryAsyncPool` takes a different input argument from that of `MemoryPool` to indicate which pool to use. Please refer to `MemoryAsyncPool`'s documentation for further detail.

Note that if you pass `malloc_async()` directly to `set_allocator()` without constructing a `MemoryAsyncPool` instance, the device's *current* memory pool will be used.

When using stream ordered memory, it is important that you maintain a correct stream semantics yourselves using, for example, the `Stream` and `Event` APIs (see *Streams and Events* for details); CuPy does not attempt to act smartly for you. Upon deallocation, the memory is freed asynchronously either on the stream it was allocated (first attempt), or on any current CuPy stream (second attempt). It is permitted that the stream on which the memory was allocated gets destroyed before all memory allocated on it is freed.

In addition, applications/libraries internally use `cudaMalloc` (CUDA’s default, synchronous allocator) could have unexpected interplay with Stream Ordered Memory Allocator. Specifically, memory freed to the memory pool might not be immediately visible to `cudaMalloc`, leading to potential out-of-memory errors. In this case, you can either call `free_all_blocks()` or just manually perform a (event/stream/device) synchronization, and retry.

Currently the `MemoryAsyncPool` interface is *experimental*. In particular, unlike `MemoryPool` or `PinnedMemoryPool` most of the pool’s methods are not supported due to CUDA’s limitation.

You can even disable the default memory pool by the code below. Be sure to do this before any other CuPy operations.

```
import cupy

# Disable memory pool for device memory (GPU)
cupy.cuda.set_allocator(None)

# Disable memory pool for pinned memory (CPU).
cupy.cuda.set_pinned_memory_allocator(None)
```

## 4.6 Performance Best Practices

Here we gather a few tricks and advices for improving CuPy’s performance.

### 4.6.1 Benchmarking

It is utterly important to first identify the performance bottleneck before making any attempt to optimize your code. To help set up a baseline benchmark, CuPy provides a useful utility `cupyx.time.repeat()` for timing the elapsed time of a Python function on both CPU and GPU:

```
>>> from cupyx.time import repeat
>>>
>>> def my_func(a):
...     return cp.sqrt(cp.sum(a**2, axis=-1))
...
>>> a = cp.random.random((256, 1024))
>>> print(repeat(my_func, (a,), n_repeat=20))
my_func          :    CPU:   44.407 us   +/- 2.428 (min:   42.516 / max:   53.098) us
↳ GPU-0:  181.565 us   +/- 1.853 (min:  180.288 / max:  188.608) us
```

Because GPU executions run asynchronously with respect to CPU executions, a common pitfall in GPU programming is to mistakenly measure the elapsed time using CPU timing utilities (such as `time.perf_counter()` from the Python Standard Library or the `%timeit` magic from IPython), which have no knowledge in the GPU runtime. `cupyx.time.repeat()` addresses this by setting up CUDA events on the *Current Stream* right before and after the function to be measured and synchronizing over the end event (see *Streams and Events* for detail). Below we sketch what is done internally in `cupyx.time.repeat()`:

```
>>> import time
>>> start_gpu = cp.cuda.Event()
>>> end_gpu = cp.cuda.Event()
>>>
>>> start_gpu.record()
>>> start_cpu = time.perf_counter()
>>> out = my_func(a)
```

(continues on next page)

(continued from previous page)

```
>>> end_cpu = time.perf_counter()
>>> end_gpu.record()
>>> end_gpu.synchronize()
>>> t_gpu = cp.cuda.get_elapsed_time(start_gpu, end_gpu)
>>> t_cpu = end_cpu - start_cpu
```

Additionally, `cupyx.time.repeat()` runs a few warm-up runs to reduce timing fluctuation and exclude the overhead in first invocations.

## 4.6.2 In-depth profiling

Under construction.

## 4.6.3 Use CUB/cuTENSOR backends for reduction operations

Under construction.

## 4.6.4 Overlapping work using streams

Under construction.

## 4.6.5 Use JIT compiler

Under construction. For now please refer to *JIT kernel definition* for a quick introduction.

## 4.6.6 Prefer float32 over float64

Under construction.

# 4.7 Interoperability

CuPy can be used in conjunction with other libraries.

## 4.7.1 CUDA functionalities

Under construction. For using CUDA streams created in foreign libraries in CuPy, see *Streams and Events*.

## 4.7.2 NumPy

`cupy.ndarray` implements `__array_ufunc__` interface (see [NEP 13 — A Mechanism for Overriding Ufuncs](#) for details). This enables NumPy ufuncs to be directly operated on CuPy arrays. `__array_ufunc__` feature requires NumPy 1.13 or later.

```
import cupy
import numpy

arr = cupy.random.randn(1, 2, 3, 4).astype(cupy.float32)
result = numpy.sum(arr)
print(type(result)) # => <class 'cupy._core.core.ndarray'>
```

`cupy.ndarray` also implements `__array_function__` interface (see [NEP 18 — A dispatch mechanism for NumPy's high level array functions](#) for details). This enables code using NumPy to be directly operated on CuPy arrays. `__array_function__` feature requires NumPy 1.16 or later; As of NumPy 1.17, `__array_function__` is enabled by default.

## 4.7.3 Numba

Numba is a Python JIT compiler with NumPy support.

`cupy.ndarray` implements `__cuda_array_interface__`, which is the CUDA array interchange interface compatible with Numba v0.39.0 or later (see [CUDA Array Interface](#) for details). It means you can pass CuPy arrays to kernels JITted with Numba. The following is a simple example code borrowed from [numba/numba#2860](#):

```
import cupy
from numba import cuda

@cuda.jit
def add(x, y, out):
    start = cuda.grid(1)
    stride = cuda.gridsize(1)
    for i in range(start, x.shape[0], stride):
        out[i] = x[i] + y[i]

a = cupy.arange(10)
b = a * 2
out = cupy.zeros_like(a)

print(out) # => [0 0 0 0 0 0 0 0 0 0]

add[1, 32](a, b, out)

print(out) # => [ 0  3  6  9 12 15 18 21 24 27]
```

In addition, `cupy.asarray()` supports zero-copy conversion from Numba CUDA array to CuPy array.

```
import numpy
import numba
import cupy

x = numpy.arange(10) # type: numpy.ndarray
```

(continues on next page)

(continued from previous page)

```
x_numba = numba.cuda.to_device(x) # type: numba.cuda.cudadrv.devicearray.DeviceNDArray
x_cupy = cupy.asarray(x_numba) # type: cupy.ndarray
```

**Warning:** `__cuda_array_interface__` specifies that the object lifetime must be managed by the user, so it is an undefined behavior if the exported object is destroyed while still in use by the consumer library.

**Note:** CuPy has a few environment variables controlling the exchange behavior, see [Environment variables](#) for details.

#### 4.7.4 mpi4py

MPI for Python ([mpi4py](#)) is a Python wrapper for the Message Passing Interface (MPI) libraries.

MPI is the most widely used standard for high-performance inter-process communications. Recently several MPI vendors, including MPICH, Open MPI and MVAPICH, have extended their support beyond the MPI-3.1 standard to enable “CUDA-awareness”; that is, passing CUDA device pointers directly to MPI calls to avoid explicit data movement between the host and the device.

With the aforementioned `__cuda_array_interface__` standard implemented in CuPy, [mpi4py](#) now provides (experimental) support for passing CuPy arrays to MPI calls, provided that [mpi4py](#) is built against a CUDA-aware MPI implementation. The following is a simple example code borrowed from [mpi4py Tutorial](#):

```
# To run this script with N MPI processes, do
# mpiexec -n N python this_script.py

import cupy
from mpi4py import MPI

comm = MPI.COMM_WORLD
size = comm.Get_size()

# Allreduce
sendbuf = cupy.arange(10, dtype='i')
recvbuf = cupy.empty_like(sendbuf)
comm.Allreduce(sendbuf, recvbuf)
assert cupy.allclose(recvbuf, sendbuf*size)
```

This new feature will be officially released in [mpi4py 3.1.0](#). To try it out, please build [mpi4py](#) from source for the time being. See the [mpi4py website](#) for more information.

### 4.7.5 PyTorch

PyTorch is a machine learning framework that provides high-performance, differentiable tensor operations.

PyTorch also supports `__cuda_array_interface__`, so zero-copy data exchange between CuPy and PyTorch can be achieved at no cost. The only caveat is PyTorch by default creates CPU tensors, which do not have the `__cuda_array_interface__` property defined, and users need to ensure the tensor is already on GPU before exchanging.

```
>>> import cupy as cp
>>> import torch
>>>
>>> # convert a torch tensor to a cupy array
>>> a = torch.rand((4, 4), device='cuda')
>>> b = cp.asarray(a)
>>> b *= b
>>> b
array([[0.8215962 , 0.82399917, 0.65607935, 0.30354425],
       [0.422695  , 0.8367199 , 0.00208597, 0.18545236],
       [0.00226746, 0.46201342, 0.6833052 , 0.47549972],
       [0.5208748 , 0.6059282 , 0.1909013 , 0.5148635 ]], dtype=float32)
>>> a
tensor([[0.8216, 0.8240, 0.6561, 0.3035],
        [0.4227, 0.8367, 0.0021, 0.1855],
        [0.0023, 0.4620, 0.6833, 0.4755],
        [0.5209, 0.6059, 0.1909, 0.5149]], device='cuda:0')
>>> # check the underlying memory pointer is the same
>>> assert a.__cuda_array_interface__['data'][0] == b.__cuda_array_interface__['data'][0]
>>>
>>> # convert a cupy array to a torch tensor
>>> a = cp.arange(10)
>>> b = torch.as_tensor(a, device='cuda')
>>> b += 3
>>> b
tensor([ 3,  4,  5,  6,  7,  8,  9, 10, 11, 12], device='cuda:0')
>>> a
array([ 3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
>>> assert a.__cuda_array_interface__['data'][0] == b.__cuda_array_interface__['data'][0]
```

PyTorch also supports zero-copy data exchange through DLPack (see *DLPack* below):

```
import cupy
import torch

from torch.utils.dlpack import to_dlpack
from torch.utils.dlpack import from_dlpack

# Create a PyTorch tensor.
tx1 = torch.randn(1, 2, 3, 4).cuda()

# Convert it into a DLPack tensor.
dx = to_dlpack(tx1)

# Convert it into a CuPy array.
```

(continues on next page)

(continued from previous page)

```

cx = cupy.fromDlpack(dx)

# Convert it back to a PyTorch tensor.
tx2 = from_dlpack(cx.toDlpack())

```

`pytorch-pfn-extras` library provides additional integration features with PyTorch, including memory pool sharing and stream sharing:

```

>>> import cupy
>>> import torch
>>> import pytorch_pfn_extras as ppe
>>>
>>> # Perform CuPy memory allocation using the PyTorch memory pool.
>>> ppe.cuda.use_torch_mempool_in_cupy()
>>> torch.cuda.memory_allocated()
0
>>> arr = cupy.arange(10)
>>> torch.cuda.memory_allocated()
512
>>>
>>> # Change the default stream in PyTorch and CuPy:
>>> stream = torch.cuda.Stream()
>>> with ppe.cuda.stream(stream):
...     ...

```

#### 4.7.6 RMM

RMM (RAPIDS Memory Manager) provides highly configurable memory allocators.

RMM provides an interface to allow CuPy to allocate memory from the RMM memory pool instead of from CuPy's own pool. It can be set up as simple as:

```

import cupy
import rmm
cupy.cuda.set_allocator(rmm.rmm_cupy_allocator)

```

Sometimes, a more performant allocator may be desirable. RMM provides an option to switch the allocator:

```

import cupy
import rmm
rmm.reinitialize(pool_allocator=True) # can also set init pool size etc here
cupy.cuda.set_allocator(rmm.rmm_cupy_allocator)

```

For more information on CuPy's memory management, see *Memory Management*.

### 4.7.7 DLPack

DLPack is a specification of tensor structure to share tensors among frameworks.

CuPy supports importing from and exporting to DLPack data structure (`cupy.fromDlpack()` and `cupy.ndarray.toDlpack()`).

Here is a simple example:

```
import cupy

# Create a CuPy array.
cx1 = cupy.random.randn(1, 2, 3, 4).astype(cupy.float32)

# Convert it into a DLPack tensor.
dx = cx1.toDlpack()

# Convert it back to a CuPy array.
cx2 = cupy.fromDlpack(dx)
```

TensorFlow also supports DLPack, so zero-copy data exchange between CuPy and TensorFlow through DLPack is possible:

```
>>> import tensorflow as tf
>>> import cupy as cp
>>>
>>> # convert a TF tensor to a cupy array
>>> with tf.device('/GPU:0'):
...     a = tf.random.uniform((10,))
...
>>> a
<tf.Tensor: shape=(10,), dtype=float32, numpy=
array([[0.9672388 , 0.57568085, 0.53163004, 0.6536236 , 0.20479882,
        0.84908986, 0.5852566 , 0.30355775, 0.1733712 , 0.9177849 ],
       dtype=float32)>
>>> a.device
'/job:localhost/replica:0/task:0/device:GPU:0'
>>> cap = tf.experimental.dlpack.to_dlpack(a)
>>> b = cp.fromDlpack(cap)
>>> b *= 3
>>> b
array([[1.4949363 , 0.60699713, 1.3276931 , 1.5781245 , 1.1914308 ,
        2.3180873 , 1.9560868 , 1.3932796 , 1.9299742 , 2.5352407 ],
       dtype=float32)
>>> a
<tf.Tensor: shape=(10,), dtype=float32, numpy=
array([[1.4949363 , 0.60699713, 1.3276931 , 1.5781245 , 1.1914308 ,
        2.3180873 , 1.9560868 , 1.3932796 , 1.9299742 , 2.5352407 ],
       dtype=float32)>
>>>
>>> # convert a cupy array to a TF tensor
>>> a = cp.arange(10)
>>> cap = a.toDlpack()
>>> b = tf.experimental.dlpack.from_dlpack(cap)
>>> b.device
```

(continues on next page)



(continued from previous page)

```

'/job:localhost/replica:0/task:0/device:GPU:0'
>>> b
<tf.Tensor: shape=(10,), dtype=int64, numpy=array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])>
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

```

Be aware that in TensorFlow all tensors are immutable, so in the latter case any changes in `b` cannot be reflected in the CuPy array `a`.

Note that as of DLPack v0.4 for correctness it (implicitly) requires users to ensure that such conversion (both importing and exporting a CuPy array) must happen on the same CUDA/HIP stream. If in doubt, the current CuPy stream in use can be fetched by, for example, calling `cupy.cuda.get_current_stream()`. Please consult the other framework's documentation for how to access and control the streams. This requirement might be relaxed/changed in a future DLPack version.

## 4.8 Difference between CuPy and NumPy

The interface of CuPy is designed to obey that of NumPy. However, there are some differences.

### 4.8.1 Cast behavior from float to integer

Some casting behaviors from float to integer are not defined in C++ specification. The casting from a negative float to unsigned integer and infinity to integer is one of such examples. The behavior of NumPy depends on your CPU architecture. This is the result on an Intel CPU:

```

>>> np.array([-1], dtype=np.float32).astype(np.uint32)
array([4294967295], dtype=uint32)
>>> cupy.array([-1], dtype=np.float32).astype(np.uint32)
array([0], dtype=uint32)

```

```

>>> np.array([float('inf')], dtype=np.float32).astype(np.int32)
array([-2147483648], dtype=int32)
>>> cupy.array([float('inf')], dtype=np.float32).astype(np.int32)
array([2147483647], dtype=int32)

```

### 4.8.2 Random methods support dtype argument

NumPy's random value generator does not support a *dtype* argument and instead always returns a `float64` value. We support the option in CuPy because `cuRAND`, which is used in CuPy, supports both `float32` and `float64`.

```

>>> np.random.randn(dtype=np.float32)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: randn() got an unexpected keyword argument 'dtype'
>>> cupy.random.randn(dtype=np.float32)
array(0.10689262300729752, dtype=float32)

```

### 4.8.3 Out-of-bounds indices

CuPy handles out-of-bounds indices differently by default from NumPy when using integer array indexing. NumPy handles them by raising an error, but CuPy wraps around them.

```
>>> x = np.array([0, 1, 2])
>>> x[[1, 3]] = 10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: index 3 is out of bounds for axis 1 with size 3
>>> x = cupy.array([0, 1, 2])
>>> x[[1, 3]] = 10
>>> x
array([10, 10, 2])
```

### 4.8.4 Duplicate values in indices

CuPy's `__setitem__` behaves differently from NumPy when integer arrays reference the same location multiple times. In that case, the value that is actually stored is undefined. Here is an example of CuPy.

```
>>> a = cupy.zeros((2,))
>>> i = cupy.arange(10000) % 2
>>> v = cupy.arange(10000).astype(np.float32)
>>> a[i] = v
>>> a
array([ 9150.,  9151.])
```

NumPy stores the value corresponding to the last element among elements referencing duplicate locations.

```
>>> a_cpu = np.zeros((2,))
>>> i_cpu = np.arange(10000) % 2
>>> v_cpu = np.arange(10000).astype(np.float32)
>>> a_cpu[i_cpu] = v_cpu
>>> a_cpu
array([9998., 9999.])
```

### 4.8.5 Zero-dimensional array

#### Reduction methods

NumPy's reduction functions (e.g. `numpy.sum()`) return scalar values (e.g. `numpy.float32`). However CuPy counterparts return zero-dimensional `cupy.ndarray`s. That is because CuPy scalar values (e.g. `cupy.float32`) are aliases of NumPy scalar values and are allocated in CPU memory. If these types were returned, it would be required to synchronize between GPU and CPU. If you want to use scalar values, cast the returned arrays explicitly.

```
>>> type(np.sum(np.arange(3))) == np.int64
True
>>> type(cupy.sum(cupy.arange(3))) == cupy._core.core.ndarray
True
```

## Type promotion

CuPy automatically promotes dtypes of `cupy.ndarray`s in a function with two or more operands, the result dtype is determined by the dtypes of the inputs. This is different from NumPy's rule on type promotion, when operands contain zero-dimensional arrays. Zero-dimensional `numpy.ndarray`s are treated as if they were scalar values if they appear in operands of NumPy's function, This may affect the dtype of its output, depending on the values of the “scalar” inputs.

```
>>> (np.array(3, dtype=np.int32) * np.array([1., 2.], dtype=np.float32)).dtype
dtype('float32')
>>> (np.array(3000000, dtype=np.int32) * np.array([1., 2.], dtype=np.float32)).dtype
dtype('float64')
>>> (cupy.array(3, dtype=np.int32) * cupy.array([1., 2.], dtype=np.float32)).dtype
dtype('float64')
```

### 4.8.6 Data types

Data type of CuPy arrays cannot be non-numeric like strings or objects. See [Overview](#) for details.

### 4.8.7 Universal Functions only work with CuPy array or scalar

Unlike NumPy, Universal Functions in CuPy only work with CuPy array or scalar. They do not accept other objects (e.g., lists or `numpy.ndarray`).

```
>>> np.power([np.arange(5)], 2)
array([[ 0,  1,  4,  9, 16]])
```

```
>>> cupy.power([cupy.arange(5)], 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Unsupported type <class 'list'>
```

### 4.8.8 Random seed arrays are hashed to scalars

Like Numpy, CuPy's RandomState objects accept seeds either as numbers or as full numpy arrays.

```
>>> seed = np.array([1, 2, 3, 4, 5])
>>> rs = cupy.random.RandomState(seed=seed)
```

However, unlike Numpy, array seeds will be hashed down to a single number and so may not communicate as much entropy to the underlying random number generator.

### 4.8.9 NaN (not-a-number) handling

By default CuPy's reduction functions (e.g., `cupy.sum()`) handle NaNs in complex numbers differently from NumPy's counterparts:

```
>>> a = [0.5 + 3.7j, complex(0.7, np.nan), complex(np.nan, -3.9), complex(np.nan, np.
↳ nan)]
>>>
>>> a_np = np.asarray(a)
>>> print(a_np.max(), a_np.min())
(0.7+nanj) (0.7+nanj)
>>>
>>> a_cp = cp.asarray(a_np)
>>> print(a_cp.max(), a_cp.min())
(nan-3.9j) (nan-3.9j)
```

The reason is that internally the reduction is performed in a strided fashion, thus it does not ensure a proper comparison order and cannot follow NumPy's rule to always propagate the first-encountered NaN.

## 4.9 API Compatibility Policy

This document expresses the design policy on compatibilities of CuPy APIs. Development team should obey this policy on deciding to add, extend, and change APIs and their behaviors.

This document is written for both users and developers. Users can decide the level of dependencies on CuPy's implementations in their codes based on this document. Developers should read through this document before creating pull requests that contain changes on the interface. Note that this document may contain ambiguities on the level of supported compatibilities.

### 4.9.1 Versioning and Backward Compatibilities

The updates of CuPy are classified into three levels: major, minor, and revision. These types have distinct levels of backward compatibilities.

- **Major update** contains disruptive changes that break the backward compatibility.
- **Minor update** contains additions and extensions to the APIs that keep the backward compatibility supported.
- **Revision update** contains improvements on the API implementations without changing any API specifications.

Note that we do not support full backward compatibility, which is almost infeasible for Python-based APIs, since there is no way to completely hide the implementation details.

### 4.9.2 Processes to Break Backward Compatibilities

#### Deprecation, Dropping, and Its Preparation

Any APIs may be *deprecated* at some minor updates. In such a case, the deprecation note is added to the API documentation, and the API implementation is changed to fire a deprecation warning (if possible). There should be another way to reimplement the same functionality previously written using the deprecated APIs.

Any APIs may be marked as *to be dropped in the future*. In such a case, the dropping is stated in the documentation with the major version number on which the API is planned to be dropped, and the API implementation is changed to fire a future warning (if possible).

The actual dropping should be done through the following steps:

- Make the API deprecated. At this point, users should not use the deprecated API in their new application codes.
- After that, mark the API as *to be dropped in the future*. It must be done in the minor update different from that of the deprecation.
- At the major version announced in the above update, drop the API.

Consequently, it takes at least two minor versions to drop any APIs after the first deprecation.

## API Changes and Its Preparation

Any APIs may be marked as *to be changed in the future* for changes without backward compatibility. In such a case, the change is stated in the documentation with the version number on which the API is planned to be changed, and the API implementation is changed to fire the future warning on the certain usages.

The actual change should be done in the following steps:

- Announce that the API will be changed in the future. At this point, the actual version of change need not be accurate.
- After the announcement, mark the API as *to be changed in the future* with version number of planned changes. At this point, users should not use the marked API in their new application codes.
- At the major update announced in the above update, change the API.

## 4.9.3 Supported Backward Compatibility

This section defines backward compatibilities that minor updates must maintain.

### Documented Interface

CuPy has an official API documentation. Many applications can be written based on the documented features. We support backward compatibilities of documented features. In other words, codes only based on the documented features run correctly with minor-/revision- updated versions.

Developers are encouraged to use apparent names for objects of implementation details. For example, attributes outside of the documented APIs should have one or more underscores at the prefix of their names.

### Undocumented behaviors

Behaviors of CuPy implementation not stated in the documentation are undefined. Undocumented behaviors are not guaranteed to be stable between different minor/revision versions.

Minor update may contain changes to undocumented behaviors. For example, suppose an API X is added at the minor update. In the previous version, attempts to use X cause `AttributeError`. This behavior is not stated in the documentation, so this is undefined. Thus, adding the API X in minor version is permissible.

Revision update may also contain changes to undefined behaviors. Typical example is a bug fix. Another example is an improvement on implementation, which may change the internal object structures not shown in the documentation. As a consequence, **even revision updates do not support compatibility of pickling, unless the full layout of pickled objects is clearly documented.**

## Documentation Error

Compatibility is basically determined based on the documentation, though it sometimes contains errors. It may make the APIs confusing to assume the documentation always stronger than the implementations. We therefore may fix the documentation errors in any updates that may break the compatibility in regard to the documentation.

---

**Note:** Developers MUST NOT fix the documentation and implementation of the same functionality at the same time in revision updates as “bug fix”. Such a change completely breaks the backward compatibility. If you want to fix the bugs in both sides, first fix the documentation to fit it into the implementation, and start the API changing procedure described above.

---

## Object Attributes and Properties

Object attributes and properties are sometimes replaced by each other at minor updates. It does not break the user codes, except for the codes depending on how the attributes and properties are implemented.

## Functions and Methods

Methods may be replaced by callable attributes keeping the compatibility of parameters and return values in minor updates. It does not break the user codes, except for the codes depending on how the methods and callable attributes are implemented.

## Exceptions and Warnings

The specifications of raising exceptions are considered as a part of standard backward compatibilities. No exception is raised in the future versions with correct usages that the documentation allows, unless the API changing process is completed.

On the other hand, warnings may be added at any minor updates for any APIs. It means minor updates do not keep backward compatibility of warnings.

## 4.9.4 Installation Compatibility

The installation process is another concern of compatibilities. We support environmental compatibilities in the following ways.

- Any changes of dependent libraries that force modifications on the existing environments must be done in major updates. Such changes include following cases:
  - dropping supported versions of dependent libraries (e.g. dropping cuDNN v2)
  - adding new mandatory dependencies (e.g. adding h5py to setup\_requires)
- Supporting optional packages/libraries may be done in minor updates (e.g. supporting h5py in optional features).

---

**Note:** The installation compatibility does not guarantee that all the features of CuPy correctly run on supported environments. It may contain bugs that only occurs in certain environments. Such bugs should be fixed in some updates.

---

## API REFERENCE

- `genindex`
  - `modindex`
- 

### 5.1 The N-dimensional array (ndarray)

`cupy.ndarray` is the CuPy counterpart of NumPy `numpy.ndarray`. It provides an intuitive interface for a fixed-size multidimensional array which resides in a CUDA device.

For the basic concept of ndarrays, please refer to the [NumPy documentation](#).

---

<code>cupy.ndarray(shape[, dtype, memptr, ...])</code>	Multi-dimensional array on a CUDA device.
--	---

---

#### 5.1.1 `cupy.ndarray`

**class** `cupy.ndarray(shape, dtype=float, memptr=None, strides=None, order='C')`

Multi-dimensional array on a CUDA device.

This class implements a subset of methods of `numpy.ndarray`. The difference is that this class allocates the array content on the current GPU device.

##### Parameters

- **shape** (*tuple of ints*) – Length of axes.
- **dtype** – Data type. It must be an argument of `numpy.dtype`.
- **memptr** (`cupy.cuda.MemoryPointer`) – Pointer to the array content head.
- **strides** (*tuple of ints or None*) – Strides of data in memory.
- **order** (`{'C', 'F'}`) – Row-major (C-style) or column-major (Fortran-style) order.

##### Variables

- **base** (*None or cupy.ndarray*) – Base array from which this array is created as a view.
- **data** (`cupy.cuda.MemoryPointer`) – Pointer to the array content head.
- **dtype** (`numpy.dtype`) – Dtype object of element type.

**See also:**

[Data type objects \(dtype\)](#)

- **size** (*int*) – Number of elements this array holds.

This is equivalent to product over the shape tuple.

**See also:**

`numpy.ndarray.size`

## Methods

### `__getitem__()`

`x.__getitem__(y) <==> x[y]`

Supports both basic and advanced indexing.

---

**Note:** Currently, it does not support `slices` that consists of more than one boolean arrays

---

---

**Note:** CuPy handles out-of-bounds indices differently from NumPy. NumPy handles them by raising an error, but CuPy wraps around them.

---

## Example

```
>>> a = cupy.arange(3)
>>> a[[1, 3]]
array([1, 0])
```

### `__setitem__()`

`x.__setitem__(slices, y) <==> x[slices] = y`

Supports both basic and advanced indexing.

---

**Note:** Currently, it does not support `slices` that consists of more than one boolean arrays

---

---

**Note:** CuPy handles out-of-bounds indices differently from NumPy when using integer array indexing. NumPy handles them by raising an error, but CuPy wraps around them.

---

```
>>> import cupy
>>> x = cupy.arange(3)
>>> x[[1, 3]] = 10
>>> x
array([10, 10,  2])
```

---

**Note:** The behavior differs from NumPy when integer arrays in `slices` reference the same location multiple times. In that case, the value that is actually stored is undefined.

---

```
>>> import cupy
>>> a = cupy.zeros((2,))
```

(continues on next page)



(continued from previous page)

```
>>> i = cupy.arange(10000) % 2
>>> v = cupy.arange(10000).astype(cupy.float_)
>>> a[i] = v
>>> a
array([9150., 9151.] )
```

On the other hand, NumPy stores the value corresponding to the last index among the indices referencing duplicate locations.

```
>>> import numpy
>>> a_cpu = numpy.zeros((2,))
>>> i_cpu = numpy.arange(10000) % 2
>>> v_cpu = numpy.arange(10000).astype(numpy.float_)
>>> a_cpu[i_cpu] = v_cpu
>>> a_cpu
array([9998., 9999.] )
```

**\_\_len\_\_()**

Return len(self).

**\_\_iter\_\_()**

Implement iter(self).

**\_\_copy\_\_(self)**

**all**(self, axis=None, out=None, keepdims=False) → ndarray

**any**(self, axis=None, out=None, keepdims=False) → ndarray

**argmax**(self, axis=None, out=None, dtype=None, keepdims=False) → ndarray

Returns the indices of the maximum along a given axis.

---

**Note:** dtype and keepdim arguments are specific to CuPy. They are not in NumPy.

---



---

**Note:** axis argument accepts a tuple of ints, but this is specific to CuPy. NumPy does not support it.

---

**See also:**

[cupy.argmax\(\)](#) for full documentation, [numpy.ndarray.argmax\(\)](#)

**argmin**(self, axis=None, out=None, dtype=None, keepdims=False) → ndarray

Returns the indices of the minimum along a given axis.

---

**Note:** dtype and keepdim arguments are specific to CuPy. They are not in NumPy.

---



---

**Note:** axis argument accepts a tuple of ints, but this is specific to CuPy. NumPy does not support it.

---

**See also:**

[cupy.argmin\(\)](#) for full documentation, [numpy.ndarray.argmin\(\)](#)

**argpartition**(*self*, *kth*, *axis=-1*) → *ndarray*

Returns the indices that would partially sort an array.

**Parameters**

- **kth** (*int* or *sequence of ints*) – Element index to partition by. If supplied with a sequence of k-th it will partition all elements indexed by k-th of them into their sorted position at once.
- **axis** (*int* or *None*) – Axis along which to sort. Default is -1, which means sort along the last axis. If None is supplied, the array is flattened before sorting.

**Returns** Array of the same type and shape as a.

**Return type** *cupy.ndarray*

**See also:**

*cupy.argpartition()* for full documentation, *numpy.ndarray.argpartition()*

**argsort**(*self*, *axis=-1*) → *ndarray*

Returns the indices that would sort an array with stable sorting

**Parameters** **axis** (*int* or *None*) – Axis along which to sort. Default is -1, which means sort along the last axis. If None is supplied, the array is flattened before sorting.

**Returns** Array of indices that sort the array.

**Return type** *cupy.ndarray*

**See also:**

*cupy.argsort()* for full documentation, *numpy.ndarray.argsort()*

**astype**(*self*, *dtype*, *order='K'*, *casting=None*, *subok=None*, *copy=True*) → *ndarray*

Casts the array to given data type.

**Parameters**

- **dtype** – Type specifier.
- **order** ({'C', 'F', 'A', 'K'}) – Row-major (C-style) or column-major (Fortran-style) order. When order is 'A', it uses 'F' if a is column-major and uses 'C' otherwise. And when order is 'K', it keeps strides as closely as possible.
- **copy** (*bool*) – If it is False and no cast happens, then this method returns the array itself. Otherwise, a copy is returned.

**Returns** If copy is False and no cast is required, then the array itself is returned. Otherwise, it returns a (possibly casted) copy of the array.

---

**Note:** This method currently does not support casting, and subok arguments.

---

**See also:**

*numpy.ndarray.astype()*

**choose**(*self*, *choices*, *out=None*, *mode='raise'*)

**clip**(*self*, *a\_min=None*, *a\_max=None*, *out=None*) → *ndarray*

Returns an array with values limited to [a\_min, a\_max].

**See also:**

*cupy.clip()* for full documentation, *numpy.ndarray.clip()*

**compress**(*self*, *condition*, *axis=None*, *out=None*) → *ndarray*  
Returns selected slices of this array along given axis.

**Warning:** This function may synchronize the device.

**See also:**

[`cupy.compress\(\)`](#) for full documentation, `numpy.ndarray.compress()`

**conj**(*self*) → *ndarray*

**conjugate**(*self*) → *ndarray*

**copy**(*self*, *order='C'*) → *ndarray*

Returns a copy of the array.

This method makes a copy of a given array in the current device. Even when a given array is located in another device, you can copy it to the current device.

**Parameters** **order** (`{'C', 'F', 'A', 'K'}`) – Row-major (C-style) or column-major (Fortran-style) order. When order is 'A', it uses 'F' if a is column-major and uses 'C' otherwise. And when order is 'K', it keeps strides as closely as possible.

**See also:**

[`cupy.copy\(\)`](#) for full documentation, `numpy.ndarray.copy()`

**cumprod**(*self*, *axis=None*, *dtype=None*, *out=None*) → *ndarray*

Returns the cumulative product of an array along a given axis.

**See also:**

[`cupy.cumprod\(\)`](#) for full documentation, `numpy.ndarray.cumprod()`

**cumsum**(*self*, *axis=None*, *dtype=None*, *out=None*) → *ndarray*

Returns the cumulative sum of an array along a given axis.

**See also:**

[`cupy.cumsum\(\)`](#) for full documentation, `numpy.ndarray.cumsum()`

**diagonal**(*self*, *offset=0*, *axis1=0*, *axis2=1*) → *ndarray*

Returns a view of the specified diagonals.

**See also:**

[`cupy.diagonal\(\)`](#) for full documentation, `numpy.ndarray.diagonal()`

**dot**(*self*, *ndarray b*, *ndarray out=None*)

Returns the dot product with given array.

**See also:**

[`cupy.dot\(\)`](#) for full documentation, `numpy.ndarray.dot()`

**dump**(*self*, *file*)

Dumps a pickle of the array to a file.

Dumped file can be read back to `cupy.ndarray` by `cupy.load()`.

**dumps**(*self*) → *bytes*

Dumps a pickle of the array to a string.

**fill**(*self*, *value*)

Fills the array with a scalar value.

**Parameters** **value** – A scalar value to fill the array content.

**See also:**

`numpy.ndarray.fill()`

**flatten**(*self*) → *ndarray*

Returns a copy of the array flatten into one dimension.

It currently supports C-order only.

**Returns** A copy of the array with one dimension.

**Return type** *cupy.ndarray*

**See also:**

`numpy.ndarray.flatten()`

**get**(*self*, *stream=None*, *order='C'*, *out=None*)

Returns a copy of the array on host memory.

**Parameters**

- **stream** (*cupy.cuda.Stream*) – CUDA stream object. If it is given, the copy runs asynchronously. Otherwise, the copy is synchronous. The default uses CUDA stream object of the current context.
- **order** ({'C', 'F', 'A'}) – The desired memory layout of the host array. When order is 'A', it uses 'F' if the array is fortran-contiguous and 'C' otherwise. The order will be ignored if out is specified.
- **out** (*numpy.ndarray*) – Output array. In order to enable asynchronous copy, the underlying memory should be a pinned memory.

**Returns** Copy of the array on host memory.

**Return type** *numpy.ndarray*

**item**(*self*)

Converts the array with one element to a Python scalar

**Returns** The element of the array.

**Return type** *int* or *float* or *complex*

**See also:**

`numpy.ndarray.item()`

**max**(*self*, *axis=None*, *out=None*, *keepdims=False*) → *ndarray*

Returns the maximum along a given axis.

**See also:**

*cupy.amax()* for full documentation, `numpy.ndarray.max()`

**mean**(*self*, *axis=None*, *dtype=None*, *out=None*, *keepdims=False*) → *ndarray*

Returns the mean along a given axis.

**See also:**

*cupy.mean()* for full documentation, `numpy.ndarray.mean()`

**min**(*self*, *axis=None*, *out=None*, *keepdims=False*) → *ndarray*

Returns the minimum along a given axis.

See also:

[`cupy.amin\(\)`](#) for full documentation, [`numpy.ndarray.min\(\)`](#)

**nonzero**(*self*) → *tuple*

Return the indices of the elements that are non-zero.

Returned Array is containing the indices of the non-zero elements in that dimension.

**Returns** Indices of elements that are non-zero.

**Return type** tuple of arrays

**Warning:** This function may synchronize the device.

See also:

[`numpy.nonzero\(\)`](#)

**partition**(*self*, *kth*, *int axis=-1*)

Partitions an array.

**Parameters**

- **kth** (*int* or *sequence of ints*) – Element index to partition by. If supplied with a sequence of k-th it will partition all elements indexed by k-th of them into their sorted position at once.
- **axis** (*int*) – Axis along which to sort. Default is -1, which means sort along the last axis.

See also:

[`cupy.partition\(\)`](#) for full documentation, [`numpy.ndarray.partition\(\)`](#)

**prod**(*self*, *axis=None*, *dtype=None*, *out=None*, *keepdims=None*) → *ndarray*

Returns the product along a given axis.

See also:

[`cupy.prod\(\)`](#) for full documentation, [`numpy.ndarray.prod\(\)`](#)

**ptp**(*self*, *axis=None*, *out=None*, *keepdims=False*) → *ndarray*

Returns (maximum - minimum) along a given axis.

See also:

[`cupy.ptp\(\)`](#) for full documentation, [`numpy.ndarray.ptp\(\)`](#)

**put**(*self*, *indices*, *values*, *mode='wrap'*)

Replaces specified elements of an array with given values.

See also:

[`cupy.put\(\)`](#) for full documentation, [`numpy.ndarray.put\(\)`](#)

**ravel**(*self*, *order='C'*) → *ndarray*

Returns an array flattened into one dimension.

See also:

[`cupy.ravel\(\)`](#) for full documentation, [`numpy.ndarray.ravel\(\)`](#)

**reduced\_view**(*self*, *dtype=None*) → *ndarray*

Returns a view of the array with minimum number of dimensions.

**Parameters** **dtype** – (Deprecated) Data type specifier. If it is given, then the memory sequence is reinterpreted as the new type.

**Returns** A view of the array with reduced dimensions.

**Return type** *cupy.ndarray*

**repeat**(*self*, *repeats*, *axis=None*)

Returns an array with repeated arrays along an axis.

**See also:**

*cupy.repeat()* for full documentation, *numpy.ndarray.repeat()*

**reshape**(*self*, *\*shape*, *order='C'*)

Returns an array of a different shape and the same content.

**See also:**

*cupy.reshape()* for full documentation, *numpy.ndarray.reshape()*

**round**(*self*, *decimals=0*, *out=None*) → *ndarray*

Returns an array with values rounded to the given number of decimals.

**See also:**

*cupy.around()* for full documentation, *numpy.ndarray.round()*

**scatter\_add**(*self*, *slices*, *value*)

Adds given values to specified elements of an array.

**See also:**

*cupyx.scatter\_add()* for full documentation.

**scatter\_max**(*self*, *slices*, *value*)

Stores a maximum value of elements specified by indices to an array.

**See also:**

*cupyx.scatter\_max()* for full documentation.

**scatter\_min**(*self*, *slices*, *value*)

Stores a minimum value of elements specified by indices to an array.

**See also:**

*cupyx.scatter\_min()* for full documentation.

**set**(*self*, *arr*, *stream=None*)

Copies an array on the host memory to *cupy.ndarray*.

**Parameters**

- **arr** (*numpy.ndarray*) – The source array on the host memory.
- **stream** (*cupy.cuda.Stream*) – CUDA stream object. If it is given, the copy runs asynchronously. Otherwise, the copy is synchronous. The default uses CUDA stream object of the current context.

**sort**(*self*, *int axis=-1*)

Sort an array, in-place with a stable sorting algorithm.

**Parameters** **axis** (*int*) – Axis along which to sort. Default is -1, which means sort along the last axis.

---

**Note:** For its implementation reason, `ndarray.sort` currently supports only arrays with their own data, and does not support `kind` and `order` parameters that `numpy.ndarray.sort` does support.

---

**See also:**

[`cupy.sort\(\)`](#) for full documentation, `numpy.ndarray.sort()`

**squeeze**(*self*, *axis=None*) → *ndarray*

Returns a view with size-one axes removed.

**See also:**

[`cupy.squeeze\(\)`](#) for full documentation, `numpy.ndarray.squeeze()`

**std**(*self*, *axis=None*, *dtype=None*, *out=None*, *ddof=0*, *keepdims=False*) → *ndarray*

Returns the standard deviation along a given axis.

**See also:**

[`cupy.std\(\)`](#) for full documentation, `numpy.ndarray.std()`

**sum**(*self*, *axis=None*, *dtype=None*, *out=None*, *keepdims=False*) → *ndarray*

Returns the sum along a given axis.

**See also:**

[`cupy.sum\(\)`](#) for full documentation, `numpy.ndarray.sum()`

**swapaxes**(*self*, *Py\_ssize\_t axis1*, *Py\_ssize\_t axis2*) → *ndarray*

Returns a view of the array with two axes swapped.

**See also:**

[`cupy.swapaxes\(\)`](#) for full documentation, `numpy.ndarray.swapaxes()`

**take**(*self*, *indices*, *axis=None*, *out=None*) → *ndarray*

Returns an array of elements at given indices along the axis.

**See also:**

[`cupy.take\(\)`](#) for full documentation, `numpy.ndarray.take()`

**toDlpack**(*self*)

Zero-copy conversion to a DLPack tensor.

DLPack is a open in memory tensor structure proposed in this repository: [dmlc/dlpack](#).

This function returns a `PyCapsule` object which contains a pointer to a DLPack tensor converted from the own `ndarray`. This function does not copy the own data to the output DLPack tensor but it shares the pointer which is pointing to the same memory region for the data.

**Returns** Output DLPack tensor which is encapsulated in a `PyCapsule` object.

**Return type** `dlpack.DlpackTensor` (`PyCapsule`)

**See also:**

[`fromDlpack\(\)`](#) is a method for zero-copy conversion from a DLPack tensor (which is encapsulated in a `PyCapsule` object) to a *ndarray*

**Warning:** As of the DLPack v0.3 specification, it is (implicitly) assumed that the user is responsible to ensure the Producer and the Consumer are operating on the same stream. This requirement might be relaxed/changed in a future DLPack version.

---

### Example

```
>>> import cupy
>>> array1 = cupy.array([0, 1, 2], dtype=cupy.float32)
>>> dltensor = array1.toDlpack()
>>> array2 = cupy.fromDlpack(dltensor)
>>> cupy.testing.assert_array_equal(array1, array2)
```

**tobytes**(*self*, *order*='C') → bytes

Turns the array into a Python bytes object.

**tofile**(*self*, *fid*, *sep*=", *format*='%s')

Writes the array to a file.

**See also:**

[numpy.ndarray.tofile\(\)](#)

**tolist**(*self*)

Converts the array to a (possibly nested) Python list.

**Returns** The possibly nested Python list of array elements.

**Return type** list

**See also:**

[numpy.ndarray.tolist\(\)](#)

**trace**(*self*, *offset*=0, *axis1*=0, *axis2*=1, *dtype*=None, *out*=None) → ndarray

Returns the sum along diagonals of the array.

**See also:**

[cupy.trace\(\)](#) for full documentation, [numpy.ndarray.trace\(\)](#)

**transpose**(*self*, \**axes*)

Returns a view of the array with axes permuted.

**See also:**

[cupy.transpose\(\)](#) for full documentation, [numpy.ndarray.reshape\(\)](#)

**var**(*self*, *axis*=None, *dtype*=None, *out*=None, *ddof*=0, *keepdims*=False) → ndarray

Returns the variance along a given axis.

**See also:**

[cupy.var\(\)](#) for full documentation, [numpy.ndarray.var\(\)](#)

**view**(*self*, *dtype*=None) → ndarray

Returns a view of the array.

**Parameters** **dtype** – If this is different from the data type of the array, the returned view reinterprets the memory sequence as an array of this type.

**Returns** A view of the array. A reference to the original array is stored at the [base](#) attribute.



**Return type** *cupy.ndarray*

**See also:**

`numpy.ndarray.view()`

`__eq__(value, /)`

Return self==value.

`__ne__(value, /)`

Return self!=value.

`__lt__(value, /)`

Return self<value.

`__le__(value, /)`

Return self<=value.

`__gt__(value, /)`

Return self>value.

`__ge__(value, /)`

Return self>=value.

`__bool__()`

self != 0

## Attributes

**T**

Shape-reversed view of the array.

If ndim < 2, then this is just a reference to the array itself.

**base**

**cstruct**

C representation of the array.

This property is used for sending an array to CUDA kernels. The type of returned C structure is different for different dtypes and ndims. The definition of C type is written in `cupy/carray.cuh`.

**data**

**device**

CUDA device on which this array resides.

**dtype**

**flags**

Object containing memory-layout information.

It only contains `c_contiguous`, `f_contiguous`, and `owndata` attributes. All of these are read-only. Accessing by indexes is also supported.

**See also:**

`numpy.ndarray.flags`

**flat**

**imag**

**itemsize**

Size of each element in bytes.

**See also:**

`numpy.ndarray.itemsize`

**nbytes**

Total size of all elements in bytes.

It does not count skips between elements.

**See also:**

`numpy.ndarray.nbytes`

**ndim**

Number of dimensions.

`a.ndim` is equivalent to `len(a.shape)`.

**See also:**

`numpy.ndarray.ndim`

**real****shape**

Lengths of axes.

Setter of this property involves reshaping without copy. If the array cannot be reshaped without copy, it raises an exception.

**size****strides**

Strides of axes in bytes.

**See also:**

`numpy.ndarray.strides`

## 5.1.2 Conversion to/from NumPy arrays

`cupy.ndarray` and `numpy.ndarray` are not implicitly convertible to each other. That means, NumPy functions cannot take `cupy.ndarrays` as inputs, and vice versa.

- To convert `numpy.ndarray` to `cupy.ndarray`, use `cupy.array()` or `cupy.asarray()`.
- To convert `cupy.ndarray` to `numpy.ndarray`, use `cupy.asnumpy()` or `cupy.ndarray.get()`.

Note that converting between `cupy.ndarray` and `numpy.ndarray` incurs data transfer between the host (CPU) device and the GPU device, which is costly in terms of performance.

<code>cupy.array(obj[, dtype, copy, order, subok, ...])</code>	Creates an array on the current device.
<code>cupy.asarray(a[, dtype, order])</code>	Converts an object to array.
<code>cupy.asnumpy(a[, stream, order])</code>	Returns an array on the host memory from an arbitrary source array.

## cupy.array

`cupy.array(obj, dtype=None, copy=True, order='K', subok=False, ndmin=0)`

Creates an array on the current device.

This function currently does not support the `subok` option.

### Parameters

- **obj** – `cupy.ndarray` object or any other object that can be passed to `numpy.array()`.
- **dtype** – Data type specifier.
- **copy** (*bool*) – If `False`, this function returns `obj` if possible. Otherwise this function always returns a new array.
- **order** (`{'C', 'F', 'A', 'K'}`) – Row-major (C-style) or column-major (Fortran-style) order. When `order` is `'A'`, it uses `'F'` if `a` is column-major and uses `'C'` otherwise. And when `order` is `'K'`, it keeps strides as closely as possible. If `obj` is `numpy.ndarray`, the function returns `'C'` or `'F'` order array.
- **subok** (*bool*) – If `True`, then sub-classes will be passed-through, otherwise the returned array will be forced to be a base-class array (default).
- **ndmin** (*int*) – Minimum number of dimensions. Ones are inserted to the head of the shape if needed.

**Returns** An array on the current device.

**Return type** `cupy.ndarray`

---

**Note:** This method currently does not support `subok` argument.

---

See also:

`numpy.array()`

## cupy.asarray

`cupy.asarray(a, dtype=None, order=None)`

Converts an object to array.

This is equivalent to `array(a, dtype, copy=False)`. This function currently does not support the `order` option.

### Parameters

- **a** – The source object.
- **dtype** – Data type specifier. It is inferred from the input by default.
- **order** (`{'C', 'F'}`) – Whether to use row-major (C-style) or column-major (Fortran-style) memory representation. Defaults to `'C'`. `order` is ignored for objects that are not `cupy.ndarray`, but have the `__cuda_array_interface__` attribute.

**Returns** An array on the current device. If `a` is already on the device, no copy is performed.

**Return type** `cupy.ndarray`

See also:

`numpy.asarray()`

## cupy.asnumpy

`cupy.asnumpy(a, stream=None, order='C')`

Returns an array on the host memory from an arbitrary source array.

### Parameters

- **a** – Arbitrary object that can be converted to `numpy.ndarray`.
- **stream** (`cupy.cuda.Stream`) – CUDA stream object. If it is specified, then the device-to-host copy runs asynchronously. Otherwise, the copy is synchronous. Note that if `a` is not a `cupy.ndarray` object, then this argument has no effect.
- **order** (`{'C', 'F', 'A'}`) – The desired memory layout of the host array. When `order` is `'A'`, it uses `'F'` if `a` is fortran-contiguous and `'C'` otherwise.

**Returns** Converted array on the host memory.

**Return type** `numpy.ndarray`

## 5.1.3 Code compatibility features

`cupy.ndarray` is designed to be interchangeable with `numpy.ndarray` in terms of code compatibility as much as possible. But occasionally, you will need to know whether the arrays you're handling are `cupy.ndarray` or `numpy.ndarray`. One example is when invoking module-level functions such as `cupy.sum()` or `numpy.sum()`. In such situations, `cupy.get_array_module()` can be used.

---

<code>cupy.get_array_module(*args)</code>	Returns the array module for arguments.
---	---

---

## cupy.get\_array\_module

`cupy.get_array_module(*args)`

Returns the array module for arguments.

This function is used to implement CPU/GPU generic code. If at least one of the arguments is a `cupy.ndarray` object, the `cupy` module is returned.

**Parameters** `args` – Values to determine whether NumPy or CuPy should be used.

**Returns** `cupy` or `numpy` is returned based on the types of the arguments.

**Return type** module

---

### Example

A NumPy/CuPy generic function can be written as follows

```
>>> def softplus(x):
...     xp = cupy.get_array_module(x)
...     return xp.maximum(0, x) + xp.log1p(xp.exp(-abs(x)))
```

---

<code>cupyx.scipy.get_array_module(*args)</code>	Returns the array module for arguments.
--	---

---

## cupyx.scipy.get\_array\_module

`cupyx.scipy.get_array_module(*args)`

Returns the array module for arguments.

This function is used to implement CPU/GPU generic code. If at least one of the arguments is a `cupy.ndarray` object, the `cupyx.scipy` module is returned.

**Parameters** `args` – Values to determine whether NumPy or CuPy should be used.

**Returns** `cupyx.scipy` or `scipy` is returned based on the types of the arguments.

**Return type** module

## 5.2 Universal functions (cupy.ufunc)

---

**Hint:** NumPy API Reference: Universal functions (`numpy.ufunc`)

---

CuPy provides universal functions (a.k.a. ufuncs) to support various elementwise operations. CuPy's ufunc supports following features of NumPy's one:

- Broadcasting
- Output type determination
- Casting rules

CuPy's ufunc currently does not provide methods such as `reduce`, `accumulate`, `reduceat`, `outer`, and `at`.

### 5.2.1 ufunc

---

<code>ufunc(name, nin, nout, _Ops ops[, preamble, ...])</code>	Universal function.
--	---------------------

---

## cupy.ufunc

**class** `cupy.ufunc(name, nin, nout, _Ops ops, preamble=u'', loop_prep=u'', doc=u'', default_casting=None, _Ops out_ops=None, *)`

Universal function.

### Variables

- **name** (`str`) – The name of the universal function.
- **nin** (`int`) – Number of input arguments.
- **nout** (`int`) – Number of output arguments.
- **nargs** (`int`) – Number of all arguments.

## Methods

### `__call__()`

Applies the universal function to arguments elementwise.

#### Parameters

- **args** – Input arguments. Each of them can be a `cupy.ndarray` object or a scalar. The output arguments can be omitted or be specified by the `out` argument.
- **out** (`cupy.ndarray`) – Output array. It outputs to new arrays default.
- **dtype** – Data type specifier.

**Returns** Output array or a tuple of output arrays.

### `__eq__(value, /)`

Return `self==value`.

### `__ne__(value, /)`

Return `self!=value`.

### `__lt__(value, /)`

Return `self<value`.

### `__le__(value, /)`

Return `self<=value`.

### `__gt__(value, /)`

Return `self>value`.

### `__ge__(value, /)`

Return `self>=value`.

## Attributes

**name**

**nargs**

**nin**

**nout**

**types**

A list of type signatures.

Each type signature is represented by type character codes of inputs and outputs separated by ‘->’.

## 5.2.2 Available ufuncs

### Math operations

<code>add</code>	Adds two arrays elementwise.
<code>subtract</code>	Subtracts arguments elementwise.
<code>multiply</code>	Multiplies two arrays elementwise.
<code>matmul(x1, x2[, out, axes])</code>	Matrix product of two arrays.
<code>divide</code>	Elementwise true division (i.e.

continues on next page

Table 6 – continued from previous page

<i>logaddexp</i>	Computes $\log(\exp(x1) + \exp(x2))$ elementwise.
<i>logaddexp2</i>	Computes $\log_2(\exp_2(x1) + \exp_2(x2))$ elementwise.
<i>true_divide</i>	Elementwise true division (i.e.
<i>floor_divide</i>	Elementwise floor division (i.e.
<i>negative</i>	Takes numerical negative elementwise.
<i>power</i>	Computes $x1 ** x2$ elementwise.
<i>remainder</i>	Computes the remainder of Python division elementwise.
<i>mod</i>	Computes the remainder of Python division elementwise.
<i>fmod</i>	Computes the remainder of C division elementwise.
<i>absolute</i>	Elementwise absolute value function.
<i>rint</i>	Rounds each element of an array to the nearest integer.
<i>sign</i>	Elementwise sign function.
<i>conj</i>	Returns the complex conjugate, element-wise.
<i>conjugate</i>	Returns the complex conjugate, element-wise.
<i>exp</i>	Elementwise exponential function.
<i>exp2</i>	Elementwise exponentiation with base 2.
<i>log</i>	Elementwise natural logarithm function.
<i>log2</i>	Elementwise binary logarithm function.
<i>log10</i>	Elementwise common logarithm function.
<i>expm1</i>	Computes $\exp(x) - 1$ elementwise.
<i>log1p</i>	Computes $\log(1 + x)$ elementwise.
<i>sqrt</i>	Elementwise square root function.
<i>square</i>	Elementwise square function.
<i>cbrt</i>	Elementwise cube root function.
<i>reciprocal</i>	Computes $1 / x$ elementwise.
<i>gcd</i>	Computes gcd of $x1$ and $x2$ elementwise.
<i>lcm</i>	Computes lcm of $x1$ and $x2$ elementwise.

**cupy.add**

`cupy.add = <ufunc 'cupy_add'>`

Adds two arrays elementwise.

**See also:**

`numpy.add`

**cupy.subtract**

`cupy.subtract = <ufunc 'cupy_subtract'>`

Subtracts arguments elementwise.

**See also:**

`numpy.subtract`

## cupy.multiply

`cupy.multiply = <ufunc 'cupy_multiply'>`

Multiplies two arrays elementwise.

**See also:**

`numpy.multiply`

## cupy.matmul

`cupy.matmul(x1, x2, out=None, *, axes=None)`

Matrix product of two arrays.

Returns the matrix product of two arrays and is the implementation of the `@` operator introduced in Python 3.5 following PEP465.

The main difference against `cupy.dot` are the handling of arrays with more than 2 dimensions. For more information see `numpy.matmul()`.

---

**Note:** The out array as input is currently not supported.

---

### Parameters

- **x1** (`cupy.ndarray`) – The left argument.
- **x2** (`cupy.ndarray`) – The right argument.
- **out** (`cupy.ndarray`, *optional*) – Output array.
- **axes** (*List of tuples of int, optional*) – A list of tuples with indices of axes the matrix multiplication should operate on.

**Returns** Output array.

**Return type** `cupy.ndarray`

**See also:**

`numpy.matmul()`

## cupy.divide

`cupy.divide = <ufunc 'cupy_true_divide'>`

Elementwise true division (i.e. division as floating values).

**See also:**

`numpy.true_divide`



### **cupy.logaddexp**

`cupy.logaddexp = <ufunc 'cupy_logaddexp'>`  
Computes  $\log(\exp(x_1) + \exp(x_2))$  elementwise.

**See also:**

`numpy.logaddexp`

### **cupy.logaddexp2**

`cupy.logaddexp2 = <ufunc 'cupy_logaddexp2'>`  
Computes  $\log_2(\exp_2(x_1) + \exp_2(x_2))$  elementwise.

**See also:**

`numpy.logaddexp2`

### **cupy.true\_divide**

`cupy.true_divide = <ufunc 'cupy_true_divide'>`  
Elementwise true division (i.e. division as floating values).

**See also:**

`numpy.true_divide`

### **cupy.floor\_divide**

`cupy.floor_divide = <ufunc 'cupy_floor_divide'>`  
Elementwise floor division (i.e. integer quotient).

**See also:**

`numpy.floor_divide`

### **cupy.negative**

`cupy.negative = <ufunc 'cupy_negative'>`  
Takes numerical negative elementwise.

**See also:**

`numpy.negative`

### **cupy.power**

`cupy.power = <ufunc 'cupy_power'>`  
Computes  $x1 ** x2$  elementwise.

**See also:**

`numpy.power`

### **cupy.remainder**

`cupy.remainder = <ufunc 'cupy_remainder'>`  
Computes the remainder of Python division elementwise.

**See also:**

`numpy.remainder`

### **cupy.mod**

`cupy.mod = <ufunc 'cupy_remainder'>`  
Computes the remainder of Python division elementwise.

**See also:**

`numpy.remainder`

### **cupy.fmod**

`cupy.fmod = <ufunc 'cupy_fmod'>`  
Computes the remainder of C division elementwise.

**See also:**

`numpy.fmod`

### **cupy.absolute**

`cupy.absolute = <ufunc 'cupy_absolute'>`  
Elementwise absolute value function.

**See also:**

`numpy.absolute`

### **cupy rint**

`cupy.rint = <ufunc 'cupy_rint'>`

Rounds each element of an array to the nearest integer.

**See also:**

`numpy.rint`

### **cupy.sign**

`cupy.sign = <ufunc 'cupy_sign'>`

Elementwise sign function.

It returns -1, 0, or 1 depending on the sign of the input.

**See also:**

`numpy.sign`

### **cupy.conj**

`cupy.conj = <ufunc 'cupy_conjugate'>`

Returns the complex conjugate, element-wise.

**See also:**

`numpy.conjugate`

### **cupy.conjugate**

`cupy.conjugate = <ufunc 'cupy_conjugate'>`

Returns the complex conjugate, element-wise.

**See also:**

`numpy.conjugate`

### **cupy.exp**

`cupy.exp = <ufunc 'cupy_exp'>`

Elementwise exponential function.

**See also:**

`numpy.exp`

### **cupy.exp2**

`cupy.exp2 = <ufunc 'cupy_exp2'>`  
Elementwise exponentiation with base 2.

**See also:**

`numpy.exp2`

### **cupy.log**

`cupy.log = <ufunc 'cupy_log'>`  
Elementwise natural logarithm function.

**See also:**

`numpy.log`

### **cupy.log2**

`cupy.log2 = <ufunc 'cupy_log2'>`  
Elementwise binary logarithm function.

**See also:**

`numpy.log2`

### **cupy.log10**

`cupy.log10 = <ufunc 'cupy_log10'>`  
Elementwise common logarithm function.

**See also:**

`numpy.log10`

### **cupy.expm1**

`cupy.expm1 = <ufunc 'cupy_expm1'>`  
Computes  $\exp(x) - 1$  elementwise.

**See also:**

`numpy.expm1`

### **cupy.log1p**

`cupy.log1p = <ufunc 'cupy_log1p'>`  
Computes  $\log(1 + x)$  elementwise.

**See also:**

`numpy.log1p`

### **cupy.sqrt**

`cupy.sqrt = <ufunc 'cupy_sqrt'>`  
Elementwise square root function.

**See also:**

`numpy.sqrt`

### **cupy.square**

`cupy.square = <ufunc 'cupy_square'>`  
Elementwise square function.

**See also:**

`numpy.square`

### **cupy.cbrt**

`cupy.cbrt = <ufunc 'cupy_cbrt'>`  
Elementwise cube root function.

**See also:**

`numpy.cbrt`

### **cupy.reciprocal**

`cupy.reciprocal = <ufunc 'cupy_reciprocal'>`  
Computes  $1 / x$  elementwise.

**See also:**

`numpy.reciprocal`

## cupy.gcd

`cupy.gcd = <ufunc 'cupy_gcd'>`  
Computes gcd of x1 and x2 elementwise.

**See also:**

`numpy.gcd`

## cupy.lcm

`cupy.lcm = <ufunc 'cupy_lcm'>`  
Computes lcm of x1 and x2 elementwise.

**See also:**

`numpy.lcm`

## Trigonometric functions

<i>sin</i>	Elementwise sine function.
<i>cos</i>	Elementwise cosine function.
<i>tan</i>	Elementwise tangent function.
<i>arcsin</i>	Elementwise inverse-sine function (a.k.a.
<i>arccos</i>	Elementwise inverse-cosine function (a.k.a.
<i>arctan</i>	Elementwise inverse-tangent function (a.k.a.
<i>arctan2</i>	Elementwise inverse-tangent of the ratio of two arrays.
<i>hypot</i>	Computes the hypoteneous of orthogonal vectors of given length.
<i>sinh</i>	Elementwise hyperbolic sine function.
<i>cosh</i>	Elementwise hyperbolic cosine function.
<i>tanh</i>	Elementwise hyperbolic tangent function.
<i>arcsinh</i>	Elementwise inverse of hyperbolic sine function.
<i>arccosh</i>	Elementwise inverse of hyperbolic cosine function.
<i>arctanh</i>	Elementwise inverse of hyperbolic tangent function.
<i>degrees</i>	Converts angles from radians to degrees elementwise.
<i>radians</i>	Converts angles from degrees to radians elementwise.
<i>deg2rad</i>	Converts angles from degrees to radians elementwise.
<i>rad2deg</i>	Converts angles from radians to degrees elementwise.

## cupy.sin

`cupy.sin = <ufunc 'cupy_sin'>`  
Elementwise sine function.

**See also:**

`numpy.sin`

### **cupy.cos**

`cupy.cos = <ufunc 'cupy_cos'>`  
Elementwise cosine function.

**See also:**

`numpy.cos`

### **cupy.tan**

`cupy.tan = <ufunc 'cupy_tan'>`  
Elementwise tangent function.

**See also:**

`numpy.tan`

### **cupy.arcsin**

`cupy.arcsin = <ufunc 'cupy_arcsin'>`  
Elementwise inverse-sine function (a.k.a. arcsine function).

**See also:**

`numpy.arcsin`

### **cupy.arccos**

`cupy.arccos = <ufunc 'cupy_arccos'>`  
Elementwise inverse-cosine function (a.k.a. arccosine function).

**See also:**

`numpy.arccos`

### **cupy.arctan**

`cupy.arctan = <ufunc 'cupy_arctan'>`  
Elementwise inverse-tangent function (a.k.a. arctangent function).

**See also:**

`numpy.arctan`

## cupy.arctan2

`cupy.arctan2 = <ufunc 'cupy_arctan2'>`  
Elementwise inverse-tangent of the ratio of two arrays.

**See also:**

`numpy.arctan2`

## cupy.hypot

`cupy.hypot = <ufunc 'cupy_hypot'>`  
Computes the hypoteneous of orthogonal vectors of given length.  
This is equivalent to  $\sqrt{x_1^2 + x_2^2}$ , while this function is more efficient.

**See also:**

`numpy.hypot`

## cupy.sinh

`cupy.sinh = <ufunc 'cupy_sinh'>`  
Elementwise hyperbolic sine function.

**See also:**

`numpy.sinh`

## cupy.cosh

`cupy.cosh = <ufunc 'cupy_cosh'>`  
Elementwise hyperbolic cosine function.

**See also:**

`numpy.cosh`

## cupy.tanh

`cupy.tanh = <ufunc 'cupy_tanh'>`  
Elementwise hyperbolic tangent function.

**See also:**

`numpy.tanh`



### **cupy.arcsinh**

`cupy.arcsinh` = <ufunc 'cupy\_arcsinh'>  
Elementwise inverse of hyperbolic sine function.

**See also:**

`numpy.arcsinh`

### **cupy.arccosh**

`cupy.arccosh` = <ufunc 'cupy\_arccosh'>  
Elementwise inverse of hyperbolic cosine function.

**See also:**

`numpy.arccosh`

### **cupy.arctanh**

`cupy.arctanh` = <ufunc 'cupy\_arctanh'>  
Elementwise inverse of hyperbolic tangent function.

**See also:**

`numpy.arctanh`

### **cupy.degrees**

`cupy.degrees` = <ufunc 'cupy\_rad2deg'>  
Converts angles from radians to degrees elementwise.

**See also:**

`numpy.rad2deg`, `numpy.degrees`

### **cupy.radians**

`cupy.radians` = <ufunc 'cupy\_deg2rad'>  
Converts angles from degrees to radians elementwise.

**See also:**

`numpy.deg2rad`, `numpy.radians`

## cupy.deg2rad

`cupy.deg2rad = <ufunc 'cupy_deg2rad'>`  
Converts angles from degrees to radians elementwise.

**See also:**

`numpy.deg2rad`, `numpy.radians`

## cupy.rad2deg

`cupy.rad2deg = <ufunc 'cupy_rad2deg'>`  
Converts angles from radians to degrees elementwise.

**See also:**

`numpy.rad2deg`, `numpy.degrees`

## Bit-twiddling functions

<i><code>bitwise_and</code></i>	Computes the bitwise AND of two arrays elementwise.
<i><code>bitwise_or</code></i>	Computes the bitwise OR of two arrays elementwise.
<i><code>bitwise_xor</code></i>	Computes the bitwise XOR of two arrays elementwise.
<i><code>invert</code></i>	Computes the bitwise NOT of an array elementwise.
<i><code>left_shift</code></i>	Shifts the bits of each integer element to the left.
<i><code>right_shift</code></i>	Shifts the bits of each integer element to the right.

## cupy.bitwise\_and

`cupy.bitwise_and = <ufunc 'cupy_bitwise_and'>`  
Computes the bitwise AND of two arrays elementwise.

Only integer and boolean arrays are handled.

**See also:**

`numpy.bitwise_and`

## cupy.bitwise\_or

`cupy.bitwise_or = <ufunc 'cupy_bitwise_or'>`  
Computes the bitwise OR of two arrays elementwise.

Only integer and boolean arrays are handled.

**See also:**

`numpy.bitwise_or`

### `cupy.bitwise_xor`

`cupy.bitwise_xor = <ufunc 'cupy_bitwise_xor'>`  
Computes the bitwise XOR of two arrays elementwise.  
Only integer and boolean arrays are handled.  
**See also:**  
`numpy.bitwise_xor`

### `cupy.invert`

`cupy.invert = <ufunc 'cupy_invert'>`  
Computes the bitwise NOT of an array elementwise.  
Only integer and boolean arrays are handled.

---

**Note:** `cupy.bitwise_not()` is an alias for `cupy.invert()`.

---

**See also:**  
`numpy.invert`

### `cupy.left_shift`

`cupy.left_shift = <ufunc 'cupy_left_shift'>`  
Shifts the bits of each integer element to the left.  
Only integer arrays are handled.  
**See also:**  
`numpy.left_shift`

### `cupy.right_shift`

`cupy.right_shift = <ufunc 'cupy_right_shift'>`  
Shifts the bits of each integer element to the right.  
Only integer arrays are handled  
**See also:**  
`numpy.right_shift`

## Comparison functions

<i>greater</i>	Tests elementwise if $x1 > x2$ .
<i>greater_equal</i>	Tests elementwise if $x1 \geq x2$ .
<i>less</i>	Tests elementwise if $x1 < x2$ .
<i>less_equal</i>	Tests elementwise if $x1 \leq x2$ .
<i>not_equal</i>	Tests elementwise if $x1 \neq x2$ .
<i>equal</i>	Tests elementwise if $x1 == x2$ .
<i>logical_and</i>	Computes the logical AND of two arrays.
<i>logical_or</i>	Computes the logical OR of two arrays.
<i>logical_xor</i>	Computes the logical XOR of two arrays.
<i>logical_not</i>	Computes the logical NOT of an array.
<i>maximum</i>	Takes the maximum of two arrays elementwise.
<i>minimum</i>	Takes the minimum of two arrays elementwise.
<i>fmax</i>	Takes the maximum of two arrays elementwise.
<i>fmin</i>	Takes the minimum of two arrays elementwise.

### cupy.greater

`cupy.greater = <ufunc 'cupy_greater'>`

Tests elementwise if  $x1 > x2$ .

**See also:**

`numpy.greater`

### cupy.greater\_equal

`cupy.greater_equal = <ufunc 'cupy_greater_equal'>`

Tests elementwise if  $x1 \geq x2$ .

**See also:**

`numpy.greater_equal`

### cupy.less

`cupy.less = <ufunc 'cupy_less'>`

Tests elementwise if  $x1 < x2$ .

**See also:**

`numpy.less`

### **cupy.less\_equal**

`cupy.less_equal = <ufunc 'cupy_less_equal'>`  
Tests elementwise if  $x1 \leq x2$ .

**See also:**

`numpy.less_equal`

### **cupy.not\_equal**

`cupy.not_equal = <ufunc 'cupy_not_equal'>`  
Tests elementwise if  $x1 \neq x2$ .

**See also:**

`numpy.equal`

### **cupy.equal**

`cupy.equal = <ufunc 'cupy_equal'>`  
Tests elementwise if  $x1 == x2$ .

**See also:**

`numpy.equal`

### **cupy.logical\_and**

`cupy.logical_and = <ufunc 'cupy_logical_and'>`  
Computes the logical AND of two arrays.

**See also:**

`numpy.logical_and`

### **cupy.logical\_or**

`cupy.logical_or = <ufunc 'cupy_logical_or'>`  
Computes the logical OR of two arrays.

**See also:**

`numpy.logical_or`

### **cupy.logical\_xor**

`cupy.logical_xor = <ufunc 'cupy_logical_xor'>`

Computes the logical XOR of two arrays.

**See also:**

`numpy.logical_xor`

### **cupy.logical\_not**

`cupy.logical_not = <ufunc 'cupy_logical_not'>`

Computes the logical NOT of an array.

**See also:**

`numpy.logical_not`

### **cupy.maximum**

`cupy.maximum = <ufunc 'cupy_maximum'>`

Takes the maximum of two arrays elementwise.

If NaN appears, it returns the NaN.

**See also:**

`numpy.maximum`

### **cupy.minimum**

`cupy.minimum = <ufunc 'cupy_minimum'>`

Takes the minimum of two arrays elementwise.

If NaN appears, it returns the NaN.

**See also:**

`numpy.minimum`

### **cupy.fmax**

`cupy.fmax = <ufunc 'cupy_fmax'>`

Takes the maximum of two arrays elementwise.

If NaN appears, it returns the other operand.

**See also:**

`numpy.fmax`

**cupy.fmin**

`cupy.fmin = <ufunc 'cupy_fmin'>`

Takes the minimum of two arrays elementwise.

If NaN appears, it returns the other operand.

**See also:**

`numpy.fmin`

**Floating functions**

<i>isfinite</i>	Tests finiteness elementwise.
<i>isinf</i>	Tests if each element is the positive or negative infinity.
<i>isnan</i>	Tests if each element is a NaN.
<i>signbit</i>	Tests elementwise if the sign bit is set (i.e.
<i>copysign</i>	Returns the first argument with the sign bit of the second elementwise.
<i>nextafter</i>	Computes the nearest neighbor float values towards the second argument.
<i>modf</i>	Extracts the fractional and integral parts of an array elementwise.
<i>ldexp</i>	Computes $x1 * 2^{x2}$ elementwise.
<i>frexp</i>	Decomposes each element to mantissa and two's exponent.
<i>fmod</i>	Computes the remainder of C division elementwise.
<i>floor</i>	Rounds each element of an array to its floor integer.
<i>ceil</i>	Rounds each element of an array to its ceiling integer.
<i>trunc</i>	Rounds each element of an array towards zero.

**cupy.isfinite**

`cupy.isfinite = <ufunc 'cupy_isfinite'>`

Tests finiteness elementwise.

Each element of returned array is True only if the corresponding element of the input is finite (i.e. not an infinity nor NaN).

**See also:**

`numpy.isfinite`

### cupy.isinf

`cupy.isinf = <ufunc 'cupy_isinf'>`  
Tests if each element is the positive or negative infinity.

**See also:**

`numpy.isinf`

### cupy.isnan

`cupy.isnan = <ufunc 'cupy_isnan'>`  
Tests if each element is a NaN.

**See also:**

`numpy.isnan`

### cupy.signbit

`cupy.signbit = <ufunc 'cupy_signbit'>`  
Tests elementwise if the sign bit is set (i.e. less than zero).

**See also:**

`numpy.signbit`

### cupy.copysign

`cupy.copysign = <ufunc 'cupy_copysign'>`  
Returns the first argument with the sign bit of the second elementwise.

**See also:**

`numpy.copysign`

### cupy.nextafter

`cupy.nextafter = <ufunc 'cupy_nextafter'>`  
Computes the nearest neighbor float values towards the second argument.

---

**Note:** For values that are close to zero (or denormal numbers), results of `cupy.nextafter()` may be different from those of `numpy.nextafter()`, because CuPy sets `-ftz=true`.

---

**See also:**

`numpy.nextafter`



### **cupy.modf**

`cupy.modf = <ufunc 'cupy_modf'>`

Extracts the fractional and integral parts of an array elementwise.

This ufunc returns two arrays.

**See also:**

`numpy.modf`

### **cupy.ldexp**

`cupy.ldexp = <ufunc 'cupy_ldexp'>`

Computes  $x1 * 2^{** x2}$  elementwise.

**See also:**

`numpy.ldexp`

### **cupy.frexp**

`cupy.frexp = <ufunc 'cupy_frexp'>`

Decomposes each element to mantissa and two's exponent.

This ufunc outputs two arrays of the input dtype and the int dtype.

**See also:**

`numpy.frexp`

### **cupy.floor**

`cupy.floor = <ufunc 'cupy_floor'>`

Rounds each element of an array to its floor integer.

**See also:**

`numpy.floor`

### **cupy.ceil**

`cupy.ceil = <ufunc 'cupy_ceil'>`

Rounds each element of an array to its ceiling integer.

**See also:**

`numpy.ceil`

## cupy.trunc

`cupy.trunc = <ufunc 'cupy_trunc'>`  
Rounds each element of an array towards zero.

See also:

`numpy.trunc`

### 5.2.3 ufunc.at

Currently, CuPy does not support `at` for ufuncs in general. However, `cupyx.scatter_add()` can substitute `add.at` as both behave identically.

## 5.3 Routines (NumPy)

The following pages describe NumPy-compatible routines. These functions cover a subset of [NumPy routines](#).

### 5.3.1 Array creation routines

---

**Hint:** [NumPy API Reference: Array creation routines](#)

---

#### Ones and zeros

<code>empty(shape[, dtype, order])</code>	Returns an array without initializing the elements.
<code>empty_like(a[, dtype, order, subok, shape])</code>	Returns a new array with same shape and dtype of a given array.
<code>eye(N[, M, k, dtype, order])</code>	Returns a 2-D array with ones on the diagonals and zeros elsewhere.
<code>identity(n[, dtype])</code>	Returns a 2-D identity array.
<code>ones(shape[, dtype, order])</code>	Returns a new array of given shape and dtype, filled with ones.
<code>ones_like(a[, dtype, order, subok, shape])</code>	Returns an array of ones with same shape and dtype as a given array.
<code>zeros(shape[, dtype, order])</code>	Returns a new array of given shape and dtype, filled with zeros.
<code>zeros_like(a[, dtype, order, subok, shape])</code>	Returns an array of zeros with same shape and dtype as a given array.
<code>full(shape, fill_value[, dtype, order])</code>	Returns a new array of given shape and dtype, filled with a given value.
<code>full_like(a, fill_value[, dtype, order, ...])</code>	Returns a full array with same shape and dtype as a given array.

## cupy.empty

`cupy.empty(shape, dtype=<class 'float'>, order='C')`  
 Returns an array without initializing the elements.

### Parameters

- **shape** (*int or tuple of ints*) – Dimensionalities of the array.
- **dtype** – Data type specifier.
- **order** (`{'C', 'F'}`) – Row-major (C-style) or column-major (Fortran-style) order.

**Returns** A new array with elements not initialized.

**Return type** `cupy.ndarray`

See also:

`numpy.empty()`

## cupy.empty\_like

`cupy.empty_like(a, dtype=None, order='K', subok=None, shape=None)`  
 Returns a new array with same shape and dtype of a given array.

This function currently does not support `subok` option.

### Parameters

- **a** (`cupy.ndarray`) – Base array.
- **dtype** – Data type specifier. The data type of `a` is used by default.
- **order** (`{'C', 'F', 'A', or 'K'}`) – Overrides the memory layout of the result. 'C' means C-order, 'F' means F-order, 'A' means 'F' if `a` is Fortran contiguous, 'C' otherwise. 'K' means match the layout of `a` as closely as possible.
- **subok** – Not supported yet, must be `None`.
- **shape** (*int or tuple of ints*) – Overrides the shape of the result. If `order='K'` and the number of dimensions is unchanged, will try to keep order, otherwise, `order='C'` is implied.

**Returns** A new array with same shape and dtype of `a` with elements not initialized.

**Return type** `cupy.ndarray`

See also:

`numpy.empty_like()`

## cupy.eye

`cupy.eye(N, M=None, k=0, dtype=<class 'float'>, order='C')`

Returns a 2-D array with ones on the diagonals and zeros elsewhere.

### Parameters

- **N** (*int*) – Number of rows.
- **M** (*int*) – Number of columns. **M** == **N** by default.
- **k** (*int*) – Index of the diagonal. Zero indicates the main diagonal, a positive index an upper diagonal, and a negative index a lower diagonal.
- **dtype** – Data type specifier.
- **order** (*{'C', 'F'}*) – Row-major (C-style) or column-major (Fortran-style) order.

**Returns** A 2-D array with given diagonals filled with ones and zeros elsewhere.

**Return type** *cupy.ndarray*

**See also:**

`numpy.eye()`

## cupy.identity

`cupy.identity(n, dtype=<class 'float'>)`

Returns a 2-D identity array.

It is equivalent to `eye(n, n, dtype)`.

### Parameters

- **n** (*int*) – Number of rows and columns.
- **dtype** – Data type specifier.

**Returns** A 2-D identity array.

**Return type** *cupy.ndarray*

**See also:**

`numpy.identity()`

## cupy.ones

`cupy.ones(shape, dtype=<class 'float'>, order='C')`

Returns a new array of given shape and dtype, filled with ones.

This function currently does not support **order** option.

### Parameters

- **shape** (*int or tuple of ints*) – Dimensionalities of the array.
- **dtype** – Data type specifier.
- **order** (*{'C', 'F'}*) – Row-major (C-style) or column-major (Fortran-style) order.

**Returns** An array filled with ones.

**Return type** *cupy.ndarray*

See also:

`numpy.ones()`

### `cupy.ones_like`

`cupy.ones_like(a, dtype=None, order='K', subok=None, shape=None)`

Returns an array of ones with same shape and dtype as a given array.

This function currently does not support `subok` option.

#### Parameters

- **a** (*cupy.ndarray*) – Base array.
- **dtype** – Data type specifier. The dtype of `a` is used by default.
- **order** (`{'C', 'F', 'A', or 'K'}`) – Overrides the memory layout of the result. 'C' means C-order, 'F' means F-order, 'A' means 'F' if `a` is Fortran contiguous, 'C' otherwise. 'K' means match the layout of `a` as closely as possible.
- **subok** – Not supported yet, must be `None`.
- **shape** (*int or tuple of ints*) – Overrides the shape of the result. If `order='K'` and the number of dimensions is unchanged, will try to keep order, otherwise, `order='C'` is implied.

**Returns** An array filled with ones.

**Return type** *cupy.ndarray*

See also:

`numpy.ones_like()`

### `cupy.zeros`

`cupy.zeros(shape, dtype=<class 'float'>, order='C')`

Returns a new array of given shape and dtype, filled with zeros.

#### Parameters

- **shape** (*int or tuple of ints*) – Dimensionalities of the array.
- **dtype** – Data type specifier.
- **order** (`{'C', 'F'}`) – Row-major (C-style) or column-major (Fortran-style) order.

**Returns** An array filled with zeros.

**Return type** *cupy.ndarray*

See also:

`numpy.zeros()`

## cupy.zeros\_like

`cupy.zeros_like(a, dtype=None, order='K', subok=None, shape=None)`

Returns an array of zeros with same shape and dtype as a given array.

This function currently does not support `subok` option.

### Parameters

- **a** (`cupy.ndarray`) – Base array.
- **dtype** – Data type specifier. The dtype of `a` is used by default.
- **order** (`{'C', 'F', 'A', or 'K'}`) – Overrides the memory layout of the result. 'C' means C-order, 'F' means F-order, 'A' means 'F' if `a` is Fortran contiguous, 'C' otherwise. 'K' means match the layout of `a` as closely as possible.
- **subok** – Not supported yet, must be `None`.
- **shape** (`int or tuple of ints`) – Overrides the shape of the result. If `order='K'` and the number of dimensions is unchanged, will try to keep order, otherwise, `order='C'` is implied.

**Returns** An array filled with zeros.

**Return type** `cupy.ndarray`

See also:

`numpy.zeros_like()`

## cupy.full

`cupy.full(shape, fill_value, dtype=None, order='C')`

Returns a new array of given shape and dtype, filled with a given value.

This function currently does not support `order` option.

### Parameters

- **shape** (`int or tuple of ints`) – Dimensionalities of the array.
- **fill\_value** – A scalar value to fill a new array.
- **dtype** – Data type specifier.
- **order** (`{'C', 'F'}`) – Row-major (C-style) or column-major (Fortran-style) order.

**Returns** An array filled with `fill_value`.

**Return type** `cupy.ndarray`

See also:

`numpy.full()`

## cupy.full\_like

`cupy.full_like(a, fill_value, dtype=None, order='K', subok=None, shape=None)`

Returns a full array with same shape and dtype as a given array.

This function currently does not support `subok` option.

### Parameters

- **a** (`cupy.ndarray`) – Base array.
- **fill\_value** – A scalar value to fill a new array.
- **dtype** – Data type specifier. The dtype of `a` is used by default.
- **order** (`{'C', 'F', 'A', or 'K'}`) – Overrides the memory layout of the result. 'C' means C-order, 'F' means F-order, 'A' means 'F' if `a` is Fortran contiguous, 'C' otherwise. 'K' means match the layout of `a` as closely as possible.
- **subok** – Not supported yet, must be `None`.
- **shape** (`int or tuple of ints`) – Overrides the shape of the result. If `order='K'` and the number of dimensions is unchanged, will try to keep order, otherwise, `order='C'` is implied.

**Returns** An array filled with `fill_value`.

**Return type** `cupy.ndarray`

**See also:**

`numpy.full_like()`

## From existing data

<code>array(obj[, dtype, copy, order, subok, ndmin])</code>	Creates an array on the current device.
<code>asarray(a[, dtype, order])</code>	Converts an object to array.
<code>asanyarray(a[, dtype, order])</code>	Converts an object to array.
<code>ascontiguousarray(a[, dtype])</code>	Returns a C-contiguous array.
<code>copy(a[, order])</code>	Creates a copy of a given array on the current device.
<code>fromfile(*args, **kwargs)</code>	Reads an array from a file.

## cupy.asanyarray

`cupy.asanyarray(a, dtype=None, order=None)`

Converts an object to array.

This is currently equivalent to `cupy.asarray()`, since there is no subclass of `cupy.ndarray` in CuPy. Note that the original `numpy.asanyarray()` returns the input array as is if it is an instance of a subtype of `numpy.ndarray`.

**See also:**

`cupy.asarray()`, `numpy.asanyarray()`

## cupy.ascontiguousarray

`cupy.ascontiguousarray(a, dtype=None)`

Returns a C-contiguous array.

### Parameters

- **a** (`cupy.ndarray`) – Source array.
- **dtype** – Data type specifier.

**Returns** If no copy is required, it returns a. Otherwise, it returns a copy of a.

**Return type** `cupy.ndarray`

See also:

`numpy.ascontiguousarray()`

## cupy.copy

`cupy.copy(a, order='K')`

Creates a copy of a given array on the current device.

This function allocates the new array on the current device. If the given array is allocated on the different device, then this function tries to copy the contents over the devices.

### Parameters

- **a** (`cupy.ndarray`) – The source array.
- **order** (`{'C', 'F', 'A', 'K'}`) – Row-major (C-style) or column-major (Fortran-style) order. When order is 'A', it uses 'F' if a is column-major and uses 'C' otherwise. And when order is 'K', it keeps strides as closely as possible.

**Returns** The copy of a on the current device.

**Return type** `cupy.ndarray`

See also:

`numpy.copy()`, `cupy.ndarray.copy()`

## cupy.fromfile

`cupy.fromfile(*args, **kwargs)`

Reads an array from a file.

---

**Note:** Uses NumPy's `fromfile` and coerces the result to a CuPy array.

---

See also:

`numpy.fromfile()`



## Numerical ranges

<code>arange(start[, stop, step, dtype])</code>	Returns an array with evenly spaced values within a given interval.
<code>linspace(start, stop[, num, endpoint, ...])</code>	Returns an array with evenly-spaced values within a given interval.
<code>logspace(start, stop[, num, endpoint, base, ...])</code>	Returns an array with evenly-spaced values on a log-scale.
<code>meshgrid(*xi, **kwargs)</code>	Return coordinate matrices from coordinate vectors.
<code>mgrid</code>	Construct a multi-dimensional “meshgrid”.
<code>ogrid</code>	Construct a multi-dimensional “meshgrid”.

### cupy.arange

`cupy.arange(start, stop=None, step=1, dtype=None)`

Returns an array with evenly spaced values within a given interval.

Values are generated within the half-open interval [start, stop). The first three arguments are mapped like the range built-in function, i.e. start and step are optional.

#### Parameters

- **start** – Start of the interval.
- **stop** – End of the interval.
- **step** – Step width between each pair of consecutive values.
- **dtype** – Data type specifier. It is inferred from other arguments by default.

**Returns** The 1-D array of range values.

**Return type** `cupy.ndarray`

See also:

`numpy.arange()`

### cupy.linspace

`cupy.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None, axis=0)`

Returns an array with evenly-spaced values within a given interval.

Instead of specifying the step width like `cupy.arange()`, this function requires the total number of elements specified.

#### Parameters

- **start** (*scalar or array\_like*) – Starting value(s) of the sequence.
- **stop** (*scalar or array\_like*) – Ending value(s) of the sequence, unless **endpoint** is set to `False`. In that case, the sequence consists of all but the last of `num + 1` evenly spaced samples, so that **stop** is excluded. Note that the step size changes when **endpoint** is `False`.
- **num** – Number of elements.
- **endpoint** (*bool*) – If `True`, the stop value is included as the last element. Otherwise, the stop value is omitted.

- **retstep** (*bool*) – If `True`, this function returns (array, step). Otherwise, it returns only the array.
- **dtype** – Data type specifier. It is inferred from the start and stop arguments by default.
- **axis** (*int*) – The axis in the result to store the samples. Relevant only if start or stop are array-like. By default `0`, the samples will be along a new axis inserted at the beginning. Use `-1` to get an axis at the end.

**Returns** The 1-D array of ranged values.

**Return type** *cupy.ndarray*

See also:

*numpy.linspace()*

## cupy.logspace

`cupy.logspace(start, stop, num=50, endpoint=True, base=10.0, dtype=None)`

Returns an array with evenly-spaced values on a log-scale.

Instead of specifying the step width like *cupy.arange()*, this function requires the total number of elements specified.

### Parameters

- **start** – Start of the interval.
- **stop** – End of the interval.
- **num** – Number of elements.
- **endpoint** (*bool*) – If `True`, the stop value is included as the last element. Otherwise, the stop value is omitted.
- **base** (*float*) – Base of the log space. The step sizes between the elements on a log-scale are the same as base.
- **dtype** – Data type specifier. It is inferred from the start and stop arguments by default.

**Returns** The 1-D array of ranged values.

**Return type** *cupy.ndarray*

See also:

*numpy.logspace()*

## cupy.meshgrid

`cupy.meshgrid(*xi, **kwargs)`

Return coordinate matrices from coordinate vectors.

Given one-dimensional coordinate arrays `x1`, `x2`, ..., `xn` this function makes N-D grids.

For one-dimensional arrays `x1`, `x2`, ..., `xn` with lengths `Ni = len(xi)`, this function returns (`N1`, `N2`, `N3`, ..., `Nn`) shaped arrays if `indexing='ij'` or (`N2`, `N1`, `N3`, ..., `Nn`) shaped arrays if `indexing='xy'`.

Unlike NumPy, CuPy currently only supports 1-D arrays as inputs.

### Parameters

- **xi** (*tuple of ndarrays*) – 1-D arrays representing the coordinates of a grid.
- **indexing** (*{'xy', 'ij'}, optional*) – Cartesian ('xy', default) or matrix ('ij') indexing of output.
- **sparse** (*bool, optional*) – If True, a sparse grid is returned in order to conserve memory. Default is False.
- **copy** (*bool, optional*) – If False, a view into the original arrays are returned. Default is True.

**Returns** list of cupy.ndarray

**See also:**

[numpy.meshgrid\(\)](#)

## cupy.mgrid

**cupy.mgrid** = <cupy.\_creation.ranges.nd\_grid object>

Construct a multi-dimensional “meshgrid”.

`grid = nd_grid()` creates an instance which will return a mesh-grid when indexed. The dimension and number of the output arrays are equal to the number of indexing dimensions. If the step length is not a complex number, then the stop is not inclusive.

However, if the step length is a **complex number** (e.g. 5j), then the integer part of its magnitude is interpreted as specifying the number of points to create between the start and stop values, where the stop value **is inclusive**.

If instantiated with an argument of `sparse=True`, the mesh-grid is open (or not fleshed out) so that only one-dimension of each returned argument is greater than 1.

**Parameters** `sparse` (*bool, optional*) – Whether the grid is sparse or not. Default is False.

**See also:**

[numpy.mgrid](#) and [numpy.ogrid](#)

## cupy.ogrid

**cupy.ogrid** = <cupy.\_creation.ranges.nd\_grid object>

Construct a multi-dimensional “meshgrid”.

`grid = nd_grid()` creates an instance which will return a mesh-grid when indexed. The dimension and number of the output arrays are equal to the number of indexing dimensions. If the step length is not a complex number, then the stop is not inclusive.

However, if the step length is a **complex number** (e.g. 5j), then the integer part of its magnitude is interpreted as specifying the number of points to create between the start and stop values, where the stop value **is inclusive**.

If instantiated with an argument of `sparse=True`, the mesh-grid is open (or not fleshed out) so that only one-dimension of each returned argument is greater than 1.

**Parameters** `sparse` (*bool, optional*) – Whether the grid is sparse or not. Default is False.

**See also:**

[numpy.mgrid](#) and [numpy.ogrid](#)

## Building matrices

<code>diag(v[, k])</code>	Returns a diagonal or a diagonal array.
<code>diagflat(v[, k])</code>	Creates a diagonal array from the flattened input.
<code>tri(N[, M, k, dtype])</code>	Creates an array with ones at and below the given diagonal.
<code>tril(m[, k])</code>	Returns a lower triangle of an array.
<code>triu(m[, k])</code>	Returns an upper triangle of an array.

### `cupy.diag`

`cupy.diag(v, k=0)`

Returns a diagonal or a diagonal array.

**Parameters**

- **v** (*array-like*) – Array or array-like object.
- **k** (*int*) – Index of diagonals. Zero indicates the main diagonal, a positive value an upper diagonal, and a negative value a lower diagonal.

**Returns** If **v** indicates a 1-D array, then it returns a 2-D array with the specified diagonal filled by **v**. If **v** indicates a 2-D array, then it returns the specified diagonal of **v**. In latter case, if **v** is a `cupy.ndarray` object, then its view is returned.

**Return type** `cupy.ndarray`

See also:

`numpy.diag()`

### `cupy.diagflat`

`cupy.diagflat(v, k=0)`

Creates a diagonal array from the flattened input.

**Parameters**

- **v** (*array-like*) – Array or array-like object.
- **k** (*int*) – Index of diagonals. See `cupy.diag()` for detail.

**Returns** A 2-D diagonal array with the diagonal copied from **v**.

**Return type** `cupy.ndarray`

See also:

`numpy.diagflat()`

## cupy.tri

`cupy.tri(N, M=None, k=0, dtype=<class 'float'>)`

Creates an array with ones at and below the given diagonal.

### Parameters

- **N** (*int*) – Number of rows.
- **M** (*int*) – Number of columns. **M** == **N** by default.
- **k** (*int*) – The sub-diagonal at and below which the array is filled. Zero is the main diagonal, a positive value is above it, and a negative value is below.
- **dtype** – Data type specifier.

**Returns** An array with ones at and below the given diagonal.

**Return type** *cupy.ndarray*

See also:

`numpy.tri()`

## cupy.tril

`cupy.tril(m, k=0)`

Returns a lower triangle of an array.

### Parameters

- **m** (*array-like*) – Array or array-like object.
- **k** (*int*) – The diagonal above which to zero elements. Zero is the main diagonal, a positive value is above it, and a negative value is below.

**Returns** A lower triangle of an array.

**Return type** *cupy.ndarray*

See also:

`numpy.tril()`

## cupy.triu

`cupy.triu(m, k=0)`

Returns an upper triangle of an array.

### Parameters

- **m** (*array-like*) – Array or array-like object.
- **k** (*int*) – The diagonal below which to zero elements. Zero is the main diagonal, a positive value is above it, and a negative value is below.

**Returns** An upper triangle of an array.

**Return type** *cupy.ndarray*

See also:

`numpy.triu()`

## 5.3.2 Array manipulation routines

---

**Hint:** [NumPy API Reference: Array manipulation routines](#)

---

### Basic operations

---

<code>copyto(dst, src[, casting, where])</code>	Copies values from one array to another with broadcasting.
<code>shape(a)</code>	Returns the shape of an array

---

### `cupy.copyto`

`cupy.copyto(dst, src, casting='same_kind', where=None)`  
Copies values from one array to another with broadcasting.

This function can be called for arrays on different devices. In this case, `casting`, `where`, and broadcasting is not supported, and an exception is raised if these are used.

#### Parameters

- **dst** (`cupy.ndarray`) – Target array.
- **src** (`cupy.ndarray`) – Source array.
- **casting** (`str`) – Casting rule. See `numpy.can_cast()` for detail.
- **where** (`cupy.ndarray of bool`) – If specified, this array acts as a mask, and an element is copied only if the corresponding element of `where` is True.

See also:

`numpy.copyto()`

### `cupy.shape`

`cupy.shape(a)`  
Returns the shape of an array

**Parameters** `a` (`array_like`) – Input array

**Returns** The elements of the shape tuple give the lengths of the corresponding array dimensions.

**Return type** tuple of ints

## Changing array shape

<code>reshape(a, newshape[, order])</code>	Returns an array with new shape and same elements.
<code>ravel(a[, order])</code>	Returns a flattened array.

### cupy.reshape

`cupy.reshape(a, newshape, order='C')`

Returns an array with new shape and same elements.

It tries to return a view if possible, otherwise returns a copy.

#### Parameters

- **a** (`cupy.ndarray`) – Array to be reshaped.
- **newshape** (*int or tuple of ints*) – The new shape of the array to return. If it is an integer, then it is treated as a tuple of length one. It should be compatible with `a.size`. One of the elements can be -1, which is automatically replaced with the appropriate value to make the shape compatible with `a.size`.
- **order** (`{'C', 'F', 'A'}`) – Read the elements of `a` using this index order, and place the elements into the reshaped array using this index order. 'C' means to read / write the elements using C-like index order, with the last axis index changing fastest, back to the first axis index changing slowest. 'F' means to read / write the elements using Fortran-like index order, with the first index changing fastest, and the last index changing slowest. Note that the 'C' and 'F' options take no account of the memory layout of the underlying array, and only refer to the order of indexing. 'A' means to read / write the elements in Fortran-like index order if `a` is Fortran contiguous in memory, C-like order otherwise.

**Returns** A reshaped view of `a` if possible, otherwise a copy.

**Return type** `cupy.ndarray`

See also:

`numpy.reshape()`

### cupy.ravel

`cupy.ravel(a, order='C')`

Returns a flattened array.

It tries to return a view if possible, otherwise returns a copy.

This function currently does not support the `order = 'K'` option.

#### Parameters

- **a** (`cupy.ndarray`) – Array to be flattened.
- **order** (`{'C', 'F', 'A'}`) – Read the elements of `a` using this index order, and place the elements into the reshaped array using this index order. 'C' means to read / write the elements using C-like index order, with the last axis index changing fastest, back to the first axis index changing slowest. 'F' means to read / write the elements using Fortran-like index order, with the first index changing fastest, and the last index changing slowest. Note that the 'C' and 'F' options take no account of the memory layout of the underlying array, and only refer to

the order of indexing. ‘A’ means to read / write the elements in Fortran-like index order if a is Fortran contiguous in memory, C-like order otherwise.

**Returns** A flattened view of a if possible, otherwise a copy.

**Return type** `cupy.ndarray`

**See also:**

`numpy.ravel()`

**See also:**

`cupy.ndarray.flat` and `cupy.ndarray.flatten()`

## Transpose-like operations

<code>moveaxis(a, source, destination)</code>	Moves axes of an array to new positions.
<code>rollaxis(a, axis[, start])</code>	Moves the specified axis backwards to the given place.
<code>swapaxes(a, axis1, axis2)</code>	Swaps the two axes.
<code>transpose(a[, axes])</code>	Permutates the dimensions of an array.

### `cupy.moveaxis`

`cupy.moveaxis(a, source, destination)`

Moves axes of an array to new positions.

Other axes remain in their original order.

#### Parameters

- **a** (`cupy.ndarray`) – Array whose axes should be reordered.
- **source** (`int` or *sequence of int*) – Original positions of the axes to move. These must be unique.
- **destination** (`int` or *sequence of int*) – Destination positions for each of the original axes. These must also be unique.

**Returns** Array with moved axes. This array is a view of the input array.

**Return type** `cupy.ndarray`

**See also:**

`numpy.moveaxis()`

### `cupy.rollaxis`

`cupy.rollaxis(a, axis, start=0)`

Moves the specified axis backwards to the given place.

#### Parameters

- **a** (`cupy.ndarray`) – Array to move the axis.
- **axis** (`int`) – The axis to move.
- **start** (`int`) – The place to which the axis is moved.



**Returns** A view of `a` that the axis is moved to start.

**Return type** `cupy.ndarray`

**See also:**

`numpy.rollaxis()`

## `cupy.swapaxes`

`cupy.swapaxes(a, axis1, axis2)`

Swaps the two axes.

**Parameters**

- **a** (`cupy.ndarray`) – Array to swap the axes.
- **axis1** (`int`) – The first axis to swap.
- **axis2** (`int`) – The second axis to swap.

**Returns** A view of `a` that the two axes are swapped.

**Return type** `cupy.ndarray`

**See also:**

`numpy.swapaxes()`

## `cupy.transpose`

`cupy.transpose(a, axes=None)`

Permutes the dimensions of an array.

**Parameters**

- **a** (`cupy.ndarray`) – Array to permute the dimensions.
- **axes** (*tuple of ints*) – Permutation of the dimensions. This function reverses the shape by default.

**Returns** A view of `a` that the dimensions are permuted.

**Return type** `cupy.ndarray`

**See also:**

`numpy.transpose()`

**See also:**

`cupy.ndarray.T`

## Changing number of dimensions

<code>atleast_1d(*arys)</code>	Converts arrays to arrays with dimensions $\geq 1$ .
<code>atleast_2d(*arys)</code>	Converts arrays to arrays with dimensions $\geq 2$ .
<code>atleast_3d(*arys)</code>	Converts arrays to arrays with dimensions $\geq 3$ .
<code>broadcast(*arrays)</code>	Object that performs broadcasting.
<code>broadcast_to(array, shape)</code>	Broadcast an array to a given shape.
<code>broadcast_arrays(*args)</code>	Broadcasts given arrays.
<code>expand_dims(a, axis)</code>	Expands given arrays.
<code>squeeze(a[, axis])</code>	Removes size-one axes from the shape of an array.

### `cupy.atleast_1d`

`cupy.atleast_1d(*arys)`

Converts arrays to arrays with dimensions  $\geq 1$ .

**Parameters** `arys` (*tuple of arrays*) – Arrays to be converted. All arguments must be `cupy.ndarray` objects. Only zero-dimensional array is affected.

**Returns** If there are only one input, then it returns its converted version. Otherwise, it returns a list of converted arrays.

**See also:**

`numpy.atleast_1d()`

### `cupy.atleast_2d`

`cupy.atleast_2d(*arys)`

Converts arrays to arrays with dimensions  $\geq 2$ .

If an input array has dimensions less than two, then this function inserts new axes at the head of dimensions to make it have two dimensions.

**Parameters** `arys` (*tuple of arrays*) – Arrays to be converted. All arguments must be `cupy.ndarray` objects.

**Returns** If there are only one input, then it returns its converted version. Otherwise, it returns a list of converted arrays.

**See also:**

`numpy.atleast_2d()`

### `cupy.atleast_3d`

`cupy.atleast_3d(*arys)`

Converts arrays to arrays with dimensions  $\geq 3$ .

If an input array has dimensions less than three, then this function inserts new axes to make it have three dimensions. The place of the new axes are following:

- If its shape is `()`, then the shape of output is `(1, 1, 1)`.
- If its shape is `(N,)`, then the shape of output is `(1, N, 1)`.

- If its shape is (M, N), then the shape of output is (M, N, 1).
- Otherwise, the output is the input array itself.

**Parameters** **arys** (*tuple of arrays*) – Arrays to be converted. All arguments must be `cupy.ndarray` objects.

**Returns** If there are only one input, then it returns its converted version. Otherwise, it returns a list of converted arrays.

**See also:**

`numpy.atleast_3d()`

## `cupy.broadcast`

**class** `cupy.broadcast(*arrays)`

Object that performs broadcasting.

CuPy actually uses this class to support broadcasting in various operations. Note that this class does not provide an iterator.

**Parameters** **arrays** (*tuple of arrays*) – Arrays to be broadcasted.

**Variables**

- **shape** (*tuple of ints*) – The broadcasted shape.
- **nd** (*int*) – Number of dimensions of the broadcasted shape.
- **size** (*int*) – Total size of the broadcasted shape.
- **values** (*list of arrays*) – The broadcasted arrays.

**See also:**

`numpy.broadcast`

## Methods

`__eq__(value, /)`  
Return self==value.

`__ne__(value, /)`  
Return self!=value.

`__lt__(value, /)`  
Return self<value.

`__le__(value, /)`  
Return self<=value.

`__gt__(value, /)`  
Return self>value.

`__ge__(value, /)`  
Return self>=value.

### Attributes

**nd**

**shape**

**size**

**values**

### `cupy.broadcast_to`

`cupy.broadcast_to(array, shape)`

Broadcast an array to a given shape.

#### Parameters

- **array** (`cupy.ndarray`) – Array to broadcast.
- **shape** (*tuple of int*) – The shape of the desired array.

**Returns** Broadcasted view.

**Return type** `cupy.ndarray`

See also:

`numpy.broadcast_to()`

### `cupy.broadcast_arrays`

`cupy.broadcast_arrays(*args)`

Broadcasts given arrays.

**Parameters** **args** (*tuple of arrays*) – Arrays to broadcast for each other.

**Returns** A list of broadcasted arrays.

**Return type** `list`

See also:

`numpy.broadcast_arrays()`

### `cupy.expand_dims`

`cupy.expand_dims(a, axis)`

Expands given arrays.

#### Parameters

- **a** (`cupy.ndarray`) – Array to be expanded.
- **axis** (`int`) – Position where new axis is to be inserted.

**Returns** The number of dimensions is one greater than that of the input array.

**Return type** `cupy.ndarray`

See also:

`numpy.expand_dims()`

**cupy.squeeze**

`cupy.squeeze(a, axis=None)`

Removes size-one axes from the shape of an array.

**Parameters**

- **a** (`cupy.ndarray`) – Array to be reshaped.
- **axis** (*int or tuple of ints*) – Axes to be removed. This function removes all size-one axes by default. If one of the specified axes is not of size one, an exception is raised.

**Returns** An array without (specified) size-one axes.

**Return type** `cupy.ndarray`

**See also:**

`numpy.squeeze()`

**Changing kind of array**

<code>asarray(a[, dtype, order])</code>	Converts an object to array.
<code>asanyarray(a[, dtype, order])</code>	Converts an object to array.
<code>asfortranarray(a[, dtype])</code>	Return an array laid out in Fortran order in memory.
<code>ascontiguousarray(a[, dtype])</code>	Returns a C-contiguous array.
<code>require(a[, dtype, requirements])</code>	Return an array which satisfies the requirements.

**cupy.asfortranarray**

`cupy.asfortranarray(a, dtype=None)`

Return an array laid out in Fortran order in memory.

**Parameters**

- **a** (`ndarray`) – The input array.
- **dtype** (*str or dtype object, optional*) – By default, the data-type is inferred from the input data.

**Returns** The input *a* in Fortran, or column-major, order.

**Return type** `ndarray`

**See also:**

`numpy.asfortranarray()`

## cupy.require

`cupy.require(a, dtype=None, requirements=None)`

Return an array which satisfies the requirements.

### Parameters

- **a** (`ndarray`) – The input array.
- **dtype** (`str` or `dtype object`, *optional*) – The required data-type. If `None` preserve the current dtype.
- **requirements** (`str` or *list of str*) – The requirements can be any of the following
  - `'F_CONTIGUOUS'` (`'F'`, `'FORTRAN'`) - ensure a Fortran-contiguous array.
  - `'C_CONTIGUOUS'` (`'C'`, `'CONTIGUOUS'`) - ensure a C-contiguous array.
  - `'OWNDATA'` (`'O'`) - ensure an array that owns its own data.

**Returns** The input array `a` with specified requirements and type if provided.

**Return type** `ndarray`

See also:

`numpy.require()`

## Joining arrays

<code>concatenate(tup[, axis, out])</code>	Joins arrays along an axis.
<code>stack(tup[, axis, out])</code>	Stacks arrays along a new axis.
<code>vstack(tup)</code>	Stacks arrays vertically.
<code>hstack(tup)</code>	Stacks arrays horizontally.
<code>dstack(tup)</code>	Stacks arrays along the third axis.
<code>column_stack(tup)</code>	Stacks 1-D and 2-D arrays as columns into a 2-D array.

## cupy.concatenate

`cupy.concatenate(tup, axis=0, out=None)`

Joins arrays along an axis.

### Parameters

- **tup** (*sequence of arrays*) – Arrays to be joined. All of these should have same dimensionalities except the specified axis.
- **axis** (`int` or `None`) – The axis to join arrays along. If `axis` is `None`, arrays are flattened before use. Default is 0.
- **out** (`cupy.ndarray`) – Output array.

**Returns** Joined array.

**Return type** `cupy.ndarray`

See also:

`numpy.concatenate()`

## cupy.stack

`cupy.stack(tup, axis=0, out=None)`  
 Stacks arrays along a new axis.

### Parameters

- **tup** (*sequence of arrays*) – Arrays to be stacked.
- **axis** (*int*) – Axis along which the arrays are stacked.
- **out** (`cupy.ndarray`) – Output array.

**Returns** Stacked array.

**Return type** `cupy.ndarray`

**See also:**

`numpy.stack()`

## cupy.vstack

`cupy.vstack(tup)`  
 Stacks arrays vertically.

If an input array has one dimension, then the array is treated as a horizontal vector and stacked along the additional axis at the head. Otherwise, the array is stacked along the first axis.

**Parameters** **tup** (*sequence of arrays*) – Arrays to be stacked. Each array is converted by `cupy.atleast_2d()` before stacking.

**Returns** Stacked array.

**Return type** `cupy.ndarray`

**See also:**

`numpy.dstack()`

## cupy.hstack

`cupy.hstack(tup)`  
 Stacks arrays horizontally.

If an input array has one dimension, then the array is treated as a horizontal vector and stacked along the first axis. Otherwise, the array is stacked along the second axis.

**Parameters** **tup** (*sequence of arrays*) – Arrays to be stacked.

**Returns** Stacked array.

**Return type** `cupy.ndarray`

**See also:**

`numpy.hstack()`

## cupy.dstack

`cupy.dstack(tup)`

Stacks arrays along the third axis.

**Parameters** *tup* (*sequence of arrays*) – Arrays to be stacked. Each array is converted by `cupy.atleast_3d()` before stacking.

**Returns** Stacked array.

**Return type** `cupy.ndarray`

**See also:**

`numpy.dstack()`

## cupy.column\_stack

`cupy.column_stack(tup)`

Stacks 1-D and 2-D arrays as columns into a 2-D array.

A 1-D array is first converted to a 2-D column array. Then, the 2-D arrays are concatenated along the second axis.

**Parameters** *tup* (*sequence of arrays*) – 1-D or 2-D arrays to be stacked.

**Returns** A new 2-D array of stacked columns.

**Return type** `cupy.ndarray`

**See also:**

`numpy.column_stack()`

## Splitting arrays

<code>split(ary, indices_or_sections[, axis])</code>	Splits an array into multiple sub arrays along a given axis.
<code>array_split(ary, indices_or_sections[, axis])</code>	Splits an array into multiple sub arrays along a given axis.
<code>dsplit(ary, indices_or_sections)</code>	Splits an array into multiple sub arrays along the third axis.
<code>hsplit(ary, indices_or_sections)</code>	Splits an array into multiple sub arrays horizontally.
<code>vsplit(ary, indices_or_sections)</code>	Splits an array into multiple sub arrays along the first axis.



## cupy.split

`cupy.split(ary, indices_or_sections, axis=0)`

Splits an array into multiple sub arrays along a given axis.

### Parameters

- **ary** (`cupy.ndarray`) – Array to split.
- **indices\_or\_sections** (*int or sequence of ints*) – A value indicating how to divide the axis. If it is an integer, then is treated as the number of sections, and the axis is evenly divided. Otherwise, the integers indicate indices to split at. Note that the sequence on the device memory is not allowed.
- **axis** (*int*) – Axis along which the array is split.

**Returns** A list of sub arrays. Each array is a view of the corresponding input array.

**See also:**

`numpy.split()`

## cupy.array\_split

`cupy.array_split(ary, indices_or_sections, axis=0)`

Splits an array into multiple sub arrays along a given axis.

This function is almost equivalent to `cupy.split()`. The only difference is that this function allows an integer sections that does not evenly divide the axis.

**See also:**

`cupy.split()` for more detail, `numpy.array_split()`

## cupy.dsplitt

`cupy.dsplitt(ary, indices_or_sections)`

Splits an array into multiple sub arrays along the third axis.

This is equivalent to `split` with `axis=2`.

**See also:**

`cupy.split()` for more detail, `numpy.dsplitt()`

## cupy.hsplitt

`cupy.hsplitt(ary, indices_or_sections)`

Splits an array into multiple sub arrays horizontally.

This is equivalent to `split` with `axis=0` if `ary` has one dimension, and otherwise that with `axis=1`.

**See also:**

`cupy.split()` for more detail, `numpy.hsplitt()`

## cupy.vsplit

`cupy.vsplit(ary, indices_or_sections)`

Splits an array into multiple sub arrays along the first axis.

This is equivalent to `split` with `axis=0`.

See also:

`cupy.split()` for more detail, `numpy.dsplit()`

## Tiling arrays

---

<code>tile(A, reps)</code>	Construct an array by repeating A the number of times given by reps.
<code>repeat(a, repeats[, axis])</code>	Repeat arrays along an axis.

---

## cupy.tile

`cupy.tile(A, reps)`

Construct an array by repeating A the number of times given by reps.

### Parameters

- **A** (`cupy.ndarray`) – Array to transform.
- **reps** (`int` or `tuple`) – The number of repeats.

**Returns** Transformed array with repeats.

**Return type** `cupy.ndarray`

See also:

`numpy.tile()`

## cupy.repeat

`cupy.repeat(a, repeats, axis=None)`

Repeat arrays along an axis.

### Parameters

- **a** (`cupy.ndarray`) – Array to transform.
- **repeats** (`int`, `list` or `tuple`) – The number of repeats.
- **axis** (`int`) – The axis to repeat.

**Returns** Transformed array with repeats.

**Return type** `cupy.ndarray`

See also:

`numpy.repeat()`

## Adding and removing elements

<code>append(arr, values[, axis])</code>	Append values to the end of an array.
<code>resize(a, new_shape)</code>	Return a new array with the specified shape.
<code>unique(ar[, return_index, return_inverse, ...])</code>	Find the unique elements of an array.
<code>trim_zeros(filt[, trim])</code>	Trim the leading and/or trailing zeros from a 1-D array or sequence.

### cupy.append

`cupy.append(arr, values, axis=None)`

Append values to the end of an array.

#### Parameters

- **arr** (*array\_like*) – Values are appended to a copy of this array.
- **values** (*array\_like*) – These values are appended to a copy of `arr`. It must be of the correct shape (the same shape as `arr`, excluding `axis`). If `axis` is not specified, `values` can be any shape and will be flattened before use.
- **axis** (*int or None*) – The axis along which `values` are appended. If `axis` is not given, both `arr` and `values` are flattened before use.

**Returns** A copy of `arr` with `values` appended to `axis`. Note that `append` does not occur in-place: a new array is allocated and filled. If `axis` is `None`, `out` is a flattened array.

**Return type** *cupy.ndarray*

See also:

`numpy.append()`

### cupy.resize

`cupy.resize(a, new_shape)`

Return a new array with the specified shape.

If the new array is larger than the original array, then the new array is filled with repeated copies of `a`. Note that this behavior is different from `a.resize(new_shape)` which fills with zeros instead of repeated copies of `a`.

#### Parameters

- **a** (*array\_like*) – Array to be resized.
- **new\_shape** (*int or tuple of int*) – Shape of resized array.

**Returns** The new array is formed from the data in the old array, repeated if necessary to fill out the required number of elements. The data are repeated in the order that they are stored in memory.

**Return type** *cupy.ndarray*

See also:

`numpy.resize()`

## cupy.unique

`cupy.unique(ar, return_index=False, return_inverse=False, return_counts=False, axis=None)`

Find the unique elements of an array.

Returns the sorted unique elements of an array. There are three optional outputs in addition to the unique elements:

- the indices of the input array that give the unique values
- the indices of the unique array that reconstruct the input array
- the number of times each unique value comes up in the input array

### Parameters

- **ar** (*array\_like*) – Input array. This will be flattened if it is not already 1-D.
- **return\_index** (*bool*, *optional*) – If True, also return the indices of *ar* (along the specified axis, if provided, or in the flattened array) that result in the unique array.
- **return\_inverse** (*bool*, *optional*) – If True, also return the indices of the unique array (for the specified axis, if provided) that can be used to reconstruct *ar*.
- **return\_counts** (*bool*, *optional*) – If True, also return the number of times each unique item appears in *ar*.
- **axis** (*int* or *None*, *optional*) – Not supported yet.

### Returns

If there are no optional outputs, it returns the `cupy.ndarray` of the sorted unique values. Otherwise, it returns the tuple which contains the sorted unique values and followings.

- The indices of the first occurrences of the unique values in the original array. Only provided if *return\_index* is True.
- The indices to reconstruct the original array from the unique array. Only provided if *return\_inverse* is True.
- The number of times each of the unique values comes up in the original array. Only provided if *return\_counts* is True.

**Return type** `cupy.ndarray` or `tuple`

**Warning:** This function may synchronize the device.

See also:

`numpy.unique()`

## cupy.trim\_zeros

`cupy.trim_zeros(filt, trim='fb')`

Trim the leading and/or trailing zeros from a 1-D array or sequence.

Returns the trimmed array

### Parameters

- **filt** (`cupy.ndarray`) – Input array
- **trim** (`str`, *optional*) – ‘fb’ default option trims the array from both sides. ‘f’ option trim zeros from front. ‘b’ option trim zeros from back.

**Returns** trimmed input

**Return type** `cupy.ndarray`

**See also:**

`numpy.trim_zeros()`

## Rearranging elements

<code>flip(a[, axis])</code>	Reverse the order of elements in an array along the given axis.
<code>fliplr(a)</code>	Flip array in the left/right direction.
<code>flipud(a)</code>	Flip array in the up/down direction.
<code>reshape(a, newshape[, order])</code>	Returns an array with new shape and same elements.
<code>roll(a, shift[, axis])</code>	Roll array elements along a given axis.
<code>rot90(a[, k, axes])</code>	Rotate an array by 90 degrees in the plane specified by axes.

## cupy.flip

`cupy.flip(a, axis=None)`

Reverse the order of elements in an array along the given axis.

Note that `flip` function has been introduced since NumPy v1.12. The contents of this document is the same as the original one.

### Parameters

- **a** (`ndarray`) – Input array.
- **axis** (*int or tuple of int or None*) – Axis or axes along which to flip over. The default, `axis=None`, will flip over all of the axes of the input array. If `axis` is negative it counts from the last to the first axis. If `axis` is a tuple of ints, flipping is performed on all of the axes specified in the tuple.

**Returns** Output array.

**Return type** `ndarray`

**See also:**

`numpy.flip()`

## cupy.fliplr

`cupy.fliplr(a)`

Flip array in the left/right direction.

Flip the entries in each row in the left/right direction. Columns are preserved, but appear in a different order than before.

**Parameters** `a` (`ndarray`) – Input array.

**Returns** Output array.

**Return type** `ndarray`

**See also:**

`numpy.fliplr()`

## cupy.flipud

`cupy.flipud(a)`

Flip array in the up/down direction.

Flip the entries in each column in the up/down direction. Rows are preserved, but appear in a different order than before.

**Parameters** `a` (`ndarray`) – Input array.

**Returns** Output array.

**Return type** `ndarray`

**See also:**

`numpy.flipud()`

## cupy.roll

`cupy.roll(a, shift, axis=None)`

Roll array elements along a given axis.

Elements that roll beyond the last position are re-introduced at the first.

**Parameters**

- `a` (`ndarray`) – Array to be rolled.
- `shift` (`int` or `tuple of int`) – The number of places by which elements are shifted. If a tuple, then `axis` must be a tuple of the same size, and each of the given axes is shifted by the corresponding number. If an int while `axis` is a tuple of ints, then the same value is used for all given axes.
- `axis` (`int` or `tuple of int` or `None`) – The axis along which elements are shifted. By default, the array is flattened before shifting, after which the original shape is restored.

**Returns** Output array.

**Return type** `ndarray`

**See also:**

`numpy.roll()`

## cupy.rot90

`cupy.rot90(a, k=1, axes=(0, 1))`

Rotate an array by 90 degrees in the plane specified by axes.

Note that `axes` argument has been introduced since NumPy v1.12. The contents of this document is the same as the original one.

### Parameters

- **a** (`ndarray`) – Array of two or more dimensions.
- **k** (`int`) – Number of times the array is rotated by 90 degrees.
- **axes** – (tuple of ints): The array is rotated in the plane defined by the axes. Axes must be different.

**Returns** Output array.

**Return type** `ndarray`

See also:

`numpy.rot90()`

## 5.3.3 Binary operations

---

**Hint:** NumPy API Reference: Binary operations

---

### Elementwise bit operations

<code>bitwise_and</code>	Computes the bitwise AND of two arrays elementwise.
<code>bitwise_or</code>	Computes the bitwise OR of two arrays elementwise.
<code>bitwise_xor</code>	Computes the bitwise XOR of two arrays elementwise.
<code>invert</code>	Computes the bitwise NOT of an array elementwise.
<code>left_shift</code>	Shifts the bits of each integer element to the left.
<code>right_shift</code>	Shifts the bits of each integer element to the right.

### Bit packing

<code>packbits(myarray)</code>	Packs the elements of a binary-valued array into bits in a uint8 array.
<code>unpackbits(myarray)</code>	Unpacks elements of a uint8 array into a binary-valued output array.

## cupy.packbits

`cupy.packbits(myarray)`

Packs the elements of a binary-valued array into bits in a uint8 array.

This function currently does not support `axis` option.

**Parameters** `myarray` (`cupy.ndarray`) – Input array.

**Returns** The packed array.

**Return type** `cupy.ndarray`

---

**Note:** When the input array is empty, this function returns a copy of it, i.e., the type of the output array is not necessarily always uint8. This exactly follows the NumPy’s behaviour (as of version 1.11), although this is inconsistent to the documentation.

---

**See also:**

`numpy.packbits()`

## cupy.unpackbits

`cupy.unpackbits(myarray)`

Unpacks elements of a uint8 array into a binary-valued output array.

This function currently does not support `axis` option.

**Parameters** `myarray` (`cupy.ndarray`) – Input array.

**Returns** The unpacked array.

**Return type** `cupy.ndarray`

**See also:**

`numpy.unpackbits()`

## Output formatting

---

`binary_repr(num[, width])`

Return the binary representation of the input number as a string.

---

## cupy.binary\_repr

`cupy.binary_repr(num, width=None)`

Return the binary representation of the input number as a string.

**See also:**

`numpy.binary_repr()`



### 5.3.4 Data type routines

---

**Hint:** NumPy API Reference: Data type routines

---

<code>can_cast(from_, to[, casting])</code>	Returns True if cast between data types can occur according to the casting rule.
<code>result_type(*arrays_and_dtypes)</code>	Returns the type that results from applying the NumPy type promotion rules to the arguments.
<code>common_type(*arrays)</code>	Return a scalar type which is common to the input arrays.

---

#### cupy.can\_cast

`cupy.can_cast(from_, to, casting='safe')`

Returns True if cast between data types can occur according to the casting rule. If from is a scalar or array scalar, also returns True if the scalar value can be cast without overflow or truncation to an integer.

**See also:**

`numpy.can_cast()`

#### cupy.result\_type

`cupy.result_type(*arrays_and_dtypes)`

Returns the type that results from applying the NumPy type promotion rules to the arguments.

**See also:**

`numpy.result_type()`

#### cupy.common\_type

`cupy.common_type(*arrays)`

Return a scalar type which is common to the input arrays.

**See also:**

`numpy.common_type()`

<code>promote_types</code> (alias of <code>numpy.promote_types()</code> )
<code>min_scalar_type</code> (alias of <code>numpy.min_scalar_type()</code> )
<code>obj2sctype</code> (alias of <code>numpy.obj2sctype()</code> )

## Creating data types

<code>dtype</code> (alias of <code>numpy.dtype</code> )
<code>format_parser</code> (alias of <code>numpy.format_parser</code> )

## Data type information

<code>finfo</code> (alias of <code>numpy.finfo</code> )
<code>iinfo</code> (alias of <code>numpy.iinfo</code> )
<code>MachAr</code> (alias of <code>numpy.MachAr</code> )

## Data type testing

<code>issctype</code> (alias of <code>numpy.issctype()</code> )
<code>issubdtype</code> (alias of <code>numpy.issubdtype()</code> )
<code>issubscdtype</code> (alias of <code>numpy.issubscdtype()</code> )
<code>issubclass_</code> (alias of <code>numpy.issubclass_()</code> )
<code>find_common_type</code> (alias of <code>numpy.find_common_type()</code> )

## Miscellaneous

<code>typename</code> (alias of <code>numpy.typename()</code> )
<code>sctype2char</code> (alias of <code>numpy.sctype2char()</code> )
<code>mintypecode</code> (alias of <code>numpy.mintypecode()</code> )

## 5.3.5 Discrete Fourier Transform (`cupy.fft`)

---

**Hint:** [NumPy API Reference: Discrete Fourier Transform \(`numpy.fft`\)](#)

---

### See also:

*Discrete Fourier transforms (`cupyx.scipy.fft`), Fast Fourier Transform with CuPy*

### Standard FFTs

<code>fft(a[, n, axis, norm])</code>	Compute the one-dimensional FFT.
<code>ifft(a[, n, axis, norm])</code>	Compute the one-dimensional inverse FFT.
<code>fft2(a[, s, axes, norm])</code>	Compute the two-dimensional FFT.
<code>ifft2(a[, s, axes, norm])</code>	Compute the two-dimensional inverse FFT.
<code>fftn(a[, s, axes, norm])</code>	Compute the N-dimensional FFT.
<code>ifftn(a[, s, axes, norm])</code>	Compute the N-dimensional inverse FFT.

**cupy.fft.fft**

`cupy.fft.fft(a, n=None, axis=-1, norm=None)`

Compute the one-dimensional FFT.

**Parameters**

- **a** (`cupy.ndarray`) – Array to be transform.
- **n** (`None` or `int`) – Length of the transformed axis of the output. If `n` is not given, the length of the input along the axis specified by `axis` is used.
- **axis** (`int`) – Axis over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is `None`, which is an alias of "backward".

**Returns** The transformed array which shape is specified by `n` and type will convert to complex if the input is other.

**Return type** `cupy.ndarray`

See also:

`numpy.fft.fft()`

**cupy.fft.ifft**

`cupy.fft.ifft(a, n=None, axis=-1, norm=None)`

Compute the one-dimensional inverse FFT.

**Parameters**

- **a** (`cupy.ndarray`) – Array to be transform.
- **n** (`None` or `int`) – Length of the transformed axis of the output. If `n` is not given, the length of the input along the axis specified by `axis` is used.
- **axis** (`int`) – Axis over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is `None`, which is an alias of "backward".

**Returns** The transformed array which shape is specified by `n` and type will convert to complex if the input is other.

**Return type** `cupy.ndarray`

See also:

`numpy.fft.ifft()`

## cupy.fft.fft2

`cupy.fft.fft2(a, s=None, axes=(-2, -1), norm=None)`

Compute the two-dimensional FFT.

### Parameters

- **a** (`cupy.ndarray`) – Array to be transform.
- **s** (*None or tuple of ints*) – Shape of the transformed axes of the output. If **s** is not given, the lengths of the input along the axes specified by **axes** are used.
- **axes** (*tuple of ints*) – Axes over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is `None`, which is an alias of "backward".

**Returns** The transformed array which shape is specified by **s** and type will convert to complex if the input is other.

**Return type** `cupy.ndarray`

See also:

`numpy.fft.fft2()`

## cupy.fft.ifft2

`cupy.fft.ifft2(a, s=None, axes=(-2, -1), norm=None)`

Compute the two-dimensional inverse FFT.

### Parameters

- **a** (`cupy.ndarray`) – Array to be transform.
- **s** (*None or tuple of ints*) – Shape of the transformed axes of the output. If **s** is not given, the lengths of the input along the axes specified by **axes** are used.
- **axes** (*tuple of ints*) – Axes over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is `None`, which is an alias of "backward".

**Returns** The transformed array which shape is specified by **s** and type will convert to complex if the input is other.

**Return type** `cupy.ndarray`

See also:

`numpy.fft.ifft2()`

**cupy.fft.fftn**

`cupy.fft.fftn(a, s=None, axes=None, norm=None)`

Compute the N-dimensional FFT.

**Parameters**

- **a** (`cupy.ndarray`) – Array to be transform.
- **s** (*None or tuple of ints*) – Shape of the transformed axes of the output. If **s** is not given, the lengths of the input along the axes specified by **axes** are used.
- **axes** (*tuple of ints*) – Axes over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is `None`, which is an alias of "backward".

**Returns** The transformed array which shape is specified by **s** and type will convert to complex if the input is other.

**Return type** `cupy.ndarray`

See also:

`numpy.fft.fftn()`

**cupy.fft.ifftn**

`cupy.fft.ifftn(a, s=None, axes=None, norm=None)`

Compute the N-dimensional inverse FFT.

**Parameters**

- **a** (`cupy.ndarray`) – Array to be transform.
- **s** (*None or tuple of ints*) – Shape of the transformed axes of the output. If **s** is not given, the lengths of the input along the axes specified by **axes** are used.
- **axes** (*tuple of ints*) – Axes over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is `None`, which is an alias of "backward".

**Returns** The transformed array which shape is specified by **s** and type will convert to complex if the input is other.

**Return type** `cupy.ndarray`

See also:

`numpy.fft.ifftn()`

## Real FFTs

<code>rfft(a[, n, axis, norm])</code>	Compute the one-dimensional FFT for real input.
<code>irfft(a[, n, axis, norm])</code>	Compute the one-dimensional inverse FFT for real input.
<code>rfft2(a[, s, axes, norm])</code>	Compute the two-dimensional FFT for real input.
<code>irfft2(a[, s, axes, norm])</code>	Compute the two-dimensional inverse FFT for real input.
<code>rfftn(a[, s, axes, norm])</code>	Compute the N-dimensional FFT for real input.
<code>irfftn(a[, s, axes, norm])</code>	Compute the N-dimensional inverse FFT for real input.

### `cupy.fft.rfft`

`cupy.fft.rfft(a, n=None, axis=-1, norm=None)`  
Compute the one-dimensional FFT for real input.

#### Parameters

- **a** (`cupy.ndarray`) – Array to be transform.
- **n** (*None* or `int`) – Number of points along transformation axis in the input to use. If **n** is not given, the length of the input along the axis specified by **axis** is used.
- **axis** (`int`) – Axis over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is *None*, which is an alias of "backward".

**Returns** The transformed array which shape is specified by **n** and type will convert to complex if the input is other. The length of the transformed axis is  $n//2+1$ .

**Return type** `cupy.ndarray`

See also:

`numpy.fft.rfft()`

### `cupy.fft.irfft`

`cupy.fft.irfft(a, n=None, axis=-1, norm=None)`  
Compute the one-dimensional inverse FFT for real input.

#### Parameters

- **a** (`cupy.ndarray`) – Array to be transform.
- **n** (*None* or `int`) – Length of the transformed axis of the output. For **n** output points,  $n//2+1$  input points are necessary. If **n** is not given, it is determined from the length of the input along the axis specified by **axis**.
- **axis** (`int`) – Axis over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is *None*, which is an alias of "backward".

**Returns** The transformed array which shape is specified by **n** and type will convert to complex if the input is other. If **n** is not given, the length of the transformed axis is  $2*(m-1)$  where *m* is the length of the transformed axis of the input.

**Return type** `cupy.ndarray`

See also:

`numpy.fft.irfft()`

### `cupy.fft.rfft2`

`cupy.fft.rfft2(a, s=None, axes=(-2, -1), norm=None)`

Compute the two-dimensional FFT for real input.

#### Parameters

- **a** (`cupy.ndarray`) – Array to be transform.
- **s** (*None or tuple of ints*) – Shape to use from the input. If **s** is not given, the lengths of the input along the axes specified by **axes** are used.
- **axes** (*tuple of ints*) – Axes over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is **None**, which is an alias of "backward".

**Returns** The transformed array which shape is specified by **s** and type will convert to complex if the input is other. The length of the last axis transformed will be  $s[-1]//2+1$ .

**Return type** `cupy.ndarray`

See also:

`numpy.fft.rfft2()`

### `cupy.fft.irfft2`

`cupy.fft.irfft2(a, s=None, axes=(-2, -1), norm=None)`

Compute the two-dimensional inverse FFT for real input.

#### Parameters

- **a** (`cupy.ndarray`) – Array to be transform.
- **s** (*None or tuple of ints*) – Shape of the output. If **s** is not given, they are determined from the lengths of the input along the axes specified by **axes**.
- **axes** (*tuple of ints*) – Axes over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is **None**, which is an alias of "backward".

**Returns** The transformed array which shape is specified by **s** and type will convert to complex if the input is other. If **s** is not given, the length of final transformed axis of output will be  $2*(m-1)$  where  $m$  is the length of the final transformed axis of the input.

**Return type** `cupy.ndarray`

See also:

`numpy.fft.irfft2()`

## cupy.fft.rfftn

`cupy.fft.rfftn(a, s=None, axes=None, norm=None)`

Compute the N-dimensional FFT for real input.

### Parameters

- **a** (`cupy.ndarray`) – Array to be transform.
- **s** (*None or tuple of ints*) – Shape to use from the input. If *s* is not given, the lengths of the input along the axes specified by *axes* are used.
- **axes** (*tuple of ints*) – Axes over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is *None*, which is an alias of "backward".

**Returns** The transformed array which shape is specified by *s* and type will convert to complex if the input is other. The length of the last axis transformed will be  $s[-1]/2+1$ .

**Return type** `cupy.ndarray`

See also:

`numpy.fft.rfftn()`

## cupy.fft.irfftn

`cupy.fft.irfftn(a, s=None, axes=None, norm=None)`

Compute the N-dimensional inverse FFT for real input.

### Parameters

- **a** (`cupy.ndarray`) – Array to be transform.
- **s** (*None or tuple of ints*) – Shape of the output. If *s* is not given, they are determined from the lengths of the input along the axes specified by *axes*.
- **axes** (*tuple of ints*) – Axes over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is *None*, which is an alias of "backward".

**Returns** The transformed array which shape is specified by *s* and type will convert to complex if the input is other. If *s* is not given, the length of final transformed axis of output will be  $2*(m-1)$  where *m* is the length of the final transformed axis of the input.

**Return type** `cupy.ndarray`

See also:

`numpy.fft.irfftn()`



## Hermitian FFTs

<code>hfft(a[, n, axis, norm])</code>	Compute the FFT of a signal that has Hermitian symmetry.
<code>ihfft(a[, n, axis, norm])</code>	Compute the FFT of a signal that has Hermitian symmetry.

### cupy.fft.hfft

`cupy.fft.hfft(a, n=None, axis=-1, norm=None)`

Compute the FFT of a signal that has Hermitian symmetry.

#### Parameters

- **a** (`cupy.ndarray`) – Array to be transform.
- **n** (`None` or `int`) – Length of the transformed axis of the output. For  $n$  output points,  $n//2+1$  input points are necessary. If  $n$  is not given, it is determined from the length of the input along the axis specified by `axis`.
- **axis** (`int`) – Axis over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is `None`, which is an alias of "backward".

**Returns** The transformed array which shape is specified by  $n$  and type will convert to complex if the input is other. If  $n$  is not given, the length of the transformed axis is  $2*(m-1)$  where  $m$  is the length of the transformed axis of the input.

**Return type** `cupy.ndarray`

See also:

`numpy.fft.hfft()`

### cupy.fft.ihfft

`cupy.fft.ihfft(a, n=None, axis=-1, norm=None)`

Compute the FFT of a signal that has Hermitian symmetry.

#### Parameters

- **a** (`cupy.ndarray`) – Array to be transform.
- **n** (`None` or `int`) – Number of points along transformation axis in the input to use. If  $n$  is not given, the length of the input along the axis specified by `axis` is used.
- **axis** (`int`) – Axis over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is `None`, which is an alias of "backward".

**Returns** The transformed array which shape is specified by  $n$  and type will convert to complex if the input is other. The length of the transformed axis is  $n//2+1$ .

**Return type** `cupy.ndarray`

See also:

`numpy.fft.ihfft()`

## Helper routines

<code>fftfreq(n[, d])</code>	Return the FFT sample frequencies.
<code>rfftfreq(n[, d])</code>	Return the FFT sample frequencies for real input.
<code>fftshift(x[, axes])</code>	Shift the zero-frequency component to the center of the spectrum.
<code>ifftshift(x[, axes])</code>	The inverse of <code>fftshift()</code> .

### `cupy.fft.fftfreq`

`cupy.fft.fftfreq(n, d=1.0)`

Return the FFT sample frequencies.

**Parameters**

- **n** (*int*) – Window length.
- **d** (*scalar*) – Sample spacing.

**Returns** Array of length **n** containing the sample frequencies.

**Return type** *cupy.ndarray*

**See also:**

`numpy.fft.fftfreq()`

### `cupy.fft.rfftfreq`

`cupy.fft.rfftfreq(n, d=1.0)`

Return the FFT sample frequencies for real input.

**Parameters**

- **n** (*int*) – Window length.
- **d** (*scalar*) – Sample spacing.

**Returns** Array of length  $n//2+1$  containing the sample frequencies.

**Return type** *cupy.ndarray*

**See also:**

`numpy.fft.rfftfreq()`

### `cupy.fft.fftshift`

`cupy.fft.fftshift(x, axes=None)`

Shift the zero-frequency component to the center of the spectrum.

**Parameters**

- **x** (*cupy.ndarray*) – Input array.
- **axes** (*int or tuple of ints*) – Axes over which to shift. Default is `None`, which shifts all axes.

**Returns** The shifted array.

**Return type** `cupy.ndarray`

**See also:**

`numpy.fft.fftshift()`

## `cupy.fft.ifftshift`

`cupy.fft.ifftshift(x, axes=None)`

The inverse of `fftshift()`.

### Parameters

- **x** (`cupy.ndarray`) – Input array.
- **axes** (`int` or `tuple of ints`) – Axes over which to shift. Default is `None`, which shifts all axes.

**Returns** The shifted array.

**Return type** `cupy.ndarray`

**See also:**

`numpy.fft.ifftshift()`

## CuPy-specific APIs

See the description below for details.

<code>config.set_cufft_callbacks(...)</code>	A context manager for setting up load and/or store callbacks.
<code>config.set_cufft_gpus(gpus)</code>	Set the GPUs to be used in multi-GPU FFT.
<code>config.get_plan_cache()</code>	Get the per-thread, per-device plan cache, or create one if not found.
<code>config.show_plan_cache_info()</code>	Show all of the plan caches' info on this thread.

## `cupy.fft.config.set_cufft_callbacks`

**class** `cupy.fft.config.set_cufft_callbacks(unicode cb_load=u'', unicode cb_store=u'', ndarray cb_load_aux_arr=None, *, ndarray cb_store_aux_arr=None)`

A context manager for setting up load and/or store callbacks.

### Parameters

- **cb\_load** (`str`) – A string contains the device kernel for the load callback. It must define `d_loadCallbackPtr`.
- **cb\_store** (`str`) – A string contains the device kernel for the store callback. It must define `d_storeCallbackPtr`.
- **cb\_load\_aux\_arr** (`cupy.ndarray`, *optional*) – A CuPy array containing data to be used in the load callback.
- **cb\_store\_aux\_arr** (`cupy.ndarray`, *optional*) – A CuPy array containing data to be used in the store callback.

**Note:** Any FFT calls living in this context will have callbacks set up. An example for a load callback is shown below:

```
code = r'''
__device__ cufftComplex CB_ConvertInputC(
    void *dataIn,
    size_t offset,
    void *callerInfo,
    void *sharedPtr) {
    // implementation
}

__device__ cufftCallbackLoadC d_loadCallbackPtr = CB_ConvertInputC;
'''

with cp.fft.config.set_cufft_callbacks(cb_load=code):
    out_arr = cp.fft.fft(in_arr, ...)
```

---

**Note:** Below are the *runtime* requirements for using this feature:

- cython >= 0.29.0
- A host compiler that supports C++11 and above; might need to set up the CXX environment variable.
- nvcc and the full CUDA Toolkit. Note that the cudatoolkit package from Conda-Forge is not enough, as it does not contain static libraries.

---

**Note:** Callbacks only work for transforms over contiguous axes; the behavior for non-contiguous transforms is in general undefined.

---

**Warning:** Using cuFFT callbacks requires compiling and loading a Python module at runtime as well as static linking for each distinct transform and callback, so the first invocation for each combination will be very slow. This is a limitation of cuFFT, so use this feature only when the callback-enabled transform is known more performant and can be reused to amortize the cost.

**Warning:** The generated Python modules are by default cached in `~/ .cupy/callback_cache` for possible reuse (with the same set of load/store callbacks). Due to static linking, however, the file sizes can be excessive! The cache position can be changed via setting `CUPY_CACHE_DIR`.

**See also:**

[cuFFT Callback Routines](#)

## Methods

```

__enter__(self)
__exit__(self, exc_type, exc_value, traceback)
__eq__(value, /)
    Return self==value.
__ne__(value, /)
    Return self!=value.
__lt__(value, /)
    Return self<value.
__le__(value, /)
    Return self<=value.
__gt__(value, /)
    Return self>value.
__ge__(value, /)
    Return self>=value.

```

## cupy.fft.config.set\_cufft\_gpus

`cupy.fft.config.set_cufft_gpus(gpus)`  
 Set the GPUs to be used in multi-GPU FFT.

**Parameters** `gpus` (*int or list of int*) – The number of GPUs or a list of GPUs to be used.  
 For the former case, the first `gpus` GPUs will be used.

**Warning:** This API is currently experimental and may be changed in the future version.

See also:

[Multiple GPU cuFFT Transforms](#)

## cupy.fft.config.get\_plan\_cache

`cupy.fft.config.get_plan_cache()` → PlanCache  
 Get the per-thread, per-device plan cache, or create one if not found.

See also:

PlanCache

## cupy.fft.config.show\_plan\_cache\_info

`cupy.fft.config.show_plan_cache_info()`

Show all of the plan caches' info on this thread.

**See also:**

PlanCache

## Normalization

The default normalization (`norm` is "backward" or `None`) has the direct transforms unscaled and the inverse transforms scaled by  $1/n$ . If the keyword argument `norm` is "forward", it is the exact opposite of "backward": the direct transforms are scaled by  $1/n$  and the inverse transforms are unscaled. Finally, if the keyword argument `norm` is "ortho", both transforms are scaled by  $1/\sqrt{n}$ .

## Code compatibility features

FFT functions of NumPy always return `numpy.ndarray` which type is `numpy.complex128` or `numpy.float64`. CuPy functions do not follow the behavior, they will return `numpy.complex64` or `numpy.float32` if the type of the input is `numpy.float16`, `numpy.float32`, or `numpy.complex64`.

Internally, `cupy.fft` always generates a *cuFFT plan* (see the [cuFFT documentation](#) for detail) corresponding to the desired transform. When possible, an n-dimensional plan will be used, as opposed to applying separate 1D plans for each axis to be transformed. Using n-dimensional planning can provide better performance for multidimensional transforms, but requires more GPU memory than separable 1D planning. The user can disable n-dimensional planning by setting `cupy.fft.config.enable_nd_planning = False`. This ability to adjust the planning type is a deviation from the NumPy API, which does not use precomputed FFT plans.

Moreover, the automatic plan generation can be suppressed by using an existing plan returned by `cupyx.scipy.fftpack.get_fft_plan()` as a context manager. This is again a deviation from NumPy.

Finally, when using the high-level NumPy-like FFT APIs as listed above, internally the cuFFT plans are cached for possible reuse. The plan cache can be retrieved by `get_plan_cache()`, and its current status can be queried by `show_plan_cache_info()`. For finer control of the plan cache, see `plan_cache`.

## Multi-GPU FFT

`cupy.fft` can use multiple GPUs. To enable (disable) this feature, set `cupy.fft.config.use_multi_gpus` to `True` (`False`). Next, to set the number of GPUs or the participating GPU IDs, use the function `cupy.fft.config.set_cufft_gpus()`. All of the limitations listed in the [cuFFT documentation](#) apply here. In particular, using more than one GPU does not guarantee better performance.

## 5.3.6 Functional programming

---

**Hint:** [NumPy API Reference: Functional programming](#)

---

---

**Note:** `cupy.vectorize` applies JIT compiler to the given Python function. See [JIT kernel definition](#) for details.

---

<code>apply_along_axis(func1d, axis, arr, *args, ...)</code>	Apply a function to 1-D slices along the given axis.
<code>vectorize(pyfunc[, otypes, doc, excluded, ...])</code>	Generalized function class.
<code>piecewise(x, condlist, funclist)</code>	Evaluate a piecewise-defined function.

## cupy.apply\_along\_axis

`cupy.apply_along_axis(func1d, axis, arr, *args, **kwargs)`

Apply a function to 1-D slices along the given axis.

### Parameters

- **func1d** (*function* ( $M, \dots \rightarrow Nj, \dots$ )) – This function should accept 1-D arrays. It is applied to 1-D slices of `arr` along the specified axis. It must return a 1-D `cupy.ndarray`.
- **axis** (*integer*) – Axis along which `arr` is sliced.
- **arr** (`cupy.ndarray` ( $Ni, \dots, M, Nk, \dots$ )) – Input array.
- **args** – Additional arguments for `func1d`.
- **kwargs** – Additional keyword arguments for `func1d`.

**Returns** The output array. The shape of `out` is identical to the shape of `arr`, except along the `axis` dimension. This axis is removed, and replaced with new dimensions equal to the shape of the return value of `func1d`. So if `func1d` returns a scalar `out` will have one fewer dimensions than `arr`.

**Return type** `cupy.ndarray`

See also:

`numpy.apply_along_axis()`

## cupy.vectorize

**class** `cupy.vectorize(pyfunc, otypes=None, doc=None, excluded=None, cache=False, signature=None)`

Generalized function class.

See also:

`numpy.vectorize`

### Methods

`__call__(*args)`

Call self as a function.

`__eq__(value, /)`

Return self==value.

`__ne__(value, /)`

Return self!=value.

`__lt__(value, /)`

Return self<value.

`__le__(value, /)`

Return self<=value.

`__gt__(value, /)`  
Return self>value.

`__ge__(value, /)`  
Return self>=value.

## cupy.piecewise

`cupy.piecewise(x, condlist, funclist)`  
Evaluate a piecewise-defined function.

### Parameters

- **x** (`cupy.ndarray`) – input domain
- **condlist** (*list of cupy.ndarray*) – Each boolean array/ scalar corresponds to a function in funclist. Length of funclist is equal to that of condlist. If one extra function is given, it is used as the default value when the otherwise condition is met
- **funclist** (*list of scalars*) – list of scalar functions.

**Returns** the scalar values in funclist on portions of x defined by condlist.

**Return type** `cupy.ndarray`

**Warning:** This function currently doesn't support callable functions, args and kw parameters.

See also:

`numpy.piecewise()`

## 5.3.7 Indexing routines

---

**Hint:** NumPy API Reference: Indexing routines

---

### Generating index arrays

---

<code>c_</code>	
<code>r_</code>	
<code>nonzero(a)</code>	Return the indices of the elements that are non-zero.
<code>where(condition[, x, y])</code>	Return elements, either from x or y, depending on condition.
<code>indices(dimensions[, dtype])</code>	Returns an array representing the indices of a grid.
<code>ix_(*args)</code>	Construct an open mesh from multiple sequences.
<code>ravel_multi_index(multi_index, dims[, mode, ...])</code>	Converts a tuple of index arrays into an array of flat indices, applying boundary modes to the multi-index.
<code>unravel_index(indices, dims[, order])</code>	Converts array of flat indices into a tuple of coordinate arrays.

---

continues on next page



Table 35 – continued from previous page

<code>diag_indices(n[, ndim])</code>	Return the indices to access the main diagonal of an array.
<code>diag_indices_from(arr)</code>	Return the indices to access the main diagonal of an n-dimensional array.

**cupy.c\_**

`cupy.c_` = <cupy.\_indexing.generate.CClass object>

**cupy.r\_**

`cupy.r_` = <cupy.\_indexing.generate.RClass object>

**cupy.nonzero**

`cupy.nonzero(a)`

Return the indices of the elements that are non-zero.

Returns a tuple of arrays, one for each dimension of `a`, containing the indices of the non-zero elements in that dimension.

**Parameters** `a` (`cupy.ndarray`) – array

**Returns** Indices of elements that are non-zero.

**Return type** tuple of arrays

**Warning:** This function may synchronize the device.

See also:

`numpy.nonzero()`

**cupy.where**

`cupy.where(condition, x=None, y=None)`

Return elements, either from `x` or `y`, depending on condition.

If only condition is given, return `condition.nonzero()`.

**Parameters**

- **condition** (`cupy.ndarray`) – When True, take `x`, otherwise take `y`.
- **x** (`cupy.ndarray`) – Values from which to choose on True.
- **y** (`cupy.ndarray`) – Values from which to choose on False.

**Returns** Each element of output contains elements of `x` when `condition` is True, otherwise elements of `y`. If only `condition` is given, return the tuple `condition.nonzero()`, the indices where `condition` is True.

**Return type** `cupy.ndarray`

**Warning:** This function may synchronize the device if both `x` and `y` are omitted.

See also:

`numpy.where()`

## cupy.indices

`cupy.indices(dimensions, dtype=<class 'int'>)`

Returns an array representing the indices of a grid.

Computes an array where the subarrays contain index values 0,1,... varying only along the corresponding axis.

### Parameters

- **dimensions** – The shape of the grid.
- **dtype** – Data type specifier. It is int by default.

**Returns** The array of grid indices, `grid.shape = (len(dimensions),) + tuple(dimensions)`.

**Return type** `ndarray`

## Examples

```
>>> grid = cupy.indices((2, 3))
>>> grid.shape
(2, 2, 3)
>>> grid[0]          # row indices
array([[0, 0, 0],
       [1, 1, 1]])
>>> grid[1]          # column indices
array([[0, 1, 2],
       [0, 1, 2]])
```

See also:

`numpy.indices()`

## cupy.ix\_

`cupy.ix_(*args)`

Construct an open mesh from multiple sequences.

This function takes N 1-D sequences and returns N outputs with N dimensions each, such that the shape is 1 in all but one dimension and the dimension with the non-unit shape value cycles through all N dimensions.

Using `ix_` one can quickly construct index arrays that will index the cross product. `a[cupy.ix_([1,3],[2,5])]` returns the array `[[a[1,2] a[1,5]], [a[3,2] a[3,5]]]`.

**Parameters** `*args` – 1-D sequences

**Returns** N arrays with N dimensions each, with N the number of input sequences. Together these arrays form an open mesh.

**Return type** tuple of ndarrays

### Examples

```
>>> a = cupy.arange(10).reshape(2, 5)
>>> a
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
>>> ixgrid = cupy.ix_([0,1], [2,4])
>>> ixgrid
(array([[0],
       [1]]), array([[2, 4]]))
```

**Warning:** This function may synchronize the device.

**See also:**

`numpy.ix_()`

### `cupy.ravel_multi_index`

`cupy.ravel_multi_index(multi_index, dims, mode='wrap', order='C')`

Converts a tuple of index arrays into an array of flat indices, applying boundary modes to the multi-index.

#### Parameters

- **multi\_index** (*tuple of cupy.ndarray*) – A tuple of integer arrays, one array for each dimension.
- **dims** (*tuple of ints*) – The shape of array into which the indices from `multi_index` apply.
- **mode** (`'raise'`, `'wrap'` or `'clip'`) – Specifies how out-of-bounds indices are handled. Can specify either one mode or a tuple of modes, one mode per index:
  - `'raise'` – raise an error
  - `'wrap'` – wrap around (default)
  - `'clip'` – clip to the range
 In `'clip'` mode, a negative index which would normally wrap will clip to 0 instead.
- **order** (`'C'` or `'F'`) – Determines whether the multi-index should be viewed as indexing in row-major (C-style) or column-major (Fortran-style) order.

**Returns** An array of indices into the flattened version of an array of dimensions `dims`.

**Return type** raveled\_indices (*cupy.ndarray*)

**Warning:** This function may synchronize the device when `mode == 'raise'`.

## Notes

Note that the default *mode* ('wrap') is different than in NumPy. This is done to avoid potential device synchronization.

## Examples

```
>>> cupy.ravel_multi_index(cupy.asarray([[3,6,6],[4,5,1]]), (7,6))
array([22, 41, 37])
>>> cupy.ravel_multi_index(cupy.asarray([[3,6,6],[4,5,1]]), (7,6),
...                          order='F')
array([31, 41, 13])
>>> cupy.ravel_multi_index(cupy.asarray([[3,6,6],[4,5,1]]), (4,6),
...                          mode='clip')
array([22, 23, 19])
>>> cupy.ravel_multi_index(cupy.asarray([[3,6,6],[4,5,1]]), (4,4),
...                          mode=('clip', 'wrap'))
array([12, 13, 13])
>>> cupy.ravel_multi_index(cupy.asarray((3,1,4,1)), (6,7,8,9))
array(1621)
```

See also:

`numpy.ravel_multi_index()`, `unravel_index()`

## cupy.unravel\_index

`cupy.unravel_index(indices, dims, order='C')`

Converts array of flat indices into a tuple of coordinate arrays.

### Parameters

- **indices** (`cupy.ndarray`) – An integer array whose elements are indices into the flattened version of an array of dimensions `dims`.
- **dims** (*tuple of ints*) – The shape of the array to use for unraveling indices.
- **order** ('C' or 'F') – Determines whether the indices should be viewed as indexing in row-major (C-style) or column-major (Fortran-style) order.

**Returns** Each array in the tuple has the same shape as the indices array.

**Return type** tuple of ndarrays

## Examples

```
>>> cupy.unravel_index(cupy.array([22, 41, 37]), (7, 6))
(array([3, 6, 6]), array([4, 5, 1]))
>>> cupy.unravel_index(cupy.array([31, 41, 13]), (7, 6), order='F')
(array([3, 6, 6]), array([4, 5, 1]))
```

**Warning:** This function may synchronize the device.

See also:

`numpy.unravel_index()`, `ravel_multi_index()`

## `cupy.diag_indices`

`cupy.diag_indices(n, ndim=2)`

Return the indices to access the main diagonal of an array.

Returns a tuple of indices that can be used to access the main diagonal of an array with `ndim >= 2` dimensions and shape `(n, n, ..., n)`.

### Parameters

- **n** (*int*) – The size, along each dimension of the arrays for which the indices are to be returned.
- **ndim** (*int*) – The number of dimensions. default 2.

## Examples

Create a set of indices to access the diagonal of a (4, 4) array:

```
>>> di = cupy.diag_indices(4)
>>> di
(array([0, 1, 2, 3]), array([0, 1, 2, 3]))
>>> a = cupy.arange(16).reshape(4, 4)
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
>>> a[di] = 100
>>> a
array([[100,  1,  2,  3],
       [ 4, 100,  6,  7],
       [ 8,  9, 100, 11],
       [12, 13, 14, 100]])
```

Create indices to manipulate a 3-D array:

```
>>> d3 = cupy.diag_indices(2, 3)
>>> d3
(array([0, 1]), array([0, 1]), array([0, 1]))
```

And use it to set the diagonal of an array of zeros to 1:

```
>>> a = cupy.zeros((2, 2, 2), dtype=int)
>>> a[d3] = 1
>>> a
array([[[1, 0],
       [0, 0]],
       [[0, 0],
       [0, 1]]])
```

See also:

`numpy.diag_indices()`

## `cupy.diag_indices_from`

`cupy.diag_indices_from(arr)`

Return the indices to access the main diagonal of an n-dimensional array. See *diag\_indices* for full details.

**Parameters** `arr` (`cupy.ndarray`) – At least 2-D.

See also:

`numpy.diag_indices_from()`

## Indexing-like operations

<code>take(a, indices[, axis, out])</code>	Takes elements of an array at specified indices along an axis.	
<code>take_along_axis(a, indices, axis)</code>	Take values from the input array by matching 1d index and data slices.	
<code>choose(a, choices[, out, mode])</code>		
<code>compress(condition, a[, axis, out])</code>	Returns selected slices of an array along given axis.	
<code>diag(v[, k])</code>	Returns a diagonal or a diagonal array.	
<code>diagonal(a[, offset, axis1, axis2])</code>	Returns specified diagonals.	
<code>select(condlist, choicelist[, default])</code>	Return an array drawn from elements in choicelist, depending on conditions.	
<code>lib.stride_tricks.as_strided(x[, shape, strides])</code>	shape,	Create a view into the array with the given shape and strides.

## `cupy.take`

`cupy.take(a, indices, axis=None, out=None)`

Takes elements of an array at specified indices along an axis.

This is an implementation of “fancy indexing” at single axis.

This function does not support mode option.

### Parameters

- **a** (`cupy.ndarray`) – Array to extract elements.
- **indices** (`int` or *array-like*) – Indices of elements that this function takes.
- **axis** (`int`) – The axis along which to select indices. The flattened input is used by default.
- **out** (`cupy.ndarray`) – Output array. If provided, it should be of appropriate shape and dtype.

**Returns** The result of fancy indexing.

**Return type** `cupy.ndarray`

See also:

`numpy.take()`

### `cupy.take_along_axis`

`cupy.take_along_axis(a, indices, axis)`

Take values from the input array by matching 1d index and data slices.

#### Parameters

- **a** (`cupy.ndarray`) – Array to extract elements.
- **indices** (`cupy.ndarray`) – Indices to take along each 1d slice of a.
- **axis** (`int`) – The axis to take 1d slices along.

**Returns** The indexed result.

**Return type** `cupy.ndarray`

See also:

`numpy.take_along_axis()`

### `cupy.choose`

`cupy.choose(a, choices, out=None, mode='raise')`

### `cupy.compress`

`cupy.compress(condition, a, axis=None, out=None)`

Returns selected slices of an array along given axis.

#### Parameters

- **condition** (*1-D array of bools*) – Array that selects which entries to return. If `len(condition)` is less than the size of `a` along the given axis, then output is truncated to the length of the condition array.
- **a** (`cupy.ndarray`) – Array from which to extract a part.
- **axis** (`int`) – Axis along which to take slices. If `None` (default), work on the flattened array.
- **out** (`cupy.ndarray`) – Output array. If provided, it should be of appropriate shape and dtype.

**Returns** A copy of `a` without the slices along `axis` for which `condition` is false.

**Return type** `cupy.ndarray`

**Warning:** This function may synchronize the device.

See also:

`numpy.compress()`

## cupy.diagonal

`cupy.diagonal(a, offset=0, axis1=0, axis2=1)`

Returns specified diagonals.

This function extracts the diagonals along two specified axes. The other axes are not changed. This function returns a writable view of this array as NumPy 1.10 will do.

### Parameters

- **a** (`cupy.ndarray`) – Array from which the diagonals are taken.
- **offset** (`int`) – Index of the diagonals. Zero indicates the main diagonals, a positive value upper diagonals, and a negative value lower diagonals.
- **axis1** (`int`) – The first axis to take diagonals from.
- **axis2** (`int`) – The second axis to take diagonals from.

**Returns** A view of the diagonals of *a*.

**Return type** `cupy.ndarray`

**See also:**

`numpy.diagonal()`

## cupy.select

`cupy.select(condlist, choicelist, default=0)`

Return an array drawn from elements in *choicelist*, depending on conditions.

### Parameters

- **condlist** (*list of bool arrays*) – The list of conditions which determine from which array in *choicelist* the output elements are taken. When multiple conditions are satisfied, the first one encountered in *condlist* is used.
- **choicelist** (*list of cupy.ndarray*) – The list of arrays from which the output elements are taken. It has to be of the same length as *condlist*.
- **default** (*scalar*) – If provided, will fill element inserted in *output* when all conditions evaluate to False. default value is 0.

**Returns** The output at position *m* is the *m*-th element of the array in *choicelist* where the *m*-th element of the corresponding array in *condlist* is True.

**Return type** `cupy.ndarray`

**See also:**

`numpy.select()`



**cupy.lib.stride\_tricks.as\_strided**

`cupy.lib.stride_tricks.as_strided(x, shape=None, strides=None)`

Create a view into the array with the given shape and strides.

**Warning:** This function has to be used with extreme care, see notes.

**Parameters**

- **x** (`ndarray`) – Array to create a new.
- **shape** (*sequence of int, optional*) – The shape of the new array. Defaults to `x.shape`.
- **strides** (*sequence of int, optional*) – The strides of the new array. Defaults to `x.strides`.

**Returns** `view`

**Return type** `ndarray`

**See also:**

`numpy.lib.stride_tricks.as_strided`

**reshape** reshape an array.

**Notes**

`as_strided` creates a view into the array given the exact strides and shape. This means it manipulates the internal data structure of `ndarray` and, if done incorrectly, the array elements can point to invalid memory and can corrupt results or crash your program.

**Inserting data into arrays**

<code>place(arr, mask, vals)</code>	Change elements of an array based on conditional and input values.
<code>put(a, ind, v[, mode])</code>	Replaces specified elements of an array with given values.
<code>putmask(a, mask, values)</code>	Changes elements of an array inplace, based on a conditional mask and input values.
<code>fill_diagonal(a, val[, wrap])</code>	Fills the main diagonal of the given array of any dimensionality.

## cupy.place

`cupy.place(arr, mask, vals)`

Change elements of an array based on conditional and input values.

This function uses the first N elements of *vals*, where N is the number of true values in *mask*.

### Parameters

- **arr** (`cupy.ndarray`) – Array to put data into.
- **mask** (*array-like*) – Boolean mask array. Must have the same size as *a*.
- **vals** (*array-like*) – Values to put into *a*. Only the first N elements are used, where N is the number of True values in *mask*. If *vals* is smaller than N, it will be repeated, and if elements of *a* are to be masked, this sequence must be non-empty.

### Examples

```
>>> arr = np.arange(6).reshape(2, 3)
>>> np.place(arr, arr>2, [44, 55])
>>> arr
array([[ 0,  1,  2],
       [44, 55, 44]])
```

**Warning:** This function may synchronize the device.

See also:

`numpy.place()`

## cupy.put

`cupy.put(a, ind, v, mode='wrap')`

Replaces specified elements of an array with given values.

### Parameters

- **a** (`cupy.ndarray`) – Target array.
- **ind** (*array-like*) – Target indices, interpreted as integers.
- **v** (*array-like*) – Values to place in *a* at target indices. If *v* is shorter than *ind* it will be repeated as necessary.
- **mode** (*str*) – How out-of-bounds indices will behave. Its value must be either `'raise'`, `'wrap'` or `'clip'`. Otherwise, `TypeError` is raised.

---

**Note:** Default *mode* is set to `'wrap'` to avoid unintended performance drop. If you need NumPy's behavior, please pass *mode*=`'raise'` manually.

---

See also:

`numpy.put()`

## cupy.putmask

`cupy.putmask(a, mask, values)`

Changes elements of an array inplace, based on a conditional mask and input values.

Sets `a.flat[n] = values[n]` for each `n` where `mask.flat[n]==True`. If `values` is not the same size as `a` and `mask` then it will repeat.

### Parameters

- **a** (`cupy.ndarray`) – Target array.
- **mask** (`cupy.ndarray`) – Boolean mask array. It has to be the same shape as `a`.
- **values** (`cupy.ndarray` or `scalar`) – Values to put into `a` where `mask` is True. If `values` is smaller than `a`, then it will be repeated.

### Examples

```
>>> x = cupy.arange(6).reshape(2, 3)
>>> cupy.putmask(x, x>2, x**2)
>>> x
array([[ 0,  1,  2],
       [ 9, 16, 25]])
```

If `values` is smaller than `a` it is repeated:

```
>>> x = cupy.arange(6)
>>> cupy.putmask(x, x>2, cupy.array([-33, -44]))
>>> x
array([ 0,  1,  2, -44, -33, -44])
```

See also:

`numpy.putmask()`

## cupy.fill\_diagonal

`cupy.fill_diagonal(a, val, wrap=False)`

Fills the main diagonal of the given array of any dimensionality.

For an array `a` with `a.ndim > 2`, the diagonal is the list of locations with indices `a[i, i, ..., i]` all identical. This function modifies the input array in-place, it does not return a value.

### Parameters

- **a** (`cupy.ndarray`) – The array, at least 2-D.
- **val** (`scalar`) – The value to be written on the diagonal. Its type must be compatible with that of the array `a`.
- **wrap** (`bool`) – If specified, the diagonal is “wrapped” after `N` columns. This affects only tall matrices.

## Examples

```
>>> a = cupy.zeros((3, 3), int)
>>> cupy.fill_diagonal(a, 5)
>>> a
array([[5, 0, 0],
       [0, 5, 0],
       [0, 0, 5]])
```

See also:

`numpy.fill_diagonal()`

## Iterating over arrays

---

`flatiter(a)`

Flat iterator object to iterate over arrays.

---

## `cupy.flatiter`

**class** `cupy.flatiter(a)`

Flat iterator object to iterate over arrays.

A flatiter iterator is returned by `x.flat` for any array `x`. It allows iterating over the array as if it were a 1-D array, either in a for-loop or by calling its `next` method.

Iteration is done in row-major, C-style order (the last index varying the fastest).

**Variables** `base` (`cupy.ndarray`) – A reference to the array that is iterated over.

---

**Note:** Restricted support of basic slicing is currently supplied. Advanced indexing is not supported yet.

---

See also:

`numpy.flatiter()`

## Methods

`__getitem__(ind)`

`__setitem__(ind, value)`

`__len__()`

`__next__()`

`__iter__()`

`copy()`

Get a copy of the iterator as a 1-D array.

`__eq__(value, /)`

Return `self==value`.

`__ne__(value, /)`

Return `self!=value`.

```

__lt__(value, /)
    Return self<value.

__le__(value, /)
    Return self<=value.

__gt__(value, /)
    Return self>value.

__ge__(value, /)
    Return self>=value.

```

### Attributes

**base**  
A reference to the array that is iterated over.

## 5.3.8 Input and output

---

**Hint:** [NumPy API Reference: Input and output](#)

---

### NumPy binary files (NPY, NPZ)

<code>load(file[, mmap_mode, allow_pickle])</code>	Loads arrays or pickled objects from <code>.npy</code> , <code>.npz</code> or pickled file.
<code>save(file, arr[, allow_pickle])</code>	Saves an array to a binary file in <code>.npy</code> format.
<code>savez(file, *args, **kwargs)</code>	Saves one or more arrays into a file in uncompressed <code>.npz</code> format.
<code>savez_compressed(file, *args, **kwargs)</code>	Saves one or more arrays into a file in compressed <code>.npz</code> format.

### cupy.load

`cupy.load(file, mmap_mode=None, allow_pickle=None)`  
Loads arrays or pickled objects from `.npy`, `.npz` or pickled file.

This function just calls `numpy.load` and then sends the arrays to the current device. NPZ file is converted to `NpzFile` object, which defers the transfer to the time of accessing the items.

#### Parameters

- **file** (*file-like object or string*) – The file to read.
- **mmap\_mode** (*None, 'r+', 'r', 'w+', 'c'*) – If not `None`, memory-map the file to construct an intermediate `numpy.ndarray` object and transfer it to the current device.
- **allow\_pickle** (*bool*) – Allow loading pickled object arrays stored in `npz` files. Reasons for disallowing pickles include security, as loading pickled data can execute arbitrary code. If pickles are disallowed, loading object arrays will fail. Please be aware that CuPy does not support arrays with dtype of *object*. The default is `False`. This option is available only for

NumPy 1.10 or later. In NumPy 1.9, this option cannot be specified (loading pickled objects is always allowed).

**Returns** CuPy array or NpzFile object depending on the type of the file. NpzFile object is a dictionary-like object with the context manager protocol (which enables us to use *with* statement on it).

**See also:**

`numpy.load()`

## cupy.save

`cupy.save(file, arr, allow_pickle=None)`

Saves an array to a binary file in `.npy` format.

### Parameters

- **file** (*file* or *str*) – File or filename to save.
- **arr** (*array\_like*) – Array to save. It should be able to feed to `cupy.asnumpy()`.
- **allow\_pickle** (*bool*) – Allow saving object arrays using Python pickles. Reasons for disallowing pickles include security (loading pickled data can execute arbitrary code) and portability (pickled objects may not be loadable on different Python installations, for example if the stored objects require libraries that are not available, and not all pickled data is compatible between Python 2 and Python 3). The default is True. This option is available only for NumPy 1.10 or later. In NumPy 1.9, this option cannot be specified (saving objects using pickles is always allowed).

**See also:**

`numpy.save()`

## cupy.savez

`cupy.savez(file, *args, **kwargs)`

Saves one or more arrays into a file in uncompressed `.npz` format.

Arguments without keys are treated as arguments with automatic keys named `arr_0`, `arr_1`, etc. corresponding to the positions in the argument list. The keys of arguments are used as keys in the `.npz` file, which are used for accessing NpzFile object when the file is read by `cupy.load()` function.

### Parameters

- **file** (*file* or *str*) – File or filename to save.
- **\*args** – Arrays with implicit keys.
- **\*\*kwargs** – Arrays with explicit keys.

**See also:**

`numpy.savez()`

## cupy.savez\_compressed

`cupy.savez_compressed(file, *args, **kws)`

Saves one or more arrays into a file in compressed .npz format.

It is equivalent to `cupy.savez()` function except the output file is compressed.

See also:

`cupy.savez()` for more detail, `numpy.savez_compressed()`

## String formatting

<code>array_repr(arr[, max_line_width, precision, ...])</code>	Returns the string representation of an array.
<code>array_str(arr[, max_line_width, precision, ...])</code>	Returns the string representation of the content of an array.

## cupy.array\_repr

`cupy.array_repr(arr, max_line_width=None, precision=None, suppress_small=None)`

Returns the string representation of an array.

### Parameters

- **arr** (*array\_like*) – Input array. It should be able to feed to `cupy.asnumpy()`.
- **max\_line\_width** (*int*) – The maximum number of line lengths.
- **precision** (*int*) – Floating point precision. It uses the current printing precision of NumPy.
- **suppress\_small** (*bool*) – If True, very small numbers are printed as zeros

**Returns** The string representation of arr.

**Return type** `str`

See also:

`numpy.array_repr()`

## cupy.array\_str

`cupy.array_str(arr, max_line_width=None, precision=None, suppress_small=None)`

Returns the string representation of the content of an array.

### Parameters

- **arr** (*array\_like*) – Input array. It should be able to feed to `cupy.asnumpy()`.
- **max\_line\_width** (*int*) – The maximum number of line lengths.
- **precision** (*int*) – Floating point precision. It uses the current printing precision of NumPy.
- **suppress\_small** (*bool*) – If True, very small number are printed as zeros.

See also:

`numpy.array_str()`

## Base-n representations

<code>binary_repr(num[, width])</code>	Return the binary representation of the input number as a string.
<code>base_repr(number[, base, padding])</code>	Return a string representation of a number in the given base system.

### cupy.base\_repr

`cupy.base_repr(number, base=2, padding=0)`

Return a string representation of a number in the given base system.

**See also:**

`numpy.base_repr()`

## 5.3.9 Linear algebra (cupy.linalg)

---

**Hint:** NumPy API Reference: Linear algebra ([numpy.linalg](#))

---

**See also:**

*Linear algebra (cupyx.scipy.linalg)*

### Matrix and vector products

<code>dot(a, b[, out])</code>	Returns a dot product of two arrays.
<code>vdot(a, b)</code>	Returns the dot product of two vectors.
<code>inner(a, b)</code>	Returns the inner product of two arrays.
<code>outer(a, b[, out])</code>	Returns the outer product of two vectors.
<code>matmul(x1, x2[, out, axes])</code>	Matrix product of two arrays.
<code>tensor_dot(a, b[, axes])</code>	Returns the tensor dot product of two arrays along specified axes.
<code>einsum(subscripts, *operands[, dtype])</code>	Evaluates the Einstein summation convention on the operands.
<code>linalg.matrix_power(M, n)</code>	Raise a square matrix to the (integer) power <i>n</i> .
<code>kron(a, b)</code>	Returns the kronecker product of two arrays.

### cupy.dot

`cupy.dot(a, b, out=None)`

Returns a dot product of two arrays.

For arrays with more than one axis, it computes the dot product along the last axis of *a* and the second-to-last axis of *b*. This is just a matrix product if the both arrays are 2-D. For 1-D arrays, it uses their unique axis as an axis to take dot product over.

#### Parameters



- **a** (`cupy.ndarray`) – The left argument.
- **b** (`cupy.ndarray`) – The right argument.
- **out** (`cupy.ndarray`) – Output array.

**Returns** The dot product of a and b.

**Return type** `cupy.ndarray`

**See also:**

`numpy.dot()`

## `cupy.vdot`

`cupy.vdot(a, b)`

Returns the dot product of two vectors.

The input arrays are flattened into 1-D vectors and then it performs inner product of these vectors.

**Parameters**

- **a** (`cupy.ndarray`) – The first argument.
- **b** (`cupy.ndarray`) – The second argument.

**Returns** Zero-dimensional array of the dot product result.

**Return type** `cupy.ndarray`

**See also:**

`numpy.vdot()`

## `cupy.inner`

`cupy.inner(a, b)`

Returns the inner product of two arrays.

It uses the last axis of each argument to take sum product.

**Parameters**

- **a** (`cupy.ndarray`) – The first argument.
- **b** (`cupy.ndarray`) – The second argument.

**Returns** The inner product of a and b.

**Return type** `cupy.ndarray`

**See also:**

`numpy.inner()`

## cupy.outer

`cupy.outer(a, b, out=None)`

Returns the outer product of two vectors.

The input arrays are flattened into 1-D vectors and then it performs outer product of these vectors.

### Parameters

- **a** (`cupy.ndarray`) – The first argument.
- **b** (`cupy.ndarray`) – The second argument.
- **out** (`cupy.ndarray`) – Output array.

**Returns** 2-D array of the outer product of a and b.

**Return type** `cupy.ndarray`

See also:

`numpy.outer()`

## cupy.tensordot

`cupy.tensordot(a, b, axes=2)`

Returns the tensor dot product of two arrays along specified axes.

This is equivalent to compute dot product along the specified axes which are treated as one axis by reshaping.

### Parameters

- **a** (`cupy.ndarray`) – The first argument.
- **b** (`cupy.ndarray`) – The second argument.
- **axes** –
  - If it is an integer, then axes axes at the last of a and the first of b are used.
  - If it is a pair of sequences of integers, then these two sequences specify the list of axes for a and b. The corresponding axes are paired for sum-product.

**Returns** The tensor dot product of a and b along the axes specified by axes.

**Return type** `cupy.ndarray`

See also:

`numpy.tensordot()`

## cupy.einsum

`cupy.einsum(subscripts, *operands, dtype=False)`

Evaluates the Einstein summation convention on the operands. Using the Einstein summation convention, many common multi-dimensional array operations can be represented in a simple fashion. This function provides a way to compute such summations.

---

**Note:** Memory contiguity of calculation result is not always compatible with `numpy.einsum`. `out`, `order`, and `casting` options are not supported.

---

**Parameters**

- **subscripts** (*str*) – Specifies the subscripts for summation.
- **operands** (*sequence of arrays*) – These are the arrays for the operation.

**Returns** The calculation based on the Einstein summation convention.

**Return type** *cupy.ndarray*

**See also:**

`numpy.einsum()`

**cupy.linalg.matrix\_power**

`cupy.linalg.matrix_power(M, n)`

Raise a square matrix to the (integer) power  $n$ .

**Parameters**

- **M** (*ndarray*) – Matrix to raise by power  $n$ .
- **n** (*~int*) – Power to raise matrix to.

**Returns** Output array.

**Return type** *ndarray*

..seealso:: `numpy.linalg.matrix_power()`

**cupy.kron**

`cupy.kron(a, b)`

Returns the kronecker product of two arrays.

**Parameters**

- **a** (*ndarray*) – The first argument.
- **b** (*ndarray*) – The second argument.

**Returns** Output array.

**Return type** *ndarray*

**See also:**

`numpy.kron()`

**Decompositions**

<code>linalg.cholesky(a)</code>	Cholesky decomposition.
<code>linalg.qr(a[, mode])</code>	QR decomposition.
<code>linalg.svd(a[, full_matrices, compute_uv])</code>	Singular Value Decomposition.

### cupy.linalg.cholesky

`cupy.linalg.cholesky(a)`  
Cholesky decomposition.

Decompose a given two-dimensional square matrix into  $L * L.T$ , where  $L$  is a lower-triangular matrix and  $.T$  is a conjugate transpose operator.

**Parameters** `a` (`cupy.ndarray`) – The input matrix with dimension  $(N, N)$

**Returns** The lower-triangular matrix.

**Return type** `cupy.ndarray`

**Warning:** This function calls one or more cuSOLVER routine(s) which may yield invalid results if input conditions are not met. To detect these invalid results, you can set the `linalg` configuration to a value that is not `ignore` in `cupyx.errstate()` or `cupyx.seterr()`.

See also:

`numpy.linalg.cholesky()`

### cupy.linalg.qr

`cupy.linalg.qr(a, mode='reduced')`  
QR decomposition.

Decompose a given two-dimensional matrix into  $Q * R$ , where  $Q$  is an orthonormal and  $R$  is an upper-triangular matrix.

**Parameters**

- `a` (`cupy.ndarray`) – The input matrix.
- `mode` (`str`) – The mode of decomposition. Currently ‘reduced’, ‘complete’, ‘r’, and ‘raw’ modes are supported. The default mode is ‘reduced’, in which matrix  $A = (M, N)$  is decomposed into  $Q, R$  with dimensions  $(M, K), (K, N)$ , where  $K = \min(M, N)$ .

**Returns** Although the type of returned object depends on the mode, it returns a tuple of  $(Q, R)$  by default. For details, please see the document of `numpy.linalg.qr()`.

**Return type** `cupy.ndarray`, or tuple of ndarray

**Warning:** This function calls one or more cuSOLVER routine(s) which may yield invalid results if input conditions are not met. To detect these invalid results, you can set the `linalg` configuration to a value that is not `ignore` in `cupyx.errstate()` or `cupyx.seterr()`.

See also:

`numpy.linalg.qr()`

## cupy.linalg.svd

`cupy.linalg.svd(a, full_matrices=True, compute_uv=True)`

Singular Value Decomposition.

Factorizes the matrix `a` as `u * np.diag(s) * v`, where `u` and `v` are unitary and `s` is an one-dimensional array of `a`'s singular values.

### Parameters

- **`a`** (`cupy.ndarray`) – The input matrix with dimension `(..., M, N)`.
- **`full_matrices`** (`bool`) – If `True`, it returns `u` and `v` with dimensions `(..., M, M)` and `(..., N, N)`. Otherwise, the dimensions of `u` and `v` are `(..., M, K)` and `(..., K, N)`, respectively, where `K = min(M, N)`.
- **`compute_uv`** (`bool`) – If `False`, it only returns singular values.

**Returns** A tuple of (`u`, `s`, `v`) such that `a = u * np.diag(s) * v`.

**Return type** tuple of `cupy.ndarray`

**Warning:** This function calls one or more cuSOLVER routine(s) which may yield invalid results if input conditions are not met. To detect these invalid results, you can set the `linalg` configuration to a value that is not `ignore` in `cupyx.errstate()` or `cupyx.seterr()`.

**Note:** On CUDA, when `a.ndim > 2` and the matrix dimensions `<= 32`, a fast code path based on Jacobian method (`gesvdj`) is taken. Otherwise, a QR method (`gesvd`) is used.

On ROCm, there is no such a fast code path that switches the underlying algorithm.

**See also:**

`numpy.linalg.svd()`

## Matrix eigenvalues

<code>linalg.eigh(a[, UPLO])</code>	Return the eigenvalues and eigenvectors of a complex Hermitian (conjugate symmetric) or a real symmetric matrix.
<code>linalg.eigvalsh(a[, UPLO])</code>	Compute the eigenvalues of a complex Hermitian or real symmetric matrix.

## cupy.linalg.eigh

`cupy.linalg.eigh(a, UPLO='L')`

Return the eigenvalues and eigenvectors of a complex Hermitian (conjugate symmetric) or a real symmetric matrix.

Returns two objects, a 1-D array containing the eigenvalues of *a*, and a 2-D square array or matrix (depending on the input type) of the corresponding eigenvectors (in columns).

### Parameters

- **a** (`cupy.ndarray`) – A symmetric 2-D square matrix (*M*, *M*) or a batch of symmetric 2-D square matrices (... , *M*, *M*).
- **UPLO** (`str`) – Select from 'L' or 'U'. It specifies which part of *a* is used. 'L' uses the lower triangular part of *a*, and 'U' uses the upper triangular part of *a*.

**Returns** Returns a tuple (*w*, *v*). *w* contains eigenvalues and *v* contains eigenvectors. *v*[:, *i*] is an eigenvector corresponding to an eigenvalue *w*[*i*]. For batch input, *v*[*k*, :, *i*] is an eigenvector corresponding to an eigenvalue *w*[*k*, *i*] of *a*[*k*].

**Return type** tuple of `ndarray`

**Warning:** This function calls one or more cuSOLVER routine(s) which may yield invalid results if input conditions are not met. To detect these invalid results, you can set the *linalg* configuration to a value that is not *ignore* in `cupyx.errstate()` or `cupyx.seterr()`.

See also:

`numpy.linalg.eigh()`

## cupy.linalg.eigvalsh

`cupy.linalg.eigvalsh(a, UPLO='L')`

Compute the eigenvalues of a complex Hermitian or real symmetric matrix.

Main difference from `eigh`: the eigenvectors are not computed.

### Parameters

- **a** (`cupy.ndarray`) – A symmetric 2-D square matrix (*M*, *M*) or a batch of symmetric 2-D square matrices (... , *M*, *M*).
- **UPLO** (`str`) – Select from 'L' or 'U'. It specifies which part of *a* is used. 'L' uses the lower triangular part of *a*, and 'U' uses the upper triangular part of *a*.

**Returns** Returns eigenvalues as a vector *w*. For batch input, *w*[*k*] is a vector of eigenvalues of matrix *a*[*k*].

**Return type** `cupy.ndarray`

**Warning:** This function calls one or more cuSOLVER routine(s) which may yield invalid results if input conditions are not met. To detect these invalid results, you can set the *linalg* configuration to a value that is not *ignore* in `cupyx.errstate()` or `cupyx.seterr()`.

See also:

`numpy.linalg.eigvalsh()`

## Norms and other numbers

<code>linalg.norm(x[, ord, axis, keepdims])</code>	Returns one of matrix norms specified by <code>ord</code> parameter.
<code>linalg.det(a)</code>	Returns the determinant of an array.
<code>linalg.matrix_rank(M[, tol])</code>	Return matrix rank of array using SVD method
<code>linalg.slogdet(a)</code>	Returns sign and logarithm of the determinant of an array.
<code>trace(a[, offset, axis1, axis2, dtype, out])</code>	Returns the sum along the diagonals of an array.

### `cupy.linalg.norm`

`cupy.linalg.norm(x, ord=None, axis=None, keepdims=False)`

Returns one of matrix norms specified by `ord` parameter.

See `numpy.linalg.norm` for more detail.

#### Parameters

- **x** (`cupy.ndarray`) – Array to take norm. If `axis` is `None`, `x` must be 1-D or 2-D.
- **ord** (*non-zero int, inf, -inf, 'fro'*) – Norm type.
- **axis** (*int, 2-tuple of ints, None*) – 1-D or 2-D norm is computed over `axis`.
- **keepdims** (*bool*) – If this is set `True`, the axes which are normed over are left.

**Returns** `cupy.ndarray`

### `cupy.linalg.det`

`cupy.linalg.det(a)`

Returns the determinant of an array.

**Parameters** **a** (`cupy.ndarray`) – The input matrix with dimension `(..., N, N)`.

**Returns** Determinant of `a`. Its shape is `a.shape[:-2]`.

**Return type** `cupy.ndarray`

See also:

`numpy.linalg.det()`

### cupy.linalg.matrix\_rank

cupy.linalg.matrix\_rank(*M*, *tol=None*)

Return matrix rank of array using SVD method

#### Parameters

- **M** (`cupy.ndarray`) – Input array. Its *ndim* must be less than or equal to 2.
- **tol** (`None` or `float`) – Threshold of singular value of *M*. When *tol* is `None`, and *eps* is the epsilon value for datatype of *M*, then *tol* is set to  $S_{\max}() * \max(M.shape) * eps$ , where *S* is the singular value of *M*. It obeys `numpy.linalg.matrix_rank()`.

**Returns** Rank of *M*.

**Return type** `cupy.ndarray`

See also:

`numpy.linalg.matrix_rank()`

### cupy.linalg.slogdet

cupy.linalg.slogdet(*a*)

Returns sign and logarithm of the determinant of an array.

It calculates the natural logarithm of the determinant of a given value.

**Parameters** **a** (`cupy.ndarray`) – The input matrix with dimension  $(\dots, N, N)$ .

**Returns** It returns a tuple (*sign*, *logdet*). *sign* represents each sign of the determinant as a real number 0, 1 or -1. 'logdet' represents the natural logarithm of the absolute of the determinant. If the determinant is zero, *sign* will be 0 and *logdet* will be -inf. The shapes of both *sign* and *logdet* are equal to *a.shape[: -2]*.

**Return type** tuple of `ndarray`

**Warning:** This function calls one or more cuSOLVER routine(s) which may yield invalid results if input conditions are not met. To detect these invalid results, you can set the *linalg* configuration to a value that is not *ignore* in `cupyx.errstate()` or `cupyx.seterr()`.

**Warning:** To produce the same results as `numpy.linalg.slogdet()` for singular inputs, set the *linalg* configuration to *raise*.

See also:

`numpy.linalg.slogdet()`



## cupy.trace

`cupy.trace(a, offset=0, axis1=0, axis2=1, dtype=None, out=None)`

Returns the sum along the diagonals of an array.

It computes the sum along the diagonals at `axis1` and `axis2`.

### Parameters

- **a** (`cupy.ndarray`) – Array to take trace.
- **offset** (`int`) – Index of diagonals. Zero indicates the main diagonal, a positive value an upper diagonal, and a negative value a lower diagonal.
- **axis1** (`int`) – The first axis along which the trace is taken.
- **axis2** (`int`) – The second axis along which the trace is taken.
- **dtype** – Data type specifier of the output.
- **out** (`cupy.ndarray`) – Output array.

**Returns** The trace of `a` along axes (`axis1`, `axis2`).

**Return type** `cupy.ndarray`

See also:

`numpy.trace()`

## Solving equations and inverting matrices

<code>linalg.solve(a, b)</code>	Solves a linear matrix equation.
<code>linalg.tensorsolve(a, b[, axes])</code>	Solves tensor equations denoted by $\mathbf{ax} = \mathbf{b}$ .
<code>linalg.lstsq(a, b[, rcond])</code>	Return the least-squares solution to a linear matrix equation.
<code>linalg.inv(a)</code>	Computes the inverse of a matrix.
<code>linalg.pinv(a[, rcond])</code>	Compute the Moore-Penrose pseudoinverse of a matrix.
<code>linalg.tensorinv(a[, ind])</code>	Computes the inverse of a tensor.

## cupy.linalg.solve

`cupy.linalg.solve(a, b)`

Solves a linear matrix equation.

It computes the exact solution of  $\mathbf{x}$  in  $\mathbf{ax} = \mathbf{b}$ , where `a` is a square and full rank matrix.

### Parameters

- **a** (`cupy.ndarray`) – The matrix with dimension  $(\dots, \mathbf{M}, \mathbf{M})$ .
- **b** (`cupy.ndarray`) – The matrix with dimension  $(\dots, \mathbf{M})$  or  $(\dots, \mathbf{M}, \mathbf{K})$ .

**Returns** The matrix with dimension  $(\dots, \mathbf{M})$  or  $(\dots, \mathbf{M}, \mathbf{K})$ .

**Return type** `cupy.ndarray`

**Warning:** This function calls one or more cuSOLVER routine(s) which may yield invalid results if input conditions are not met. To detect these invalid results, you can set the *linalg* configuration to a value that is not *ignore* in `cupyx.errstate()` or `cupyx.seterr()`.

See also:

`numpy.linalg.solve()`

## `cupy.linalg.tensorsolve`

`cupy.linalg.tensorsolve(a, b, axes=None)`

Solves tensor equations denoted by  $\mathbf{ax} = \mathbf{b}$ .

Suppose that  $\mathbf{b}$  is equivalent to `cupy.tensordot(a, x)`. This function computes tensor  $\mathbf{x}$  from  $\mathbf{a}$  and  $\mathbf{b}$ .

### Parameters

- **a** (`cupy.ndarray`) – The tensor with `len(shape) >= 1`
- **b** (`cupy.ndarray`) – The tensor with `len(shape) >= 1`
- **axes** (*tuple of ints*) – Axes in  $\mathbf{a}$  to reorder to the right before inversion.

**Returns** The tensor with shape  $\mathbf{Q}$  such that  $\mathbf{b.shape} + \mathbf{Q} == \mathbf{a.shape}$ .

**Return type** `cupy.ndarray`

**Warning:** This function calls one or more cuSOLVER routine(s) which may yield invalid results if input conditions are not met. To detect these invalid results, you can set the *linalg* configuration to a value that is not *ignore* in `cupyx.errstate()` or `cupyx.seterr()`.

See also:

`numpy.linalg.tensorsolve()`

## `cupy.linalg.lstsq`

`cupy.linalg.lstsq(a, b, rcond='warn')`

Return the least-squares solution to a linear matrix equation.

Solves the equation  $\mathbf{a} \mathbf{x} = \mathbf{b}$  by computing a vector  $\mathbf{x}$  that minimizes the Euclidean 2-norm  $\|\mathbf{b} - \mathbf{a} \mathbf{x}\|^2$ . The equation may be under-, well-, or over- determined (i.e., the number of linearly independent rows of  $\mathbf{a}$  can be less than, equal to, or greater than its number of linearly independent columns). If  $\mathbf{a}$  is square and of full rank, then  $\mathbf{x}$  (but for round-off error) is the “exact” solution of the equation.

### Parameters

- **a** (`cupy.ndarray`) – “Coefficient” matrix with dimension  $(\mathbf{M}, \mathbf{N})$
- **b** (`cupy.ndarray`) – “Dependent variable” values with dimension  $(\mathbf{M},)$  or  $(\mathbf{M}, \mathbf{K})$
- **rcond** (*float*) – Cutoff parameter for small singular values. For stability it computes the largest singular value denoted by  $\mathbf{s}$ , and sets all singular values smaller than  $\mathbf{s}$  to zero.

**Returns** A tuple of  $(\mathbf{x}, \mathbf{residuals}, \mathbf{rank}, \mathbf{s})$ . Note  $\mathbf{x}$  is the least-squares solution with shape  $(\mathbf{N},)$  or  $(\mathbf{N}, \mathbf{K})$  depending if  $\mathbf{b}$  was two-dimensional. The sums of `residuals` is the squared Euclidean 2-norm for each column in  $\mathbf{b} - \mathbf{a} \mathbf{x}$ . The `residuals` is an empty array if the rank of

$a$  is  $< N$  or  $M \leq N$ , but iff  $b$  is 1-dimensional, this is a (1,) shape array, Otherwise the shape is (K,). The rank of matrix  $a$  is an integer. The singular values of  $a$  are  $s$ .

**Return type** `tuple`

**Warning:** This function calls one or more cuSOLVER routine(s) which may yield invalid results if input conditions are not met. To detect these invalid results, you can set the `linalg` configuration to a value that is not `ignore` in `cupyx.errstate()` or `cupyx.seterr()`.

**See also:**

`numpy.linalg.lstsq()`

### `cupy.linalg.inv`

`cupy.linalg.inv(a)`

Computes the inverse of a matrix.

This function computes matrix `a_inv` from  $n$ -dimensional regular matrix `a` such that `dot(a, a_inv) == eye(n)`.

**Parameters** `a` (`cupy.ndarray`) – The regular matrix

**Returns** The inverse of a matrix.

**Return type** `cupy.ndarray`

**Warning:** This function calls one or more cuSOLVER routine(s) which may yield invalid results if input conditions are not met. To detect these invalid results, you can set the `linalg` configuration to a value that is not `ignore` in `cupyx.errstate()` or `cupyx.seterr()`.

**See also:**

`numpy.linalg.inv()`

### `cupy.linalg.pinv`

`cupy.linalg.pinv(a, rcond=1e-15)`

Compute the Moore-Penrose pseudoinverse of a matrix.

It computes a pseudoinverse of a matrix `a`, which is a generalization of the inverse matrix with Singular Value Decomposition (SVD). Note that it automatically removes small singular values for stability.

**Parameters**

- `a` (`cupy.ndarray`) – The matrix with dimension  $(\dots, M, N)$
- `rcond` (`float` or `cupy.ndarray`) – Cutoff parameter for small singular values. For stability it computes the largest singular value denoted by  $s$ , and sets all singular values smaller than `rcond * s` to zero. Broadcasts against the stack of matrices.

**Returns** The pseudoinverse of `a` with dimension  $(\dots, N, M)$ .

**Return type** `cupy.ndarray`

**Warning:** This function calls one or more cuSOLVER routine(s) which may yield invalid results if input conditions are not met. To detect these invalid results, you can set the *linalg* configuration to a value that is not *ignore* in `cupyx.errstate()` or `cupyx.seterr()`.

See also:

`numpy.linalg.pinv()`

### `cupy.linalg.tensorinv`

`cupy.linalg.tensorinv(a, ind=2)`

Computes the inverse of a tensor.

This function computes tensor `a_inv` from tensor `a` such that `tensordot(a_inv, a, ind) == I`, where `I` denotes the identity tensor.

#### Parameters

- **a** (`cupy.ndarray`) – The tensor such that `prod(a.shape[:ind]) == prod(a.shape[ind:])`.
- **ind** (`int`) – The positive number used in `axes` option of `tensordot`.

**Returns** The inverse of a tensor whose shape is equivalent to `a.shape[ind:] + a.shape[:ind]`.

**Return type** `cupy.ndarray`

**Warning:** This function calls one or more cuSOLVER routine(s) which may yield invalid results if input conditions are not met. To detect these invalid results, you can set the *linalg* configuration to a value that is not *ignore* in `cupyx.errstate()` or `cupyx.seterr()`.

See also:

`numpy.linalg.tensorinv()`

## 5.3.10 Logic functions

---

**Hint:** NumPy API Reference: Logic functions

---

### Truth value testing

<code>all(a[, axis, out, keepdims])</code>	Tests whether all array elements along a given axis evaluate to True.
<code>any(a[, axis, out, keepdims])</code>	Tests whether any array elements along a given axis evaluate to True.

## cupy.all

`cupy.all(a, axis=None, out=None, keepdims=False)`

Tests whether all array elements along a given axis evaluate to True.

### Parameters

- **a** (`cupy.ndarray`) – Input array.
- **axis** (`int` or `tuple of ints`) – Along which axis to compute all. The flattened array is used by default.
- **out** (`cupy.ndarray`) – Output array.
- **keepdims** (`bool`) – If True, the axis is remained as an axis of size one.

**Returns** An array reduced of the input array along the axis.

**Return type** `cupy.ndarray`

See also:

`numpy.all()`

## cupy.any

`cupy.any(a, axis=None, out=None, keepdims=False)`

Tests whether any array elements along a given axis evaluate to True.

### Parameters

- **a** (`cupy.ndarray`) – Input array.
- **axis** (`int` or `tuple of ints`) – Along which axis to compute all. The flattened array is used by default.
- **out** (`cupy.ndarray`) – Output array.
- **keepdims** (`bool`) – If True, the axis is remained as an axis of size one.

**Returns** An array reduced of the input array along the axis.

**Return type** `cupy.ndarray`

See also:

`numpy.any()`

## Array contents

<code>isfinite</code>	Tests finiteness elementwise.
<code>isinf</code>	Tests if each element is the positive or negative infinity.
<code>isnan</code>	Tests if each element is a NaN.

## Array type testing

<code>iscomplex(x)</code>	Returns a bool array, where True if input element is complex.
<code>iscomplexobj(x)</code>	Check for a complex type or an array of complex numbers.
<code>isfortran(a)</code>	Returns True if the array is Fortran contiguous but <i>not</i> C contiguous.
<code>isreal(x)</code>	Returns a bool array, where True if input element is real.
<code>isrealobj(x)</code>	Return True if x is a not complex type or an array of complex numbers.
<code>isscalar(element)</code>	Returns True if the type of num is a scalar type.

## cupy.iscomplex

`cupy.iscomplex(x)`

Returns a bool array, where True if input element is complex.

What is tested is whether the input has a non-zero imaginary part, not if the input type is complex.

**Parameters** `x` (`cupy.ndarray`) – Input array.

**Returns** Boolean array of the same shape as `x`.

**Return type** `cupy.ndarray`

See also:

`isreal()`, `iscomplexobj()`

## Examples

```
>>> cupy.iscomplex(cupy.array([1+1j, 1+0j, 4.5, 3, 2, 2j]))
array([ True, False, False, False, False,  True])
```

## cupy.iscomplexobj

`cupy.iscomplexobj(x)`

Check for a complex type or an array of complex numbers.

The type of the input is checked, not the value. Even if the input has an imaginary part equal to zero, `iscomplexobj` evaluates to True.

**Parameters** `x` (`cupy.ndarray`) – Input array.

**Returns** The return value, True if `x` is of a complex type or has at least one complex element.

**Return type** `bool`

See also:

`isrealobj()`, `iscomplex()`

## Examples

```
>>> cupy.iscomplexobj(cupy.array([3, 1+0j, True]))
True
>>> cupy.iscomplexobj(cupy.array([3, 1, True]))
False
```

## cupy.isfortran

### cupy.isfortran(a)

Returns True if the array is Fortran contiguous but *not* C contiguous.

If you only want to check if an array is Fortran contiguous use `a.flags.f_contiguous` instead.

**Parameters** `a` (`cupy.ndarray`) – Input array.

**Returns** The return value, True if `a` is Fortran contiguous but not C contiguous.

**Return type** `bool`

See also:

`isfortran()`

## Examples

`cupy.array` allows to specify whether the array is written in C-contiguous order (last index varies the fastest), or FORTRAN-contiguous order in memory (first index varies the fastest).

```
>>> a = cupy.array([[1, 2, 3], [4, 5, 6]], order='C')
>>> a
array([[1, 2, 3],
       [4, 5, 6]])
>>> cupy.isfortran(a)
False
```

```
>>> b = cupy.array([[1, 2, 3], [4, 5, 6]], order='F')
>>> b
array([[1, 2, 3],
       [4, 5, 6]])
>>> cupy.isfortran(b)
True
```

The transpose of a C-ordered array is a FORTRAN-ordered array.

```
>>> a = cupy.array([[1, 2, 3], [4, 5, 6]], order='C')
>>> a
array([[1, 2, 3],
       [4, 5, 6]])
>>> cupy.isfortran(a)
False
>>> b = a.T
>>> b
array([[1, 4],
```

(continues on next page)

(continued from previous page)

```
[2, 5],  
[3, 6]])  
>>> cupy.isfortran(b)  
True
```

C-ordered arrays evaluate as False even if they are also FORTRAN-ordered.

```
>>> cupy.isfortran(np.array([1, 2], order='F'))  
False
```

## cupy.isreal

### cupy.isreal(*x*)

Returns a bool array, where True if input element is real.

If element has complex type with zero complex part, the return value for that element is True.

**Parameters** *x* ([cupy.ndarray](#)) – Input array.

**Returns** Boolean array of same shape as *x*.

**Return type** [cupy.ndarray](#)

**See also:**

[iscomplex\(\)](#), [isrealobj\(\)](#)

### Examples

```
>>> cupy.isreal(cp.array([1+1j, 1+0j, 4.5, 3, 2, 2j]))  
array([False,  True,  True,  True,  True, False])
```

## cupy.isrealobj

### cupy.isrealobj(*x*)

Return True if *x* is a not complex type or an array of complex numbers.

The type of the input is checked, not the value. So even if the input has an imaginary part equal to zero, *isrealobj* evaluates to False if the data type is complex.

**Parameters** *x* ([cupy.ndarray](#)) – The input can be of any type and shape.

**Returns** The return value, False if *x* is of a complex type.

**Return type** [bool](#)

**See also:**

[iscomplexobj\(\)](#), [isreal\(\)](#)



## Examples

```
>>> cupy.isrealobj(cupy.array([3, 1+0j, True]))
False
>>> cupy.isrealobj(cupy.array([3, 1, True]))
True
```

## cupy.isscalar

`cupy.isscalar(element)`

Returns True if the type of num is a scalar type.

**See also:**

`numpy.isscalar()`

## Logic operations

<code>logical_and</code>	Computes the logical AND of two arrays.
<code>logical_or</code>	Computes the logical OR of two arrays.
<code>logical_not</code>	Computes the logical NOT of an array.
<code>logical_xor</code>	Computes the logical XOR of two arrays.

## Comparison

<code>allclose(a, b[, rtol, atol, equal_nan])</code>	Returns True if two arrays are element-wise equal within a tolerance.
<code>isclose(a, b[, rtol, atol, equal_nan])</code>	Returns a boolean array where two arrays are equal within a tolerance.
<code>array_equal(a1, a2[, equal_nan])</code>	Returns True if two arrays are element-wise exactly equal.
<code>greater</code>	Tests elementwise if $x1 > x2$ .
<code>greater_equal</code>	Tests elementwise if $x1 \geq x2$ .
<code>less</code>	Tests elementwise if $x1 < x2$ .
<code>less_equal</code>	Tests elementwise if $x1 \leq x2$ .
<code>equal</code>	Tests elementwise if $x1 == x2$ .
<code>not_equal</code>	Tests elementwise if $x1 \neq x2$ .

## cupy.allclose

`cupy.allclose(a, b, rtol=1e-05, atol=1e-08, equal_nan=False)`

Returns True if two arrays are element-wise equal within a tolerance.

Two values in *a* and *b* are considered equal when the following equation is satisfied.

$$|a - b| \leq atol + rtol|b|$$

### Parameters

- a* (`cupy.ndarray`) – Input array to compare.

- **b** (`cupy.ndarray`) – Input array to compare.
- **rtol** (`float`) – The relative tolerance.
- **atol** (`float`) – The absolute tolerance.
- **equal\_nan** (`bool`) – If True, NaN's in a will be considered equal to NaN's in b.

**Returns** A boolean 0-dim array. If its value is True, two arrays are element-wise equal within a tolerance.

**Return type** `cupy.ndarray`

**See also:**

`numpy.allclose()`

### `cupy.isclose`

`cupy.isclose(a, b, rtol=1e-05, atol=1e-08, equal_nan=False)`

Returns a boolean array where two arrays are equal within a tolerance.

Two values in a and b are considered equal when the following equation is satisfied.

$$|a - b| \leq \text{atol} + \text{rtol}|b|$$

#### Parameters

- **a** (`cupy.ndarray`) – Input array to compare.
- **b** (`cupy.ndarray`) – Input array to compare.
- **rtol** (`float`) – The relative tolerance.
- **atol** (`float`) – The absolute tolerance.
- **equal\_nan** (`bool`) – If True, NaN's in a will be considered equal to NaN's in b.

**Returns** A boolean array storing where a and b are equal.

**Return type** `cupy.ndarray`

**See also:**

`numpy.isclose()`

### `cupy.array_equal`

`cupy.array_equal(a1, a2, equal_nan=False)`

Returns True if two arrays are element-wise exactly equal.

#### Parameters

- **a1** (`cupy.ndarray`) – Input array to compare.
- **a2** (`cupy.ndarray`) – Input array to compare.
- **equal\_nan** (`bool`) – If True, NaN's in a1 will be considered equal to NaN's in a2.

**Returns** A boolean 0-dim array. If its value is True, two arrays are element-wise equal.

**Return type** `cupy.ndarray`

See also:

`numpy.array_equal()`

### 5.3.11 Mathematical functions

---

**Hint:** NumPy API Reference: Mathematical functions

---

#### Trigonometric functions

<i><code>sin</code></i>	Elementwise sine function.
<i><code>cos</code></i>	Elementwise cosine function.
<i><code>tan</code></i>	Elementwise tangent function.
<i><code>arcsin</code></i>	Elementwise inverse-sine function (a.k.a.
<i><code>arccos</code></i>	Elementwise inverse-cosine function (a.k.a.
<i><code>arctan</code></i>	Elementwise inverse-tangent function (a.k.a.
<i><code>hypot</code></i>	Computes the hypoteneous of orthogonal vectors of given length.
<i><code>arctan2</code></i>	Elementwise inverse-tangent of the ratio of two arrays.
<i><code>degrees</code></i>	Converts angles from radians to degrees elementwise.
<i><code>radians</code></i>	Converts angles from degrees to radians elementwise.
<i><code>unwrap(p[, discount, axis])</code></i>	Unwrap by changing deltas between values to 2*pi complement.
<i><code>deg2rad</code></i>	Converts angles from degrees to radians elementwise.
<i><code>rad2deg</code></i>	Converts angles from radians to degrees elementwise.

#### `cupy.unwrap`

`cupy.unwrap(p, discount=3.141592653589793, axis=-1)`

Unwrap by changing deltas between values to 2\*pi complement.

##### Parameters

- **p** (`cupy.ndarray`) – Input array.
- **discount** (`float`) – Maximum discontinuity between values, default is `pi`.
- **axis** (`int`) – Axis along which unwrap will operate, default is the last axis.

**Returns** The result array.

**Return type** `cupy.ndarray`

See also:

`numpy.unwrap()`

## Hyperbolic functions

<i><a href="#">sinh</a></i>	Elementwise hyperbolic sine function.
<i><a href="#">cosh</a></i>	Elementwise hyperbolic cosine function.
<i><a href="#">tanh</a></i>	Elementwise hyperbolic tangent function.
<i><a href="#">arcsinh</a></i>	Elementwise inverse of hyperbolic sine function.
<i><a href="#">arccosh</a></i>	Elementwise inverse of hyperbolic cosine function.
<i><a href="#">arctanh</a></i>	Elementwise inverse of hyperbolic tangent function.

## Rounding

<i><a href="#">around</a></i> (a[, decimals, out])	Rounds to the given number of decimals.
<i><a href="#">round_</a></i> (a[, decimals, out])	
<i><a href="#">rint</a></i>	Rounds each element of an array to the nearest integer.
<i><a href="#">fix</a></i>	If given value x is positive, it return floor(x).
<i><a href="#">floor</a></i>	Rounds each element of an array to its floor integer.
<i><a href="#">ceil</a></i>	Rounds each element of an array to its ceiling integer.
<i><a href="#">trunc</a></i>	Rounds each element of an array towards zero.

### `cupy.around`

`cupy.around(a, decimals=0, out=None)`

Rounds to the given number of decimals.

#### Parameters

- **a** (`cupy.ndarray`) – The source array.
- **decimals** (`int`) – Number of decimal places to round to (default: 0). If decimals is negative, it specifies the number of positions to the left of the decimal point.
- **out** (`cupy.ndarray`) – Output array.

**Returns** Rounded array.

**Return type** `cupy.ndarray`

**See also:**

`numpy.around()`

### `cupy.round_`

`cupy.round_(a, decimals=0, out=None)`

## cupy.fix

`cupy.fix = <ufunc 'cupy_fix'>`

If given value *x* is positive, it return `floor(x)`. Else, it return `ceil(x)`.

**See also:**

`numpy.fix()`

## Sums, products, differences

<code>prod(a[, axis, dtype, out, keepdims])</code>	Returns the product of an array along given axes.
<code>sum(a[, axis, dtype, out, keepdims])</code>	Returns the sum of an array along given axes.
<code>nanprod(a[, axis, dtype, out, keepdims])</code>	Returns the product of an array along given axes treating Not a Numbers (NaNs) as zero.
<code>nansum(a[, axis, dtype, out, keepdims])</code>	Returns the sum of an array along given axes treating Not a Numbers (NaNs) as zero.
<code>cumprod(a[, axis, dtype, out])</code>	Returns the cumulative product of an array along a given axis.
<code>cumsum(a[, axis, dtype, out])</code>	Returns the cumulative sum of an array along a given axis.
<code>nancumprod(a[, axis, dtype, out])</code>	Returns the cumulative product of an array along a given axis treating Not a Numbers (NaNs) as one.
<code>nancumsum(a[, axis, dtype, out])</code>	Returns the cumulative sum of an array along a given axis treating Not a Numbers (NaNs) as zero.
<code>diff(a[, n, axis, prepend, append])</code>	Calculate the <i>n</i> -th discrete difference along the given axis.
<code>gradient(f, *varargs[, axis, edge_order])</code>	Return the gradient of an <i>N</i> -dimensional array.
<code>cross(a, b[, axisa, axisb, axisc, axis])</code>	Returns the cross product of two vectors.

## cupy.prod

`cupy.prod(a, axis=None, dtype=None, out=None, keepdims=False)`

Returns the product of an array along given axes.

### Parameters

- **a** (`cupy.ndarray`) – Array to take product.
- **axis** (*int or sequence of ints*) – Axes along which the product is taken.
- **dtype** – Data type specifier.
- **out** (`cupy.ndarray`) – Output array.
- **keepdims** (*bool*) – If True, the specified axes are remained as axes of length one.

**Returns** The result array.

**Return type** `cupy.ndarray`

**See also:**

`numpy.prod()`

## cupy.sum

`cupy.sum(a, axis=None, dtype=None, out=None, keepdims=False)`

Returns the sum of an array along given axes.

### Parameters

- **a** (`cupy.ndarray`) – Array to take sum.
- **axis** (*int or sequence of ints*) – Axes along which the sum is taken.
- **dtype** – Data type specifier.
- **out** (`cupy.ndarray`) – Output array.
- **keepdims** (*bool*) – If True, the specified axes are remained as axes of length one.

**Returns** The result array.

**Return type** `cupy.ndarray`

**See also:**

`numpy.sum()`

## cupy.nanprod

`cupy.nanprod(a, axis=None, dtype=None, out=None, keepdims=False)`

Returns the product of an array along given axes treating Not a Numbers (NaNs) as zero.

### Parameters

- **a** (`cupy.ndarray`) – Array to take product.
- **axis** (*int or sequence of ints*) – Axes along which the product is taken.
- **dtype** – Data type specifier.
- **out** (`cupy.ndarray`) – Output array.
- **keepdims** (*bool*) – If True, the specified axes are remained as axes of length one.

**Returns** The result array.

**Return type** `cupy.ndarray`

**See also:**

`numpy.nanprod()`

## cupy.nansum

`cupy.nansum(a, axis=None, dtype=None, out=None, keepdims=False)`

Returns the sum of an array along given axes treating Not a Numbers (NaNs) as zero.

### Parameters

- **a** (`cupy.ndarray`) – Array to take sum.
- **axis** (*int or sequence of ints*) – Axes along which the sum is taken.
- **dtype** – Data type specifier.
- **out** (`cupy.ndarray`) – Output array.

- **keepdims** (*bool*) – If True, the specified axes are remained as axes of length one.

**Returns** The result array.

**Return type** *cupy.ndarray*

See also:

*numpy.nansum()*

## cupy.cumprod

`cupy.cumprod(a, axis=None, dtype=None, out=None)`

Returns the cumulative product of an array along a given axis.

### Parameters

- **a** (*cupy.ndarray*) – Input array.
- **axis** (*int*) – Axis along which the cumulative product is taken. If it is not specified, the input is flattened.
- **dtype** – Data type specifier.
- **out** (*cupy.ndarray*) – Output array.

**Returns** The result array.

**Return type** *cupy.ndarray*

See also:

*numpy.cumprod()*

## cupy.cumsum

`cupy.cumsum(a, axis=None, dtype=None, out=None)`

Returns the cumulative sum of an array along a given axis.

### Parameters

- **a** (*cupy.ndarray*) – Input array.
- **axis** (*int*) – Axis along which the cumulative sum is taken. If it is not specified, the input is flattened.
- **dtype** – Data type specifier.
- **out** (*cupy.ndarray*) – Output array.

**Returns** The result array.

**Return type** *cupy.ndarray*

See also:

*numpy.cumsum()*

## cupy.nancumprod

`cupy.nancumprod(a, axis=None, dtype=None, out=None)`

Returns the cumulative product of an array along a given axis treating Not a Numbers (NaNs) as one.

### Parameters

- **a** (`cupy.ndarray`) – Input array.
- **axis** (`int`) – Axis along which the cumulative product is taken. If it is not specified, the input is flattened.
- **dtype** – Data type specifier.
- **out** (`cupy.ndarray`) – Output array.

**Returns** The result array.

**Return type** `cupy.ndarray`

See also:

`numpy.nancumprod()`

## cupy.nancumsum

`cupy.nancumsum(a, axis=None, dtype=None, out=None)`

Returns the cumulative sum of an array along a given axis treating Not a Numbers (NaNs) as zero.

### Parameters

- **a** (`cupy.ndarray`) – Input array.
- **axis** (`int`) – Axis along which the cumulative sum is taken. If it is not specified, the input is flattened.
- **dtype** – Data type specifier.
- **out** (`cupy.ndarray`) – Output array.

**Returns** The result array.

**Return type** `cupy.ndarray`

See also:

`numpy.nancumsum()`

## cupy.diff

`cupy.diff(a, n=1, axis=-1, prepend=None, append=None)`

Calculate the n-th discrete difference along the given axis.

### Parameters

- **a** (`cupy.ndarray`) – Input array.
- **n** (`int`) – The number of times values are differenced. If zero, the input is returned as-is.
- **axis** (`int`) – The axis along which the difference is taken, default is the last axis.
- **prepend** (`int`, `float`, `cupy.ndarray`) – Value to prepend to a.
- **append** (`int`, `float`, `cupy.ndarray`) – Value to append to a.



**Returns** The result array.

**Return type** `cupy.ndarray`

**See also:**

`numpy.diff()`

## `cupy.gradient`

`cupy.gradient(f, *varargs, axis=None, edge_order=1)`

Return the gradient of an N-dimensional array.

The gradient is computed using second order accurate central differences in the interior points and either first or second order accurate one-sides (forward or backwards) differences at the boundaries. The returned gradient hence has the same shape as the input array.

### Parameters

- **f** (`cupy.ndarray`) – An N-dimensional array containing samples of a scalar function.
- **varargs** (*list of scalar or array, optional*) – Spacing between f values. Default unitary spacing for all dimensions. Spacing can be specified using:
  1. single scalar to specify a sample distance for all dimensions.
  2. N scalars to specify a constant sample distance for each dimension. i.e.  $dx, dy, dz, \dots$
  3. N arrays to specify the coordinates of the values along each dimension of F. The length of the array must match the size of the corresponding dimension
  4. Any combination of N scalars/arrays with the meaning of 2. and 3.
 If *axis* is given, the number of varargs must equal the number of axes. Default: 1.
- **edge\_order** (`{1, 2}`, *optional*) – The gradient is calculated using N-th order accurate differences at the boundaries. Default: 1.
- **axis** (`None` or `int` or *tuple of ints, optional*) – The gradient is calculated only along the given axis or axes. The default (*axis = None*) is to calculate the gradient for all the axes of the input array. *axis* may be negative, in which case it counts from the last to the first axis.

**Returns** A set of ndarrays (or a single ndarray if there is only one dimension) corresponding to the derivatives of f with respect to each dimension. Each derivative has the same shape as f.

**Return type** gradient (`cupy.ndarray` or list of `cupy.ndarray`)

**See also:**

`numpy.gradient()`

## cupy.cross

`cupy.cross(a, b, axisa=-1, axisb=-1, axisc=-1, axis=None)`

Returns the cross product of two vectors.

The cross product of **a** and **b** in  $R^3$  is a vector perpendicular to both **a** and **b**. If **a** and **b** are arrays of vectors, the vectors are defined by the last axis of **a** and **b** by default, and these axes can have dimensions 2 or 3. Where the dimension of either **a** or **b** is 2, the third component of the input vector is assumed to be zero and the cross product calculated accordingly. In cases where both input vectors have dimension 2, the z-component of the cross product is returned.

### Parameters

- **a** (`cupy.ndarray`) – Components of the first vector(s).
- **b** (`cupy.ndarray`) – Components of the second vector(s).
- **axisa** (`int`, *optional*) – Axis of **a** that defines the vector(s). By default, the last axis.
- **axisb** (`int`, *optional*) – Axis of **b** that defines the vector(s). By default, the last axis.
- **axisc** (`int`, *optional*) – Axis of **c** containing the cross product vector(s). Ignored if both input vectors have dimension 2, as the return is scalar. By default, the last axis.
- **axis** (`int`, *optional*) – If defined, the axis of **a**, **b** and **c** that defines the vector(s) and cross product(s). Overrides **axisa**, **axisb** and **axisc**.

**Returns** Vector cross product(s).

**Return type** `cupy.ndarray`

**See also:**

`numpy.cross()`

## Exponents and logarithms

<code>exp</code>	Elementwise exponential function.
<code>expm1</code>	Computes $\exp(x) - 1$ elementwise.
<code>exp2</code>	Elementwise exponentiation with base 2.
<code>log</code>	Elementwise natural logarithm function.
<code>log10</code>	Elementwise common logarithm function.
<code>log2</code>	Elementwise binary logarithm function.
<code>log1p</code>	Computes $\log(1 + x)$ elementwise.
<code>logaddexp</code>	Computes $\log(\exp(x1) + \exp(x2))$ elementwise.
<code>logaddexp2</code>	Computes $\log_2(\exp_2(x1) + \exp_2(x2))$ elementwise.

## Other special functions

<i>i0</i>	Modified Bessel function of the first kind, order 0.
<i>sinc</i>	Elementwise sinc function.

### cupy.i0

`cupy.i0 = <ufunc 'cupy_i0'>`

Modified Bessel function of the first kind, order 0.

See also:

`numpy.i0()`

### cupy.sinc

`cupy.sinc = <ufunc 'cupy_sinc'>`

Elementwise sinc function.

See also:

`numpy.sinc()`

## Floating point routines

<i>signbit</i>	Tests elementwise if the sign bit is set (i.e.
<i>copysign</i>	Returns the first argument with the sign bit of the second elementwise.
<i>frexp</i>	Decomposes each element to mantissa and two's exponent.
<i>ldexp</i>	Computes $x1 * 2 ** x2$ elementwise.
<i>nextafter</i>	Computes the nearest neighbor float values towards the second argument.

## Rational routines

<i>lcm</i>	Computes lcm of <i>x1</i> and <i>x2</i> elementwise.
<i>gcd</i>	Computes gcd of <i>x1</i> and <i>x2</i> elementwise.

## Arithmetic operations

<i>add</i>	Adds two arrays elementwise.
<i>reciprocal</i>	Computes $1 / x$ elementwise.
<i>negative</i>	Takes numerical negative elementwise.
<i>multiply</i>	Multiplies two arrays elementwise.
<i>divide</i>	Elementwise true division (i.e.
<i>power</i>	Computes $x1 ** x2$ elementwise.
<i>subtract</i>	Subtracts arguments elementwise.
<i>true_divide</i>	Elementwise true division (i.e.
<i>floor_divide</i>	Elementwise floor division (i.e.
<i>fmod</i>	Computes the remainder of C division elementwise.
<i>mod</i>	Computes the remainder of Python division elementwise.
<i>modf</i>	Extracts the fractional and integral parts of an array elementwise.
<i>remainder</i>	Computes the remainder of Python division elementwise.
<i>divmod</i>	

### cupy.divmod

```
cupy.divmod = <ufunc 'cupy_divmod'>
```

## Handling complex numbers

<i>angle</i>	Returns the angle of the complex argument.
<i>real(val)</i>	Returns the real part of the elements of the array.
<i>imag(val)</i>	Returns the imaginary part of the elements of the array.
<i>conj</i>	Returns the complex conjugate, element-wise.
<i>conjugate</i>	Returns the complex conjugate, element-wise.

### cupy.angle

```
cupy.angle = <ufunc 'cupy_angle'>
```

Returns the angle of the complex argument.

**See also:**

`numpy.angle()`

**cupy.real****cupy.real**(*val*)

Returns the real part of the elements of the array.

**See also:**`numpy.real()`**cupy.imag****cupy.imag**(*val*)

Returns the imaginary part of the elements of the array.

**See also:**`numpy.imag()`**Miscellaneous**

<code>convolve(a, v[, mode])</code>	Returns the discrete, linear convolution of two one-dimensional sequences.
<code>clip(a[, a_min, a_max, out])</code>	Clips the values of an array to a given interval.
<code>sqrt</code>	Elementwise square root function.
<code>cbrt</code>	Elementwise cube root function.
<code>square</code>	Elementwise square function.
<code>absolute</code>	Elementwise absolute value function.
<code>sign</code>	Elementwise sign function.
<code>maximum</code>	Takes the maximum of two arrays elementwise.
<code>minimum</code>	Takes the minimum of two arrays elementwise.
<code>fmax</code>	Takes the maximum of two arrays elementwise.
<code>fmin</code>	Takes the minimum of two arrays elementwise.
<code>nan_to_num</code>	Elementwise nan_to_num function.
<code>interp(x, xp, fp[, left, right, period])</code>	One-dimensional linear interpolation.

**cupy.convolve****cupy.convolve**(*a*, *v*, *mode*='full')

Returns the discrete, linear convolution of two one-dimensional sequences.

**Parameters**

- **a** (`cupy.ndarray`) – first 1-dimensional input.
- **v** (`cupy.ndarray`) – second 1-dimensional input.
- **mode** (`str`, *optional*) – *valid*, *same*, *full*

**Returns** Discrete, linear convolution of *a* and *v*.**Return type** `cupy.ndarray`**See also:**`numpy.convolve()`

## cupy.clip

`cupy.clip(a, a_min=None, a_max=None, out=None)`

Clips the values of an array to a given interval.

This is equivalent to `maximum(minimum(a, a_max), a_min)`, while this function is more efficient.

### Parameters

- **a** (`cupy.ndarray`) – The source array.
- **a\_min** (*scalar*, `cupy.ndarray` or `None`) – The left side of the interval. When it is `None`, it is ignored.
- **a\_max** (*scalar*, `cupy.ndarray` or `None`) – The right side of the interval. When it is `None`, it is ignored.
- **out** (`cupy.ndarray`) – Output array.

**Returns** Clipped array.

**Return type** `cupy.ndarray`

See also:

`numpy.clip()`

## cupy.nan\_to\_num

`cupy.nan_to_num = <ufunc 'cupy_nan_to_num'>`

Elementwise `nan_to_num` function.

See also:

`numpy.nan_to_num`

## cupy.interp

`cupy.interp(x, xp, fp, left=None, right=None, period=None)`

One-dimensional linear interpolation.

### Parameters

- **x** (`cupy.ndarray`) – a 1D array of points on which the interpolation is performed.
- **xp** (`cupy.ndarray`) – a 1D array of points on which the function values (`fp`) are known.
- **fp** (`cupy.ndarray`) – a 1D array containing the function values at the the points `xp`.
- **left** (*float* or *complex*) – value to return if `x < xp[0]`. Default is `fp[0]`.
- **right** (*float* or *complex*) – value to return if `x > xp[-1]`. Default is `fp[-1]`.
- **period** (`None` or *float*) – a period for the x-coordinates. Parameters `left` and `right` are ignored if `period` is specified. Default is `None`.

**Returns** The interpolated values, same shape as `x`.

**Return type** `cupy.ndarray`

---

**Note:** This function may synchronize if `left` or `right` is not already on the device.

---

**See also:**

`numpy.interp()`

### 5.3.12 Miscellaneous routines

---

**Hint:** NumPy API Reference: Miscellaneous routines

---

#### Memory ranges

---

`shares_memory(a, b[, max_work])`

---



---

`may_share_memory(a, b[, max_work])`

---

#### `cupy.shares_memory`

`cupy.shares_memory(a, b, max_work=None)`

#### `cupy.may_share_memory`

`cupy.may_share_memory(a, b, max_work=None)`

#### Utility

---

<code>show_config(*[, _full])</code>	Prints the current runtime configuration to standard output.
--------------------------------------	--

---

#### `cupy.show_config`

`cupy.show_config(*, _full=False)`  
 Prints the current runtime configuration to standard output.

## Matlab-like Functions

---

<code>who([vardict])</code>	Print the CuPy arrays in the given dictionary.
-----------------------------	--

---

### cupy.who

`cupy.who(vardict=None)`

Print the CuPy arrays in the given dictionary.

Prints out the name, shape, bytes and type of all of the ndarrays present in *vardict*.

If there is no dictionary passed in or *vardict* is None then returns CuPy arrays in the `globals()` dictionary (all CuPy arrays in the namespace).

**Parameters** **vardict** – (None or dict) A dictionary possibly containing ndarrays. Default is `globals()` if None specified

---

#### Example

```
>>> a = cupy.arange(10)
>>> b = cupy.ones(20)
>>> cupy.who()
Name          Shape          Bytes          Type
=====
a              10              80             int64
b              20             160            float64

Upper bound on total bytes =      240
>>> d = {'x': cupy.arange(2.0),
...      'y': cupy.arange(3.0), 'txt': 'Some str',
...      'idx': 5}
>>> cupy.who(d)
Name          Shape          Bytes          Type
=====
x              2              16             float64
y              3              24             float64

Upper bound on total bytes =      40
```

---

### 5.3.13 Padding arrays

---

**Hint:** [NumPy API Reference: Padding arrays](#)

---

---

<code>pad(array, pad_width[, mode])</code>	Pads an array with specified widths and values.
--	---

---



## cupy.pad

`cupy.pad(array, pad_width, mode='constant', **kwargs)`

Pads an array with specified widths and values.

### Parameters

- **array** (`cupy.ndarray`) – The array to pad.
- **pad\_width** (*sequence, array\_like or int*) – Number of values padded to the edges of each axis. ((before\_1, after\_1), ... (before\_N, after\_N)) unique pad widths for each axis. ((before, after),) yields same before and after pad for each axis. (pad,) or int is a shortcut for before = after = pad width for all axes. You cannot specify `cupy.ndarray`.
- **mode** (*str or function, optional*) – One of the following string values or a user supplied function
  - 'constant' (default) Pads with a constant value.
  - 'edge' Pads with the edge values of array.
  - 'linear\_ramp' Pads with the linear ramp between end\_value and the array edge value.
  - 'maximum' Pads with the maximum value of all or part of the vector along each axis.
  - 'mean' Pads with the mean value of all or part of the vector along each axis.
  - 'median' Pads with the median value of all or part of the vector along each axis. (Not Implemented)
  - 'minimum' Pads with the minimum value of all or part of the vector along each axis.
  - 'reflect' Pads with the reflection of the vector mirrored on the first and last values of the vector along each axis.
  - 'symmetric' Pads with the reflection of the vector mirrored along the edge of the array.
  - 'wrap' Pads with the wrap of the vector along the axis. The first values are used to pad the end and the end values are used to pad the beginning.
  - 'empty' Pads with undefined values.
  - <function> Padding function, see Notes.
- **stat\_length** (*sequence or int, optional*) – Used in 'maximum', 'mean', 'median', and 'minimum'. Number of values at edge of each axis used to calculate the statistic value. ((before\_1, after\_1), ... (before\_N, after\_N)) unique statistic lengths for each axis. ((before, after),) yields same before and after statistic lengths for each axis. (stat\_length,) or int is a shortcut for before = after = statistic length for all axes. Default is None, to use the entire axis. You cannot specify `cupy.ndarray`.
- **constant\_values** (*sequence or scalar, optional*) – Used in 'constant'. The values to set the padded values for each axis. ((before\_1, after\_1), ... (before\_N, after\_N)) unique pad constants for each axis. ((before, after),) yields same before and after constants for each axis. (constant,) or constant is a shortcut for before = after = constant for all axes. Default is 0. You cannot specify `cupy.ndarray`.
- **end\_values** (*sequence or scalar, optional*) – Used in 'linear\_ramp'. The values used for the ending value of the linear\_ramp and that will form the edge of the padded array. ((before\_1, after\_1), ... (before\_N, after\_N)) unique end values for each axis. ((before, after),) yields same before and after end values for each axis. (constant,) or constant is a shortcut for before = after = constant for all axes. Default is 0. You cannot specify `cupy.ndarray`.

- **reflect\_type** ({'even', 'odd'}, optional) – Used in 'reflect', and 'symmetric'. The 'even' style is the default with an unaltered reflection around the edge value. For the 'odd' style, the extended part of the array is created by subtracting the reflected values from two times the edge value.

**Returns** Padded array with shape extended by `pad_width`.

**Return type** `cupy.ndarray`

---

**Note:** For an array with rank greater than 1, some of the padding of later axes is calculated from padding of previous axes. This is easiest to think about with a rank 2 array where the corners of the padded array are calculated by using padded values from the first axis.

The padding function, if used, should modify a rank 1 array in-place. It has the following signature:

```
padding_func(vector, iaxis_pad_width, iaxis, kwargs)
```

where

**vector** (`cupy.ndarray`) A rank 1 array already padded with zeros. Padded values are `vector[:iaxis_pad_width[0]]` and `vector[-iaxis_pad_width[1]:]`.

**iaxis\_pad\_width** (`tuple`) A 2-tuple of ints, `iaxis_pad_width[0]` represents the number of values padded at the beginning of vector where `iaxis_pad_width[1]` represents the number of values padded at the end of vector.

**iaxis** (`int`) The axis currently being calculated.

**kwargs** (`dict`) Any keyword arguments the function requires.

---

## Examples

```
>>> a = cupy.array([1, 2, 3, 4, 5])
>>> cupy.pad(a, (2, 3), 'constant', constant_values=(4, 6))
array([4, 4, 1, ..., 6, 6, 6])
```

```
>>> cupy.pad(a, (2, 3), 'edge')
array([1, 1, 1, ..., 5, 5, 5])
```

```
>>> cupy.pad(a, (2, 3), 'linear_ramp', end_values=(5, -4))
array([ 5,  3,  1,  2,  3,  4,  5,  2, -1, -4])
```

```
>>> cupy.pad(a, (2,), 'maximum')
array([5, 5, 1, 2, 3, 4, 5, 5, 5])
```

```
>>> cupy.pad(a, (2,), 'mean')
array([3, 3, 1, 2, 3, 4, 5, 3, 3])
```

```
>>> a = cupy.array([[1, 2], [3, 4]])
>>> cupy.pad(a, ((3, 2), (2, 3)), 'minimum')
array([[1, 1, 1, 2, 1, 1, 1],
       [1, 1, 1, 2, 1, 1, 1],
       [1, 1, 1, 2, 1, 1, 1],
       [1, 1, 1, 2, 1, 1, 1],
```

(continues on next page)

(continued from previous page)

```
[3, 3, 3, 4, 3, 3, 3],
[1, 1, 1, 2, 1, 1, 1],
[1, 1, 1, 2, 1, 1, 1]])
```

```
>>> a = cupy.array([1, 2, 3, 4, 5])
>>> cupy.pad(a, (2, 3), 'reflect')
array([3, 2, 1, 2, 3, 4, 5, 4, 3, 2])
```

```
>>> cupy.pad(a, (2, 3), 'reflect', reflect_type='odd')
array([-1, 0, 1, 2, 3, 4, 5, 6, 7, 8])
```

```
>>> cupy.pad(a, (2, 3), 'symmetric')
array([2, 1, 1, 2, 3, 4, 5, 5, 4, 3])
```

```
>>> cupy.pad(a, (2, 3), 'symmetric', reflect_type='odd')
array([0, 1, 1, 2, 3, 4, 5, 5, 6, 7])
```

```
>>> cupy.pad(a, (2, 3), 'wrap')
array([4, 5, 1, 2, 3, 4, 5, 1, 2, 3])
```

```
>>> def pad_with(vector, pad_width, iaxis, kwargs):
...     pad_value = kwargs.get('padder', 10)
...     vector[:pad_width[0]] = pad_value
...     vector[-pad_width[1]:] = pad_value
>>> a = cupy.arange(6)
>>> a = a.reshape((2, 3))
>>> cupy.pad(a, 2, pad_with)
array([[10, 10, 10, 10, 10, 10, 10],
       [10, 10, 10, 10, 10, 10, 10],
       [10, 10, 0, 1, 2, 10, 10],
       [10, 10, 3, 4, 5, 10, 10],
       [10, 10, 10, 10, 10, 10, 10],
       [10, 10, 10, 10, 10, 10, 10]])
>>> cupy.pad(a, 2, pad_with, padder=100)
array([[100, 100, 100, 100, 100, 100, 100],
       [100, 100, 100, 100, 100, 100, 100],
       [100, 100, 0, 1, 2, 100, 100],
       [100, 100, 3, 4, 5, 100, 100],
       [100, 100, 100, 100, 100, 100, 100],
       [100, 100, 100, 100, 100, 100, 100]])
```

### 5.3.14 Polynomials

---

**Hint:** NumPy API Reference: Polynomials

---

#### Power Series (`cupy.polynomial.polynomial`)

---

**Hint:** NumPy API Reference: Power Series (`numpy.polynomial.polynomial`)

---

#### Misc Functions

---

<code>polyvander(x, deg)</code>	Computes the Vandermonde matrix of given degree.
<code>polycompanion(c)</code>	Computes the companion matrix of c.

---

#### `cupy.polynomial.polynomial.polyvander`

`cupy.polynomial.polynomial.polyvander(x, deg)`  
Computes the Vandermonde matrix of given degree.

**Parameters**

- **x** (`cupy.ndarray`) – array of points
- **deg** (`int`) – degree of the resulting matrix.

**Returns** The Vandermonde matrix

**Return type** `cupy.ndarray`

**See also:**

`numpy.polynomial.polynomial.polyvander()`

#### `cupy.polynomial.polynomial.polycompanion`

`cupy.polynomial.polynomial.polycompanion(c)`  
Computes the companion matrix of c.

**Parameters** **c** (`cupy.ndarray`) – 1-D array of polynomial coefficients ordered from low to high degree.

**Returns** Companion matrix of dimensions (deg, deg).

**Return type** `cupy.ndarray`

**See also:**

`numpy.polynomial.polynomial.polycompanion()`

## Polyutils

---

**Hint:** NumPy API Reference: Polyutils

---

## Functions

<code>as_series(alist[, trim])</code>	Returns argument as a list of 1-d arrays.
<code>trimseq(seq)</code>	Removes small polynomial series coefficients.
<code>trimcoef(c[, tol])</code>	Removes small trailing coefficients from a polynomial.

### `cupy.polynomial.polyutils.as_series`

`cupy.polynomial.polyutils.as_series(alist, trim=True)`

Returns argument as a list of 1-d arrays.

**Parameters**

- **alist** (`cupy.ndarray` or list of `cupy.ndarray`) – 1-D or 2-D input array.
- **trim** (*bool*, optional) – trim trailing zeros.

**Returns** list of 1-D arrays.

**Return type** list of `cupy.ndarray`

See also:

`numpy.polynomial.polyutils.as_series()`

### `cupy.polynomial.polyutils.trimseq`

`cupy.polynomial.polyutils.trimseq(seq)`

Removes small polynomial series coefficients.

**Parameters** **seq** (`cupy.ndarray`) – input array.

**Returns** input array with trailing zeros removed. If the resulting output is empty, it returns the first element.

**Return type** `cupy.ndarray`

See also:

`numpy.polynomial.polyutils.trimseq()`

### cupy.polynomial.polyutils.trimcoef

`cupy.polynomial.polyutils.trimcoef(c, tol=0)`

Removes small trailing coefficients from a polynomial.

#### Parameters

- **c** (`cupy.ndarray`) – 1d array of coefficients from lowest to highest order.
- **tol** (*number, optional*) – trailing coefficients whose absolute value are less than or equal to `tol` are trimmed.

**Returns** trimmed 1d array.

**Return type** `cupy.ndarray`

See also:

`numpy.polynomial.polyutils.trimcoef()`

## Poly1d

---

**Hint:** NumPy API Reference: `Poly1d`

---

### Basics

<code>poly1d(c_or_r[, r, variable])</code>	A one-dimensional polynomial class.
<code>polyval(p, x)</code>	Evaluates a polynomial at specific values.
<code>roots(p)</code>	Computes the roots of a polynomial with given coefficients.

### cupy.poly1d

**class** `cupy.poly1d(c_or_r, r=False, variable=None)`

A one-dimensional polynomial class.

---

**Note:** This is a counterpart of an old polynomial class in NumPy. Note that the new NumPy polynomial API (`numpy.polynomial.polynomial`) has different convention, e.g. order of coefficients is reversed.

---

#### Parameters

- **c\_or\_r** (*array\_like*) – The polynomial’s coefficients in decreasing powers
- **r** (*bool, optional*) – If True, `c_or_r` specifies the polynomial’s roots; the default is False.
- **variable** (*str, optional*) – Changes the variable used when printing the polynomial from `x` to `variable`

See also:

`numpy.poly1d`

## Methods

**\_\_call\_\_**(\*args, \*\*kwargs)

Call self as a function.

**\_\_getitem\_\_**(key, /)

Return self[key].

**\_\_setitem\_\_**(key, value, /)

Set self[key] to value.

**\_\_len\_\_**()

Return len(self).

**\_\_iter\_\_**()

Implement iter(self).

**deriv**(self, m=1)

**get**(self, stream=None)

Returns a copy of poly1d object on host memory.

**Parameters** **stream** ([cupy.cuda.Stream](#)) – CUDA stream object. If it is given, the copy runs asynchronously. Otherwise, the copy is synchronous. The default uses CUDA stream object of the current context.

**Returns** Copy of poly1d object on host memory.

**Return type** [numpy.poly1d](#)

**integ**(self, m=1, k=0)

**set**(self, polyin, stream=None)

Copies a poly1d object on the host memory to [cupy.poly1d](#).

**Parameters**

- **polyin** ([numpy.poly1d](#)) – The source object on the host memory.
- **stream** ([cupy.cuda.Stream](#)) – CUDA stream object. If it is given, the copy runs asynchronously. Otherwise, the copy is synchronous. The default uses CUDA stream object of the current context.

**\_\_eq\_\_**(value, /)

Return self==value.

**\_\_ne\_\_**(value, /)

Return self!=value.

**\_\_lt\_\_**(value, /)

Return self<value.

**\_\_le\_\_**(value, /)

Return self<=value.

**\_\_gt\_\_**(value, /)

Return self>value.

**\_\_ge\_\_**(value, /)

Return self>=value.

### Attributes

`c`  
`coef`  
`coefficients`  
`coeffs`  
`o`  
`order`  
`r`  
`roots`  
`variable`

### `cupy.polyval`

`cupy.polyval(p, x)`

Evaluates a polynomial at specific values.

#### Parameters

- **p** (`cupy.ndarray` or `cupy.poly1d`) – input polynomial.
- **x** (*scalar*, `cupy.ndarray`) – values at which the polynomial
- **evaluated.** (*is*) –

**Returns** polynomial evaluated at x.

**Return type** `cupy.ndarray` or `cupy.poly1d`

**Warning:** This function doesn't currently support `poly1d` values to evaluate.

See also:

`numpy.polyval()`

### `cupy.roots`

`cupy.roots(p)`

Computes the roots of a polynomial with given coefficients.

**Parameters** **p** (`cupy.ndarray` or `cupy.poly1d`) – polynomial coefficients.

**Returns** polynomial roots.

**Return type** `cupy.ndarray`

**Warning:** This function doesn't support currently polynomial coefficients whose companion matrices are general 2d square arrays. Only those with complex Hermitian or real symmetric 2d arrays are allowed.

The current `cupy.roots` doesn't guarantee the order of results.



See also:

`numpy.roots()`

## Fitting

---

<code>polyfit(x, y, deg[, rcond, full, w, cov])</code>	Returns the least squares fit of polynomial of degree <code>deg</code> to the data <code>y</code> sampled at <code>x</code> .
--	---

---

### `cupy.polyfit`

`cupy.polyfit(x, y, deg, rcond=None, full=False, w=None, cov=False)`

Returns the least squares fit of polynomial of degree `deg` to the data `y` sampled at `x`.

#### Parameters

- **x** (`cupy.ndarray`) – x-coordinates of the sample points of shape (M,).
- **y** (`cupy.ndarray`) – y-coordinates of the sample points of shape (M,) or (M, K).
- **deg** (`int`) – degree of the fitting polynomial.
- **rcond** (`float`, *optional*) – relative condition number of the fit. The default value is `len(x) * eps`.
- **full** (`bool`, *optional*) – indicator of the return value nature. When False (default), only the coefficients are returned. When True, diagnostic information is also returned.
- **w** (`cupy.ndarray`, *optional*) – weights applied to the y-coordinates of the sample points of shape (M,).
- **cov** (`bool` or `str`, *optional*) – if given, returns the coefficients along with the covariance matrix.

#### Returns

**p** (`cupy.ndarray` of shape **(deg + 1)** or **(deg + 1, K)**): Polynomial coefficients from highest to lowest degree

**residuals, rank, singular\_values, rcond** (`cupy.ndarray`, `int`, `cupy.ndarray`, `float`): Present only if `full=True`. Sum of squared residuals of the least-squares fit, rank of the scaled Vandermonde coefficient matrix, its singular values, and the specified value of `rcond`.

**V** (`cupy.ndarray` of shape **(M, M)** or **(M, M, K)**): Present only if `full=False` and `cov=True`. The covariance matrix of the polynomial coefficient estimates.

**Return type** `cupy.ndarray` or `tuple`

**Warning:** `numpy.RankWarning`: The rank of the coefficient matrix in the least-squares fit is deficient. It is raised if `full=False`.

See also:

`numpy.polyfit()`

## Arithmetic

<code>polyadd(a1, a2)</code>	Computes the sum of two polynomials.
<code>polysub(a1, a2)</code>	Computes the difference of two polynomials.
<code>polymul(a1, a2)</code>	Computes the product of two polynomials.

### cupy.polyadd

`cupy.polyadd(a1, a2)`

Computes the sum of two polynomials.

**Parameters**

- **a1** (*scalar*, `cupy.ndarray` or `cupy.poly1d`) – first input polynomial.
- **a2** (*scalar*, `cupy.ndarray` or `cupy.poly1d`) – second input polynomial.

**Returns** The sum of the inputs.

**Return type** `cupy.ndarray` or `cupy.poly1d`

See also:

`numpy.polyadd()`

### cupy.polysub

`cupy.polysub(a1, a2)`

Computes the difference of two polynomials.

**Parameters**

- **a1** (*scalar*, `cupy.ndarray` or `cupy.poly1d`) – first input polynomial.
- **a2** (*scalar*, `cupy.ndarray` or `cupy.poly1d`) – second input polynomial.

**Returns** The difference of the inputs.

**Return type** `cupy.ndarray` or `cupy.poly1d`

See also:

`numpy.polysub()`

### cupy.polymul

`cupy.polymul(a1, a2)`

Computes the product of two polynomials.

**Parameters**

- **a1** (*scalar*, `cupy.ndarray` or `cupy.poly1d`) – first input polynomial.
- **a2** (*scalar*, `cupy.ndarray` or `cupy.poly1d`) – second input polynomial.

**Returns** The product of the inputs.

**Return type** `cupy.ndarray` or `cupy.poly1d`

See also:

`numpy.polymul()`

### 5.3.15 Random sampling (`cupy.random`)

Differences between `cupy.random` and `numpy.random`:

- Most functions under `cupy.random` support the `dtype` option, which do not exist in the corresponding NumPy APIs. This option enables generation of float32 values directly without any space overhead.
- `cupy.random.default_rng()` uses XORWOW bit generator by default.
- Random states cannot be serialized. See the description below for details.
- CuPy does not guarantee that the same number generator is used across major versions. This means that numbers generated by `cupy.random` by new major version may not be the same as the previous one, even if the same seed and distribution are used.

#### New Random Generator API

---

**Hint:** NumPy API Reference: Random sampling (`numpy.random`)

---

#### Random Generator

---

**Hint:** NumPy API Reference: Random Generator

---

<code>default_rng([seed])</code>	Construct a new Generator with the default BitGenerator (XORWOW).
<code>Generator(bit_generator)</code>	Container for the BitGenerators.

#### `cupy.random.default_rng`

`cupy.random.default_rng(seed=None)`

Construct a new Generator with the default BitGenerator (XORWOW).

**Parameters** `seed` (`None`, `int`, `array_like[ints]`, `numpy.random.SeedSequence`, `cupy.random.BitGenerator`, `cupy.random.Generator`, *optional*) – A seed to initialize the `cupy.random.BitGenerator`. If an `int` or `array_like[ints]` or `None` is passed, then it will be passed to `numpy.random.SeedSequence` to derive the initial `BitGenerator` state. One may also pass in a `SeedSequence` instance. Additionally, when passed `:class: 'BitGenerator'`, it will be wrapped by `Generator`. If passed a `Generator`, it will be returned unaltered.

**Returns** The initialized generator object.

**Return type** `Generator`

## cupy.random.Generator

**class** `cupy.random.Generator(bit_generator)`

Container for the BitGenerators.

`Generator` exposes a number of methods for generating random numbers drawn from a variety of probability distributions. In addition to the distribution-specific arguments, each method takes a keyword argument *size* that defaults to `None`. If *size* is `None`, then a single value is generated and returned. If *size* is an integer, then a 1-D array filled with generated values is returned. If *size* is a tuple, then an array with that shape is filled and returned. The function `numpy.random.default_rng()` will instantiate a *Generator* with numpy's default *BitGenerator*.  
**No Compatibility Guarantee** `Generator` does not provide a version compatibility guarantee. In particular, as better algorithms evolve the bit stream may change.

**Parameters** `bit_generator` – (`cupy.random.BitGenerator`): *BitGenerator* to use as the core generator.

### Methods

**beta**(*self*, *a*, *b*, *size=None*, *dtype=numpy.float64*)

Beta distribution.

Returns an array of samples drawn from the beta distribution. Its probability density function is defined as

$$f(x) = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha, \beta)}.$$

#### Parameters

- **a** (*float*) – Parameter of the beta distribution  $\alpha$ .
- **b** (*float*) – Parameter of the beta distribution  $\beta$ .
- **size** (*int or tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

**Returns** Samples drawn from the beta distribution.

**Return type** `cupy.ndarray`

**See also:**

`numpy.random.Generator.beta()`

**exponential**(*self*, *scale=1.0*, *size=None*)

Exponential distribution.

Returns an array of samples drawn from the exponential distribution. Its probability density function is defined as

$$f(x) = \frac{1}{\beta} \exp\left(-\frac{x}{\beta}\right).$$

#### Parameters

- **scale** (*float or array\_like of floats*) – The scale parameter  $\beta$ .
- **size** (*int or tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.

**Returns** Samples drawn from the exponential distribution.

**Return type** `cupy.ndarray`

**See also:**

`numpy.random.Generator.exponential()`

**gamma**(*self*, *shape*, *scale*=1.0, *size*=None)

Gamma distribution.

Returns an array of samples drawn from the gamma distribution. Its probability density function is defined as

$$f(x) = \frac{1}{\Gamma(k)\theta^k} x^{k-1} e^{-x/\theta}.$$

**Parameters**

- **shape** (*float* or *array\_like of float*) – The shape of the gamma distribution. Must be non-negative.
- **scale** (*float* or *array\_like of float*) – The scale of the gamma distribution. Must be non-negative. Default equals to 1
- **size** (*int* or *tuple of ints*) – The shape of the array. If None, a zero-dimensional array is generated.

**See also:**

- `numpy.random.Generator.gamma()`

**integers**(*self*, *low*, *high*=None, *size*=None, *dtype*=`numpy.int64`, *endpoint*=False)

Returns a scalar or an array of integer values over an interval.

Each element of returned values are independently sampled from uniform distribution over the [*low*, *high*) or [*low*, *high*] intervals.

**Parameters**

- **low** (*int*) – If *high* is not None, it is the lower bound of the interval. Otherwise, it is the **upper** bound of the interval and lower bound of the interval is set to 0.
- **high** (*int*) – Upper bound of the interval.
- **size** (*None* or *int* or *tuple of ints*) – The shape of returned value.
- **dtype** – Data type specifier.
- **endpoint** (*bool*) – If True, sample from [*low*, *high*]. Defaults to False

**Returns** If *size* is None, it is single integer sampled. If *size* is integer, it is the 1D-array of length *size* element. Otherwise, it is the array whose shape specified by *size*.

**Return type** `int` or `cupy.ndarray` of `ints`

**See also:**

- `numpy.random.Generator.integers()`

**poisson**(*self*, *lam*=1.0, *size*=None)

Poisson distribution.

Returns an array of samples drawn from the poisson distribution. Its probability mass function is defined as

$$f(x) = \frac{\lambda^x e^{-\lambda}}{x!}.$$

#### Parameters

- **lam** (*array\_like of floats*) – Parameter of the poisson distribution  $\lambda$ .
- **size** (*int or tuple of ints*) – The shape of the array. If None, this function generate an array whose shape is `lam.shape`.

**Returns** Samples drawn from the poisson distribution.

**Return type** *cupy.ndarray*

**See also:**

`numpy.random.Generator.poisson()`

**random**(*self*, *size=None*, *dtype=numpy.float64*, *out=None*)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the “continuous uniform” distribution over the stated interval. To sample  $Unif[a, b)$ ,  $b > a$  multiply the output of *random* by  $(b-a)$  and add  $a$ :

$$(b - a) * \text{random}() + a$$

#### Parameters

- **size** (*None or int or tuple of ints*) – The shape of returned value.
- **dtype** – Data type specifier.
- **out** (*cupy.ndarray, optional*) – If specified, values will be written to this array

**Returns** Samples uniformly drawn from the [0, 1) interval

**Return type** *cupy.ndarray*

**See also:**

- `numpy.random.Generator.random()`

**standard\_exponential**(*self*, *size=None*, *dtype=numpy.float64*, *method='inv'*, *out=None*)

Standard exponential distribution.

Returns an array of samples drawn from the standard exponential distribution. Its probability density function is defined as

$$f(x) = e^{-x}.$$

#### Parameters

- **size** (*int or tuple of ints*) – The shape of the array. If None, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

- **method** (*str*) – Method to sample. Currently only 'inv', sampling from the default inverse CDF, is supported.
- **out** (*cupy.ndarray, optional*) – If specified, values will be written to this array

**Returns** Samples drawn from the standard exponential distribution.

**Return type** *cupy.ndarray*

**See also:**

`numpy.random.Generator.standard_exponential()`

**standard\_gamma**(*self, shape, size=None, dtype=numpy.float64, out=None*)

Standard gamma distribution.

Returns an array of samples drawn from the standard gamma distribution. Its probability density function is defined as

$$f(x) = \frac{1}{\Gamma(k)} x^{k-1} e^{-x}.$$

**Parameters**

- **shape** (*float or array\_like of float*) – The shape of the gamma distribution. Must be non-negative.
- **size** (*int or tuple of ints*) – The shape of the array. If None, a zero-dimensional array is generated.
- **dtype** – Data type specifier.
- **out** (*cupy.ndarray, optional*) – If specified, values will be written to this array

**See also:**

- `numpy.random.Generator.standard_gamma()`

**standard\_normal**(*self, size=None, dtype=numpy.float64, out=None*)

Standard normal distribution.

Returns an array of samples drawn from the standard normal distribution.

**Parameters**

- **size** (*int or tuple of ints*) – The shape of the array. If None, a zero-dimensional array is generated.
- **dtype** – Data type specifier.
- **out** (*cupy.ndarray, optional*) – If specified, values will be written to this array

**Returns** Samples drawn from the standard normal distribution.

**Return type** *cupy.ndarray*

**See also:**

- `numpy.random.Generator.standard_normal()`

**\_\_eq\_\_**(*value, /*)

Return self==value.

**\_\_ne\_\_**(*value, /*)

Return self!=value.

```
__lt__(value, /)
    Return self<value.

__le__(value, /)
    Return self<=value.

__gt__(value, /)
    Return self>value.

__ge__(value, /)
    Return self>=value.
```

## Bit Generators

---

**Hint:** [NumPy API Reference: Bit Generators](#)

---

---

<i>BitGenerator</i> ([seed])	Generic BitGenerator.
------------------------------	-----------------------

---

### cupy.random.BitGenerator

**class** `cupy.random.BitGenerator`(*seed=None*)

Generic BitGenerator.

Base Class for generic BitGenerators, which provide a stream of random bits based on different algorithms. Must be overridden.

**Parameters** *seed* (*int*, *array\_like[ints]*, *numpy.random.SeedSequence*, *optional*) – A seed to initialize the *BitGenerator*. If *None*, then fresh, unpredictable entropy will be pulled from the OS. If an *int* or *array\_like[ints]* is passed, then it will be passed to `~numpy.random.SeedSequence` to derive the initial *BitGenerator* state. One may also pass in a *SeedSequence* instance.

### Methods

**random\_raw**(*self*, *size=None*, *output=True*)

```
__eq__(value, /)
    Return self==value.

__ne__(value, /)
    Return self!=value.

__lt__(value, /)
    Return self<value.

__le__(value, /)
    Return self<=value.

__gt__(value, /)
    Return self>value.

__ge__(value, /)
    Return self>=value.
```



CuPy provides the following bit generator implementations:

<code>XORWOW</code> ([seed, size])	BitGenerator that uses cuRAND XORWOW device generator.
<code>MRG32k3a</code> ([seed, size])	BitGenerator that uses cuRAND MRG32k3a device generator.
<code>Philox4x3210</code> ([seed, size])	BitGenerator that uses cuRAND Philox4x3210 device generator.

## cupy.random.XORWOW

**class** `cupy.random.XORWOW`(seed=None, \*, size=256000)

BitGenerator that uses cuRAND XORWOW device generator.

This generator allocates the state using the cuRAND device API.

### Parameters

- **seed** (*None*, *int*, *array\_like[ints]*, *numpy.random.SeedSequence*) – A seed to initialize the *BitGenerator*. If *None*, then fresh, unpredictable entropy will be pulled from the OS. If an *int* or *array\_like[ints]* is passed, then it will be passed to `~numpy.random.SeedSequence` to derive the initial *BitGenerator* state. One may also pass in a *SeedSequence* instance.
- **size** (*int*) – Maximum number of samples that can be generated at once. defaults to 1000 \* 256.

### Methods

**random\_raw**(self, size=None, output=True)

Return randoms as generated by the underlying BitGenerator.

### Parameters

- **size** (*int* or *tuple of ints*, *optional*) – Output shape. If the given shape is, e.g., (m, n, k), then  $m * n * k$  samples are drawn. Default is *None*, in which case a single value is returned.
- **output** (*bool*, *optional*) – Output values. Used for performance testing since the generated values are not returned.

**Returns** Drawn samples.

**Return type** *cupy.ndarray*

---

**Note:** This method directly exposes the the raw underlying pseudo-random number generator. All values are returned as unsigned 64-bit values irrespective of the number of bits produced by the PRNG. See the class docstring for the number of bits returned.

---

**state**(self)

**\_\_eq\_\_**(value, /)

Return self==value.

**\_\_ne\_\_**(value, /)

Return self!=value.

```
__lt__(value, /)
    Return self<value.

__le__(value, /)
    Return self<=value.

__gt__(value, /)
    Return self>value.

__ge__(value, /)
    Return self>=value.
```

## Attributes

**generator** = 0

## cupy.random.MRG32k3a

**class** cupy.random.MRG32k3a(*seed=None, \*, size=256000*)  
BitGenerator that uses cuRAND MRG32k3a device generator.

This generator allocates the state using the cuRAND device API.

### Parameters

- **seed** (*int*, *array\_like[ints]*, *numpy.random.SeedSequence*, *optional*) – A seed to initialize the *BitGenerator*. If *None*, then fresh, unpredictable entropy will be pulled from the OS. If an *int* or *array\_like[ints]* is passed, then it will be passed to `~numpy.random.SeedSequence`` to derive the initial *BitGenerator* state. One may also pass in a *SeedSequence* instance.
- **size** (*int*) – Maximum number of samples that can be generated at once. defaults to 1000 \* 256.

## Methods

**random\_raw**(*self, size=None, output=True*)  
Return randoms as generated by the underlying BitGenerator.

### Parameters

- **size** (*int* or *tuple of ints*, *optional*) – Output shape. If the given shape is, e.g., (m, n, k), then m \* n \* k samples are drawn. Default is *None*, in which case a single value is returned.
- **output** (*bool*, *optional*) – Output values. Used for performance testing since the generated values are not returned.

**Returns** Drawn samples.

**Return type** *cupy.ndarray*

---

**Note:** This method directly exposes the the raw underlying pseudo-random number generator. All values are returned as unsigned 64-bit values irrespective of the number of bits produced by the PRNG. See the class docstring for the number of bits returned.

---

**state**(*self*)

```

__eq__(value, /)
    Return self==value.

__ne__(value, /)
    Return self!=value.

__lt__(value, /)
    Return self<value.

__le__(value, /)
    Return self<=value.

__gt__(value, /)
    Return self>value.

__ge__(value, /)
    Return self>=value.

```

### Attributes

**generator** = 1

## cupy.random.Philox4x3210

**class** cupy.random.**Philox4x3210**(seed=None, \*, size=256000)  
 BitGenerator that uses cuRAND Philox4x3210 device generator.

This generator allocates the state using the cuRAND device API.

### Parameters

- **seed** (*int*, *array\_like[ints]*, *numpy.random.SeedSequence*, *optional*) – A seed to initialize the *BitGenerator*. If None, then fresh, unpredictable entropy will be pulled from the OS. If an *int* or *array\_like[ints]* is passed, then it will be passed to `~numpy.random.SeedSequence` to derive the initial *BitGenerator* state. One may also pass in a *SeedSequence* instance.
- **size** (*int*) – Maximum number of samples that can be generated at once. defaults to 1000 \* 256.

### Methods

**random\_raw**(self, size=None, output=True)  
 Return randoms as generated by the underlying BitGenerator.

### Parameters

- **size** (*int* or *tuple of ints*, *optional*) – Output shape. If the given shape is, e.g., (m, n, k), then m \* n \* k samples are drawn. Default is None, in which case a single value is returned.
- **output** (*bool*, *optional*) – Output values. Used for performance testing since the generated values are not returned.

**Returns** Drawn samples.

**Return type** *cupy.ndarray*

---

**Note:** This method directly exposes the the raw underlying pseudo-random number generator. All values are returned as unsigned 64-bit values irrespective of the number of bits produced by the PRNG. See the class docstring for the number of bits returned.

---

```
state(self)

__eq__(value, /)
    Return self==value.

__ne__(value, /)
    Return self!=value.

__lt__(value, /)
    Return self<value.

__le__(value, /)
    Return self<=value.

__gt__(value, /)
    Return self>value.

__ge__(value, /)
    Return self>=value.
```

### Attributes

**generator** = 2

## Legacy Random Generation

---

### Hint:

- [NumPy API Reference: Legacy Random Generation](#)
  - [NumPy 1.16 Reference](#)
- 

---

*RandomState*([seed, method])

Portable container of a pseudo-random number generator.

---

### cupy.random.RandomState

**class** `cupy.random.RandomState`(seed=None, method=100)

Portable container of a pseudo-random number generator.

An instance of this class holds the state of a random number generator. The state is available only on the device which has been current at the initialization of the instance.

Functions of `cupy.random` use global instances of this class. Different instances are used for different devices. The global state for the current device can be obtained by the `cupy.random.get_random_state()` function.

#### Parameters

- **seed** (*None* or *int*) – Seed of the random number generator. See the `seed()` method for

detail.

- **method** (*int*) – Method of the random number generator. Following values are available:

```
cupy.cuda.curand.CURAND_RNG_PSEUDO_DEFAULT
cupy.cuda.curand.CURAND_RNG_PSEUDO_XORWOW
cupy.cuda.curand.CURAND_RNG_PSEUDO_MRG32K3A
cupy.cuda.curand.CURAND_RNG_PSEUDO_MTGP32
cupy.cuda.curand.CURAND_RNG_PSEUDO_MT19937
cupy.cuda.curand.CURAND_RNG_PSEUDO_PHILOX4_32_10
```

## Methods

**beta**(*a*, *b*, *size=None*, *dtype=<class 'float'>*)

Returns an array of samples drawn from the beta distribution.

**See also:**

- [`cupy.random.beta\(\)`](#) for full documentation
- [`numpy.random.RandomState.beta\(\)`](#)

**binomial**(*n*, *p*, *size=None*, *dtype=<class 'int'>*)

Returns an array of samples drawn from the binomial distribution.

**See also:**

- [`cupy.random.binomial\(\)`](#) for full documentation
- [`numpy.random.RandomState.binomial\(\)`](#)

**chisquare**(*df*, *size=None*, *dtype=<class 'float'>*)

Returns an array of samples drawn from the chi-square distribution.

**See also:**

- [`cupy.random.chisquare\(\)`](#) for full documentation
- [`numpy.random.RandomState.chisquare\(\)`](#)

**choice**(*a*, *size=None*, *replace=True*, *p=None*)

Returns an array of random values from a given 1-D array.

**See also:**

- [`cupy.random.choice\(\)`](#) for full documentation
- [`numpy.random.choice\(\)`](#)

**dirichlet**(*alpha*, *size=None*, *dtype=<class 'float'>*)

Returns an array of samples drawn from the dirichlet distribution.

**See also:**

- [`cupy.random.dirichlet\(\)`](#) for full documentation
- [`numpy.random.RandomState.dirichlet\(\)`](#)

**exponential**(*scale=1.0, size=None, dtype=<class 'float'>*)

Returns an array of samples drawn from a exponential distribution.

**Warning:** This function may synchronize the device.

**See also:**

- [`cupy.random.exponential\(\)`](#) for full documentation
- [`numpy.random.RandomState.exponential\(\)`](#)

**f**(*dfnum, dfden, size=None, dtype=<class 'float'>*)

Returns an array of samples drawn from the f distribution.

**See also:**

- [`cupy.random.f\(\)`](#) for full documentation
- [`numpy.random.RandomState.f\(\)`](#)

**gamma**(*shape, scale=1.0, size=None, dtype=<class 'float'>*)

Returns an array of samples drawn from a gamma distribution.

**See also:**

- [`cupy.random.gamma\(\)`](#) for full documentation
- [`numpy.random.RandomState.gamma\(\)`](#)

**geometric**(*p, size=None, dtype=<class 'int'>*)

Returns an array of samples drawn from the geometric distribution.

**See also:**

- [`cupy.random.geometric\(\)`](#) for full documentation
- [`numpy.random.RandomState.geometric\(\)`](#)

**gumbel**(*loc=0.0, scale=1.0, size=None, dtype=<class 'float'>*)

Returns an array of samples drawn from a Gumbel distribution.

**See also:**

- [`cupy.random.gumbel\(\)`](#) for full documentation
- [`numpy.random.RandomState.gumbel\(\)`](#)

**hypergeometric**(*ngood, nbad, nsample, size=None, dtype=<class 'int'>*)

Returns an array of samples drawn from the hypergeometric distribution.

**See also:**

- [`cupy.random.hypergeometric\(\)`](#) for full documentation
- [`numpy.random.RandomState.hypergeometric\(\)`](#)

**laplace**(*loc=0.0, scale=1.0, size=None, dtype=<class 'float'>*)  
Returns an array of samples drawn from the laplace distribution.

See also:

- `cupy.random.laplace()` for full documentation
- `numpy.random.RandomState.laplace()`

**logistic**(*loc=0.0, scale=1.0, size=None, dtype=<class 'float'>*)  
Returns an array of samples drawn from the logistic distribution.

See also:

- `cupy.random.logistic()` for full documentation
- `numpy.random.RandomState.logistic()`

**lognormal**(*mean=0.0, sigma=1.0, size=None, dtype=<class 'float'>*)  
Returns an array of samples drawn from a log normal distribution.

See also:

- `cupy.random.lognormal()` for full documentation
- `numpy.random.RandomState.lognormal()`

**logseries**(*p, size=None, dtype=<class 'int'>*)  
Returns an array of samples drawn from a log series distribution.

**Warning:** This function may synchronize the device.

See also:

- `cupy.random.logseries()` for full documentation
- `numpy.random.RandomState.logseries()`

**multivariate\_normal**(*mean, cov, size=None, check\_valid='ignore', tol=1e-08, method='cholesky', dtype=<class 'float'>*)  
Returns an array of samples drawn from the multivariate normal distribution.

**Warning:** This function calls one or more cuSOLVER routine(s) which may yield invalid results if input conditions are not met. To detect these invalid results, you can set the `linalg` configuration to a value that is not `ignore` in `cupyx.errstate()` or `cupyx.seterr()`.

See also:

- `cupy.random.multivariate_normal()` for full documentation
- `numpy.random.RandomState.multivariate_normal()`

**negative\_binomial**(*n, p, size=None, dtype=<class 'int'>*)  
Returns an array of samples drawn from the negative binomial distribution.

**Warning:** This function may synchronize the device.

See also:

- `cupy.random.negative_binomial()` for full documentation
- `numpy.random.RandomState.negative_binomial()`

**noncentral\_chisquare**(*df, nonc, size=None, dtype=<class 'float'>*)

Returns an array of samples drawn from the noncentral chi-square distribution.

**Warning:** This function may synchronize the device.

See also:

- `cupy.random.noncentral_chisquare()` for full documentation
- `numpy.random.RandomState.noncentral_chisquare()`

**noncentral\_f**(*dfnum, dfden, nonc, size=None, dtype=<class 'float'>*)

Returns an array of samples drawn from the noncentral F distribution.

**Warning:** This function may synchronize the device.

See also:

- `cupy.random.noncentral_f()` for full documentation
- `numpy.random.RandomState.noncentral_f()`

**normal**(*loc=0.0, scale=1.0, size=None, dtype=<class 'float'>*)

Returns an array of normally distributed samples.

See also:

- `cupy.random.normal()` for full documentation
- `numpy.random.RandomState.normal()`

**pareto**(*a, size=None, dtype=<class 'float'>*)

Returns an array of samples drawn from the pareto II distribution.

See also:

- `cupy.random.pareto()` for full documentation
- `numpy.random.RandomState.pareto()`

**permutation**(*a*)

Returns a permuted range or a permutation of an array.



**poisson**(*lam=1.0, size=None, dtype=<class 'int'>*)

Returns an array of samples drawn from the poisson distribution.

**See also:**

- [`cupy.random.poisson\(\)`](#) for full documentation
- [`numpy.random.RandomState.poisson\(\)`](#)

**power**(*a, size=None, dtype=<class 'float'>*)

Returns an array of samples drawn from the power distribution.

**Warning:** This function may synchronize the device.

**See also:**

- [`cupy.random.power\(\)`](#) for full documentation
- [`numpy.random.RandomState.power\(\)`](#)

**rand**(*\*size, \*\*kwarg*)

Returns uniform random values over the interval `[0, 1)`.

**See also:**

- [`cupy.random.rand\(\)`](#) for full documentation
- [`numpy.random.RandomState.rand\(\)`](#)

**randint**(*low, high=None, size=None, dtype=<class 'int'>*)

Returns a scalar or an array of integer values over `[low, high)`.

**See also:**

- [`cupy.random.randint\(\)`](#) for full documentation
- [`numpy.random.RandomState.randint\(\)`](#)

**randn**(*\*size, \*\*kwarg*)

Returns an array of standard normal random values.

**See also:**

- [`cupy.random.randn\(\)`](#) for full documentation
- [`numpy.random.RandomState.randn\(\)`](#)

**random\_sample**(*size=None, dtype=<class 'float'>*)

Returns an array of random values over the interval `[0, 1)`.

**See also:**

- [`cupy.random.random\_sample\(\)`](#) for full documentation
- [`numpy.random.RandomState.random\_sample\(\)`](#)

**rayleigh**(*scale=1.0, size=None, dtype=<class 'float'>*)  
Returns an array of samples drawn from a rayleigh distribution.

**Warning:** This function may synchronize the device.

**See also:**

- [`cupy.random.rayleigh\(\)`](#) for full documentation
- [`numpy.random.RandomState.rayleigh\(\)`](#)

**seed**(*seed=None*)  
Resets the state of the random number generator with a seed.

**See also:**

- [`cupy.random.seed\(\)`](#) for full documentation
- [`numpy.random.RandomState.seed\(\)`](#)

**shuffle**(*a*)  
Returns a shuffled array.

**See also:**

- [`cupy.random.shuffle\(\)`](#) for full documentation
- [`numpy.random.shuffle\(\)`](#)

**standard\_cauchy**(*size=None, dtype=<class 'float'>*)  
Returns an array of samples drawn from the standard cauchy distribution.

**See also:**

- [`cupy.random.standard\_cauchy\(\)`](#) for full documentation
- [`numpy.random.RandomState.standard\_cauchy\(\)`](#)

**standard\_exponential**(*size=None, dtype=<class 'float'>*)  
Returns an array of samples drawn from the standard exp distribution.

**See also:**

- [`cupy.random.standard\_exponential\(\)`](#) for full documentation
- [`numpy.random.RandomState.standard\_exponential\(\)`](#)

**standard\_gamma**(*shape, size=None, dtype=<class 'float'>*)  
Returns an array of samples drawn from a standard gamma distribution.

**See also:**

- [`cupy.random.standard\_gamma\(\)`](#) for full documentation
- [`numpy.random.RandomState.standard\_gamma\(\)`](#)

**standard\_normal**(*size=None, dtype=<class 'float'>*)

Returns samples drawn from the standard normal distribution.

See also:

- [`cupy.random.standard\_normal\(\)`](#) for full documentation
- [`numpy.random.RandomState.standard\_normal\(\)`](#)

**standard\_t**(*df, size=None, dtype=<class 'float'>*)

Returns an array of samples drawn from the standard t distribution.

See also:

- [`cupy.random.standard\_t\(\)`](#) for full documentation
- [`numpy.random.RandomState.standard\_t\(\)`](#)

**tomaxint**(*size=None*)

Draws integers between 0 and max integer inclusive.

Return a sample of uniformly distributed random integers in the interval  $[0, \text{np.iinfo}(\text{np.int\_}).\text{max}]$ . The `np.int_` type translates to the C long integer type and its precision is platform dependent.

**Parameters** *size* (*int* or *tuple of ints*) – Output shape.

**Returns** Drawn samples.

**Return type** [`cupy.ndarray`](#)

See also:

[`numpy.random.RandomState.tomaxint\(\)`](#)

**triangular**(*left, mode, right, size=None, dtype=<class 'float'>*)

Returns an array of samples drawn from the triangular distribution.

**Warning:** This function may synchronize the device.

See also:

- [`cupy.random.triangular\(\)`](#) for full documentation
- [`numpy.random.RandomState.triangular\(\)`](#)

**uniform**(*low=0.0, high=1.0, size=None, dtype=<class 'float'>*)

Returns an array of uniformly-distributed samples over an interval.

See also:

- [`cupy.random.uniform\(\)`](#) for full documentation
- [`numpy.random.RandomState.uniform\(\)`](#)

**vonmises**(*mu, kappa, size=None, dtype=<class 'float'>*)

Returns an array of samples drawn from the von Mises distribution.

See also:

- [`cupy.random.vonmises\(\)`](#) for full documentation

- `numpy.random.RandomState.vonmises()`

**wald**(*mean, scale, size=None, dtype=<class 'float'>*)

Returns an array of samples drawn from the Wald distribution.

**See also:**

- `cupy.random.wald()` for full documentation
- `numpy.random.RandomState.wald()`

**weibull**(*a, size=None, dtype=<class 'float'>*)

Returns an array of samples drawn from the weibull distribution.

<b>Warning:</b> This function may synchronize the device.
---

**See also:**

- `cupy.random.weibull()` for full documentation
- `numpy.random.RandomState.weibull()`

**zipf**(*a, size=None, dtype=<class 'int'>*)

Returns an array of samples drawn from the Zipf distribution.

<b>Warning:</b> This function may synchronize the device.
---

**See also:**

- `cupy.random.zipf()` for full documentation
- `numpy.random.RandomState.zipf()`

**\_\_eq\_\_**(*value, /*)

Return self==value.

**\_\_ne\_\_**(*value, /*)

Return self!=value.

**\_\_lt\_\_**(*value, /*)

Return self<value.

**\_\_le\_\_**(*value, /*)

Return self<=value.

**\_\_gt\_\_**(*value, /*)

Return self>value.

**\_\_ge\_\_**(*value, /*)

Return self>=value.

Functions in `cupy.random`

<code>beta(a, b[, size, dtype])</code>	Beta distribution.
<code>binomial(n, p[, size, dtype])</code>	Binomial distribution.
<code>bytes(length)</code>	Returns random bytes.
<code>chisquare(df[, size, dtype])</code>	Chi-square distribution.
<code>choice(a[, size, replace, p])</code>	Returns an array of random values from a given 1-D array.
<code>dirichlet(alpha[, size, dtype])</code>	Dirichlet distribution.
<code>exponential(scale[, size, dtype])</code>	Exponential distribution.
<code>f(dfnum, dfden[, size, dtype])</code>	F distribution.
<code>gamma(shape[, scale, size, dtype])</code>	Gamma distribution.
<code>geometric(p[, size, dtype])</code>	Geometric distribution.
<code>gumbel([loc, scale, size, dtype])</code>	Returns an array of samples drawn from a Gumbel distribution.
<code>hypergeometric(ngood, nbad, nsample[, size, ...])</code>	hypergeometric distribution.
<code>laplace([loc, scale, size, dtype])</code>	Laplace distribution.
<code>logistic([loc, scale, size, dtype])</code>	Logistic distribution.
<code>lognormal([mean, sigma, size, dtype])</code>	Returns an array of samples drawn from a log normal distribution.
<code>logseries(p[, size, dtype])</code>	Log series distribution.
<code>multinomial(n, pvals[, size])</code>	Returns an array from multinomial distribution.
<code>multivariate_normal(mean, cov[, size, ...])</code>	Multivariate normal distribution.
<code>negative_binomial(n, p[, size, dtype])</code>	Negative binomial distribution.
<code>noncentral_chisquare(df, nonc[, size, dtype])</code>	Noncentral chisquare distribution.
<code>noncentral_f(dfnum, dfden, nonc[, size, dtype])</code>	Noncentral F distribution.
<code>normal([loc, scale, size, dtype])</code>	Returns an array of normally distributed samples.
<code>pareto(a[, size, dtype])</code>	Pareto II or Lomax distribution.
<code>permutation(a)</code>	Returns a permuted range or a permutation of an array.
<code>poisson([lam, size, dtype])</code>	Poisson distribution.
<code>power(a[, size, dtype])</code>	Power distribution.
<code>rand(*size, **kwarg)</code>	Returns an array of uniform random values over the interval <code>[0, 1)</code> .
<code>randint(low[, high, size, dtype])</code>	Returns a scalar or an array of integer values over <code>[low, high)</code> .
<code>randn(*size, **kwarg)</code>	Returns an array of standard normal random values.
<code>random([size, dtype])</code>	Returns an array of random values over the interval <code>[0, 1)</code> .
<code>random_integers(low[, high, size])</code>	Return a scalar or an array of integer values over <code>[low, high]</code>
<code>random_sample([size, dtype])</code>	Returns an array of random values over the interval <code>[0, 1)</code> .
<code>ranf([size, dtype])</code>	Returns an array of random values over the interval <code>[0, 1)</code> .
<code>rayleigh([scale, size, dtype])</code>	Rayleigh distribution.
<code>sample([size, dtype])</code>	Returns an array of random values over the interval <code>[0, 1)</code> .
<code>seed([seed])</code>	Resets the state of the random number generator with a seed.
<code>shuffle(a)</code>	Shuffles an array.
<code>standard_cauchy([size, dtype])</code>	Standard cauchy distribution.

continues on next page

Table 76 – continued from previous page

<code>standard_exponential([size, dtype])</code>	Standard exponential distribution.
<code>standard_gamma(shape[, size, dtype])</code>	Standard gamma distribution.
<code>standard_normal([size, dtype])</code>	Returns an array of samples drawn from the standard normal distribution.
<code>standard_t(df[, size, dtype])</code>	Standard Student's t distribution.
<code>triangular(left, mode, right[, size, dtype])</code>	Triangular distribution.
<code>uniform([low, high, size, dtype])</code>	Returns an array of uniformly-distributed samples over an interval.
<code>vonmises(mu, kappa[, size, dtype])</code>	von Mises distribution.
<code>wald(mean, scale[, size, dtype])</code>	Wald distribution.
<code>weibull(a[, size, dtype])</code>	weibull distribution.
<code>zipf(a[, size, dtype])</code>	Zipf distribution.

## cupy.random.beta

`cupy.random.beta(a, b, size=None, dtype=<class 'float'>)`

Beta distribution.

Returns an array of samples drawn from the beta distribution. Its probability density function is defined as

$$f(x) = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha, \beta)}.$$

### Parameters

- **a** (*float*) – Parameter of the beta distribution  $\alpha$ .
- **b** (*float*) – Parameter of the beta distribution  $\beta$ .
- **size** (*int* or *tuple of ints*) – The shape of the array. If *None*, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

**Returns** Samples drawn from the beta distribution.

**Return type** `cupy.ndarray`

See also:

`numpy.random.beta()`

## cupy.random.binomial

`cupy.random.binomial(n, p, size=None, dtype=<class 'int'>)`

Binomial distribution.

Returns an array of samples drawn from the binomial distribution. Its probability mass function is defined as

$$f(x) = \binom{n}{x} p^x (1-p)^{n-x}.$$

### Parameters

- **n** (*int*) – Trial number of the binomial distribution.
- **p** (*float*) – Success probability of the binomial distribution.

- **size** (*int or tuple of ints*) – The shape of the array. If None, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.int32` and `numpy.int64` types are allowed.

**Returns** Samples drawn from the binomial distribution.

**Return type** `cupy.ndarray`

**See also:**

`numpy.random.binomial()`

### `cupy.random.bytes`

`cupy.random.bytes(length)`

Returns random bytes.

---

**Note:** This function is just a wrapper for `numpy.random.bytes`. The resulting bytes are generated on the host (NumPy), not GPU.

---

**See also:**

`numpy.random.bytes`

### `cupy.random.chisquare`

`cupy.random.chisquare(df, size=None, dtype=<class 'float'>)`

Chi-square distribution.

Returns an array of samples drawn from the chi-square distribution. Its probability density function is defined as

$$f(x) = \frac{(1/2)^{k/2}}{\Gamma(k/2)} x^{k/2-1} e^{-x/2}.$$

#### Parameters

- **df** (*int or array\_like of ints*) – Degree of freedom  $k$ .
- **size** (*int or tuple of ints*) – The shape of the array. If None, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

**Returns** Samples drawn from the chi-square distribution.

**Return type** `cupy.ndarray`

**See also:**

`numpy.random.chisquare()`

## cupy.random.choice

`cupy.random.choice(a, size=None, replace=True, p=None)`

Returns an array of random values from a given 1-D array.

Each element of the returned array is independently sampled from `a` according to `p` or uniformly.

---

**Note:** Currently `p` is not supported when `replace=False`.

---

### Parameters

- **a** (*1-D array-like or int*) – If an array-like, a random sample is generated from its elements. If an int, the random sample is generated as if `a` was `cupy.arange(n)`
- **size** (*int or tuple of ints*) – The shape of the array.
- **replace** (*boolean*) – Whether the sample is with or without replacement.
- **p** (*1-D array-like*) – The probabilities associated with each entry in `a`. If not given the sample assumes a uniform distribution over all entries in `a`.

**Returns** An array of a values distributed according to `p` or uniformly.

**Return type** `cupy.ndarray`

See also:

`numpy.random.choice()`

## cupy.random.dirichlet

`cupy.random.dirichlet(alpha, size=None, dtype=<class 'float'>)`

Dirichlet distribution.

Returns an array of samples drawn from the dirichlet distribution. Its probability density function is defined as

$$f(x) = \frac{\Gamma(\sum_{i=1}^K \alpha_i)}{\prod_{i=1}^K \Gamma(\alpha_i)} \prod_{i=1}^K x_i^{\alpha_i-1}.$$

### Parameters

- **alpha** (*array*) – Parameters of the dirichlet distribution  $\alpha$ .
- **size** (*int or tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

**Returns** Samples drawn from the dirichlet distribution.

**Return type** `cupy.ndarray`

See also:

`numpy.random.dirichlet()`



**cupy.random.exponential**

`cupy.random.exponential(scale, size=None, dtype=<class 'float'>)`  
 Exponential distribution.

Returns an array of samples drawn from the exponential distribution. Its probability density function is defined as

$$f(x) = \frac{1}{\beta} \exp\left(-\frac{x}{\beta}\right).$$

**Parameters**

- **scale** (*float or array\_like of floats*) – The scale parameter  $\beta$ .
- **size** (*int or tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

**Returns** Samples drawn from the exponential distribution.

**Return type** *cupy.ndarray*

See also:

`numpy.random.exponential()`

**cupy.random.f**

`cupy.random.f(dfnum, dfden, size=None, dtype=<class 'float'>)`  
 F distribution.

Returns an array of samples drawn from the f distribution. Its probability density function is defined as

$$f(x) = \frac{1}{B\left(\frac{d_1}{2}, \frac{d_2}{2}\right)} \left(\frac{d_1}{d_2}\right)^{\frac{d_1}{2}} x^{\frac{d_1}{2}-1} \left(1 + \frac{d_1}{d_2}x\right)^{-\frac{d_1+d_2}{2}}.$$

**Parameters**

- **dfnum** (*float or array\_like of floats*) – Parameter of the f distribution  $d_1$ .
- **dfden** (*float or array\_like of floats*) – Parameter of the f distribution  $d_2$ .
- **size** (*int or tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

**Returns** Samples drawn from the f distribution.

**Return type** *cupy.ndarray*

See also:

`numpy.random.f()`

## cupy.random.gamma

`cupy.random.gamma(shape, scale=1.0, size=None, dtype=<class 'float'>)`  
Gamma distribution.

Returns an array of samples drawn from the gamma distribution. Its probability density function is defined as

$$f(x) = \frac{1}{\Gamma(k)\theta^k} x^{k-1} e^{-x/\theta}.$$

### Parameters

- **shape** (*array*) – Parameter of the gamma distribution  $k$ .
- **scale** (*array*) – Parameter of the gamma distribution  $\theta$
- **size** (*int or tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

Returns: `cupy.ndarray`: Samples drawn from the gamma distribution.

See also:

`numpy.random.gamma()`

## cupy.random.geometric

`cupy.random.geometric(p, size=None, dtype=<class 'int'>)`  
Geometric distribution.

Returns an array of samples drawn from the geometric distribution. Its probability mass function is defined as

$$f(x) = p(1 - p)^{x-1}.$$

### Parameters

- **p** (*float*) – Success probability of the geometric distribution.
- **size** (*int or tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.int32` and `numpy.int64` types are allowed.

**Returns** Samples drawn from the geometric distribution.

**Return type** `cupy.ndarray`

See also:

`numpy.random.geometric()`

**cupy.random.gumbel**

`cupy.random.gumbel(loc=0.0, scale=1.0, size=None, dtype=<class 'float'>)`

Returns an array of samples drawn from a Gumbel distribution.

The samples are drawn from a Gumbel distribution with location `loc` and scale `scale`. Its probability density function is defined as

$$f(x) = \frac{1}{\eta} \exp \left\{ -\frac{x - \mu}{\eta} \right\} \exp \left[ -\exp \left\{ -\frac{x - \mu}{\eta} \right\} \right],$$

where  $\mu$  is `loc` and  $\eta$  is `scale`.

**Parameters**

- **loc** (*float*) – The location of the mode  $\mu$ .
- **scale** (*float*) – The scale parameter  $\eta$ .
- **size** (*int or tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

**Returns** Samples drawn from the Gumbel distribution.

**Return type** *cupy.ndarray*

**See also:**

`numpy.random.gumbel()`

**cupy.random.hypergeometric**

`cupy.random.hypergeometric(ngood, nbad, nsample, size=None, dtype=<class 'int'>)`

hypergeometric distribution.

Returns an array of samples drawn from the hypergeometric distribution. Its probability mass function is defined as

$$f(x) = \frac{\binom{m}{n} \binom{N-m}{n-x}}{\binom{N}{n}}.$$

**Parameters**

- **ngood** (*int or array\_like of ints*) – Parameter of the hypergeometric distribution  $n$ .
- **nbad** (*int or array\_like of ints*) – Parameter of the hypergeometric distribution  $m$ .
- **nsample** (*int or array\_like of ints*) – Parameter of the hypergeometric distribution  $N$ .
- **size** (*int or tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.int32` and `numpy.int64` types are allowed.

**Returns** Samples drawn from the hypergeometric distribution.

**Return type** *cupy.ndarray*

**See also:**

`numpy.random.hypergeometric()`

### cupy.random.laplace

`cupy.random.laplace(loc=0.0, scale=1.0, size=None, dtype=<class 'float'>)`  
Laplace distribution.

Returns an array of samples drawn from the laplace distribution. Its probability density function is defined as

$$f(x) = \frac{1}{2b} \exp\left(-\frac{|x - \mu|}{b}\right).$$

#### Parameters

- **loc** (*float*) – The location of the mode  $\mu$ .
- **scale** (*float*) – The scale parameter  $b$ .
- **size** (*int* or *tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

**Returns** Samples drawn from the laplace distribution.

**Return type** *cupy.ndarray*

See also:

`numpy.random.laplace()`

### cupy.random.logistic

`cupy.random.logistic(loc=0.0, scale=1.0, size=None, dtype=<class 'float'>)`  
Logistic distribution.

Returns an array of samples drawn from the logistic distribution. Its probability density function is defined as

$$f(x) = \frac{e^{-(x-\mu)/s}}{s(1 + e^{-(x-\mu)/s})^2}.$$

#### Parameters

- **loc** (*float*) – The location of the mode  $\mu$ .
- **scale** (*float*) – The scale parameter  $s$ .
- **size** (*int* or *tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

**Returns** Samples drawn from the logistic distribution.

**Return type** *cupy.ndarray*

See also:

`numpy.random.logistic()`

### cupy.random.lognormal

`cupy.random.lognormal(mean=0.0, sigma=1.0, size=None, dtype=<class 'float'>)`

Returns an array of samples drawn from a log normal distribution.

The samples are natural log of samples drawn from a normal distribution with mean `mean` and deviation `sigma`.

#### Parameters

- **mean** (*float*) – Mean of the normal distribution.
- **sigma** (*float*) – Standard deviation of the normal distribution.
- **size** (*int or tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

**Returns** Samples drawn from the log normal distribution.

**Return type** *cupy.ndarray*

See also:

`numpy.random.lognormal()`

### cupy.random.logseries

`cupy.random.logseries(p, size=None, dtype=<class 'int'>)`

Log series distribution.

Returns an array of samples drawn from the log series distribution. Its probability mass function is defined as

$$f(x) = \frac{-p^x}{x \ln(1-p)}.$$

#### Parameters

- **p** (*float*) – Parameter of the log series distribution  $p$ .
- **size** (*int or tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.int32` and `numpy.int64` types are allowed.

**Returns** Samples drawn from the log series distribution.

**Return type** *cupy.ndarray*

See also:

`numpy.random.logseries()`

### cupy.random.multinomial

`cupy.random.multinomial(n, pvals, size=None)`  
Returns an array from multinomial distribution.

#### Parameters

- **n** (*int*) – Number of trials.
- **pvals** (*cupy.ndarray*) – Probabilities of each of the *p* different outcomes. The sum of these values must be 1.
- **size** (*int* or *tuple of ints* or *None*) – Shape of a sample in each trial. For example when *size* is (*a*, *b*), shape of returned value is (*a*, *b*, *p*) where *p* is `len(pvals)`. If *size* is *None*, it is treated as (). So, shape of returned value is (*p*,).

**Returns** An array drawn from multinomial distribution.

**Return type** *cupy.ndarray*

---

**Note:** It does not support `sum(pvals) < 1` case.

---

See also:

`numpy.random.multinomial()`

### cupy.random.multivariate\_normal

`cupy.random.multivariate_normal(mean, cov, size=None, check_valid='ignore', tol=1e-08, method='cholesky', dtype=<class 'float'>)`

Multivariate normal distribution.

Returns an array of samples drawn from the multivariate normal distribution. Its probability density function is defined as

$$f(x) = \frac{1}{(2\pi|\Sigma|)^{(n/2)}} \exp\left(-\frac{1}{2}(x - \mu)^T \Sigma^{-1}(x - \mu)\right).$$

#### Parameters

- **mean** (*1-D array\_like, of length N*) – Mean of the multivariate normal distribution  $\mu$ .
- **cov** (*2-D array\_like, of shape (N, N)*) – Covariance matrix  $\Sigma$  of the multivariate normal distribution. It must be symmetric and positive-semidefinite for proper sampling.
- **size** (*int* or *tuple of ints*) – The shape of the array. If *None*, a zero-dimensional array is generated.
- **check\_valid** (*'warn', 'raise', 'ignore'*) – Behavior when the covariance matrix is not positive semidefinite.
- **tol** (*float*) – Tolerance when checking the singular values in covariance matrix.
- **method** – { *'cholesky', 'eigh', 'svd'* }, optional The *cov* input is used to compute a factor matrix *A* such that *A* @ *A.T* = *cov*. This argument is used to select the method used to compute the factor matrix *A*. The default method *'cholesky'* is the fastest, while *'svd'* is the slowest but more robust than the fastest method. The method *eigh* uses eigen decomposition to compute *A* and is faster than *svd* but slower than *cholesky*.

- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

**Returns** Samples drawn from the multivariate normal distribution.

**Return type** `cupy.ndarray`

---

**Note:** Default *method* is set to fastest, ‘cholesky’, unlike `numpy` which defaults to ‘svd’. Cholesky decomposition in CuPy will fail silently if the input covariance matrix is not positive definite and give invalid results, unlike in `numpy`, where an invalid covariance matrix will raise an exception. Setting *check\_valid* to ‘raise’ will replicate `numpy` behavior by checking the input, but will also force device synchronization. If validity of input is unknown, setting *method* to ‘einh’ or ‘svd’ and *check\_valid* to ‘warn’ will use cholesky decomposition for positive definite matrices, and fallback to the specified *method* for other matrices (i.e., not positive semi-definite), and will warn if decomposition is suspect.

---

**See also:**

`numpy.random.multivariate_normal()`

### `cupy.random.negative_binomial`

`cupy.random.negative_binomial(n, p, size=None, dtype=<class 'int'>)`

Negative binomial distribution.

Returns an array of samples drawn from the negative binomial distribution. Its probability mass function is defined as

$$f(x) = \binom{x+n-1}{n-1} p^n (1-p)^x.$$

#### Parameters

- **n** (*int*) – Parameter of the negative binomial distribution  $n$ .
- **p** (*float*) – Parameter of the negative binomial distribution  $p$ .
- **size** (*int or tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.int32` and `numpy.int64` types are allowed.

**Returns** Samples drawn from the negative binomial distribution.

**Return type** `cupy.ndarray`

**See also:**

`numpy.random.negative_binomial()`

### `cupy.random.noncentral_chisquare`

`cupy.random.noncentral_chisquare(df, nonc, size=None, dtype=<class 'float'>)`

Noncentral chisquare distribution.

Returns an array of samples drawn from the noncentral chisquare distribution. Its probability density function is defined as

$$f(x) = \frac{1}{2} e^{-(x+\lambda)/2} \left(\frac{x}{\lambda}\right)^{k/4-1/2} I_{k/2-1}(\sqrt{\lambda x}),$$

where  $I$  is the modified Bessel function of the first kind.

**Parameters**

- **df** (*float*) – Parameter of the noncentral chisquare distribution  $k$ .
- **nonc** (*float*) – Parameter of the noncentral chisquare distribution  $\lambda$ .
- **size** (*int or tuple of ints*) – The shape of the array. If *None*, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

**Returns** Samples drawn from the noncentral chisquare distribution.

**Return type** *cupy.ndarray*

**See also:**

`numpy.random.noncentral_chisquare()`

**cupy.random.noncentral\_f**

`cupy.random.noncentral_f(dfnum, dfden, nonc, size=None, dtype=<class 'float'>)`

Noncentral F distribution.

Returns an array of samples drawn from the noncentral F distribution.

Reference: [https://en.wikipedia.org/wiki/Noncentral\\_F-distribution](https://en.wikipedia.org/wiki/Noncentral_F-distribution)

**Parameters**

- **dfnum** (*float*) – Parameter of the noncentral F distribution.
- **dfden** (*float*) – Parameter of the noncentral F distribution.
- **nonc** (*float*) – Parameter of the noncentral F distribution.
- **size** (*int or tuple of ints*) – The shape of the array. If *None*, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

**Returns** Samples drawn from the noncentral F distribution.

**Return type** *cupy.ndarray*

**See also:**

`numpy.random.noncentral_f()`

**cupy.random.normal**

`cupy.random.normal(loc=0.0, scale=1.0, size=None, dtype=<class 'float'>)`

Returns an array of normally distributed samples.

**Parameters**

- **loc** (*float or array\_like of floats*) – Mean of the normal distribution.
- **scale** (*float or array\_like of floats*) – Standard deviation of the normal distribution.
- **size** (*int or tuple of ints*) – The shape of the array. If *None*, a zero-dimensional array is generated.



- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

**Returns** Normally distributed samples.

**Return type** `cupy.ndarray`

**See also:**

`numpy.random.normal()`

### `cupy.random.pareto`

`cupy.random.pareto(a, size=None, dtype=<class 'float'>)`

Pareto II or Lomax distribution.

Returns an array of samples drawn from the Pareto II distribution. Its probability density function is defined as

$$f(x) = \alpha(1+x)^{-(\alpha+1)}.$$

#### Parameters

- **a** (*float or array\_like of floats*) – Parameter of the Pareto II distribution  $\alpha$ .
- **size** (*int or tuple of ints*) – The shape of the array. If `None`, this function generate an array whose shape is `a.shape`.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

**Returns** Samples drawn from the Pareto II distribution.

**Return type** `cupy.ndarray`

**See also:**

`numpy.random.pareto()`

### `cupy.random.permutation`

`cupy.random.permutation(a)`

Returns a permuted range or a permutation of an array.

**Parameters** **a** (*int or cupy.ndarray*) – The range or the array to be shuffled.

**Returns** If *a* is an integer, it is permutation range between 0 and *a* - 1. Otherwise, it is a permutation of *a*.

**Return type** `cupy.ndarray`

**See also:**

`numpy.random.permutation()`

## cupy.random.poisson

`cupy.random.poisson(lam=1.0, size=None, dtype=<class 'int'>)`  
Poisson distribution.

Returns an array of samples drawn from the poisson distribution. Its probability mass function is defined as

$$f(x) = \frac{\lambda^x e^{-\lambda}}{x!}.$$

### Parameters

- **lam** (*array\_like of floats*) – Parameter of the poisson distribution  $\lambda$ .
- **size** (*int or tuple of ints*) – The shape of the array. If None, this function generate an array whose shape is *lam.shape*.
- **dtype** – Data type specifier. Only `numpy.int32` and `numpy.int64` types are allowed.

**Returns** Samples drawn from the poisson distribution.

**Return type** *cupy.ndarray*

See also:

`numpy.random.poisson()`

## cupy.random.power

`cupy.random.power(a, size=None, dtype=<class 'float'>)`  
Power distribution.

Returns an array of samples drawn from the power distribution. Its probability density function is defined as

$$f(x) = ax^{a-1}.$$

### Parameters

- **a** (*float*) – Parameter of the power distribution  $a$ .
- **size** (*int or tuple of ints*) – The shape of the array. If None, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

**Returns** Samples drawn from the power distribution.

**Return type** *cupy.ndarray*

See also:

`numpy.random.power()`

## cupy.random.rand

`cupy.random.rand(*size, **kwarg)`

Returns an array of uniform random values over the interval  $[0, 1)$ .

Each element of the array is uniformly distributed on the half-open interval  $[0, 1)$ . All elements are identically and independently distributed (i.i.d.).

### Parameters

- **size** (*ints*) – The shape of the array.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed. The default is `numpy.float64`.

**Returns** A random array.

**Return type** `cupy.ndarray`

See also:

`numpy.random.rand()`

### Example

```
>>> cupy.random.rand(3, 2)
array([[0.86476479, 0.05633727], # random
       [0.27283185, 0.38255354], # random
       [0.16592278, 0.75150313]]) # random

>>> cupy.random.rand(3, 2, dtype=cupy.float32)
array([[0.9672306 , 0.9590486 ], # random
       [0.6851264 , 0.70457625], # random
       [0.22382522, 0.36055237]], dtype=float32) # random
```

## cupy.random.randint

`cupy.random.randint(low, high=None, size=None, dtype='l')`

Returns a scalar or an array of integer values over  $[low, high)$ .

Each element of returned values are independently sampled from uniform distribution over left-close and right-open interval  $[low, high)$ .

### Parameters

- **low** (*int*) – If **high** is not `None`, it is the lower bound of the interval. Otherwise, it is the **upper** bound of the interval and lower bound of the interval is set to `0`.
- **high** (*int*) – Upper bound of the interval.
- **size** (*None or int or tuple of ints*) – The shape of returned value.
- **dtype** – Data type specifier.

**Returns** If **size** is `None`, it is single integer sampled. If **size** is integer, it is the 1D-array of length **size** element. Otherwise, it is the array whose shape specified by **size**.

**Return type** `int` or `cupy.ndarray` of ints

## cupy.random.randn

cupy.random.randn(\*size, \*\*kwarg)

Returns an array of standard normal random values.

Each element of the array is normally distributed with zero mean and unit variance. All elements are identically and independently distributed (i.i.d.).

### Parameters

- **size** (*ints*) – The shape of the array.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed. The default is `numpy.float64`.

**Returns** An array of standard normal random values.

**Return type** `cupy.ndarray`

See also:

`numpy.random.randn()`

---

### Example

```
>>> cupy.random.randn(3, 2)
array([[0.41193321, 1.59579542],  # random
       [0.47904589, 0.18566376],  # random
       [0.59748424, 2.32602829]]) # random

>>> cupy.random.randn(3, 2, dtype=cupy.float32)
array([[ 0.1373886 ,  2.403238  ],  # random
       [ 0.84020025,  1.5089266 ],  # random
       [-1.2268474 , -0.48219103]], dtype=float32) # random
```

---

## cupy.random.random

cupy.random.random(size=None, dtype=<class 'float'>)

Returns an array of random values over the interval `[0, 1)`.

This is a variant of `cupy.random.rand()`.

### Parameters

- **size** (*int* or *tuple of ints*) – The shape of the array.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

**Returns** An array of uniformly distributed random values.

**Return type** `cupy.ndarray`

See also:

`numpy.random.random_sample()`

## cupy.random.random\_integers

`cupy.random.random_integers(low, high=None, size=None)`

Return a scalar or an array of integer values over `[low, high]`

Each element of returned values are independently sampled from uniform distribution over closed interval `[low, high]`.

### Parameters

- **low** (*int*) – If `high` is not `None`, it is the lower bound of the interval. Otherwise, it is the **upper** bound of the interval and the lower bound is set to 1.
- **high** (*int*) – Upper bound of the interval.
- **size** (*None or int or tuple of ints*) – The shape of returned value.

**Returns** If `size` is `None`, it is single integer sampled. If `size` is integer, it is the 1D-array of length `size` element. Otherwise, it is the array whose shape specified by `size`.

**Return type** *int* or `cupy.ndarray` of *ints*

## cupy.random.random\_sample

`cupy.random.random_sample(size=None, dtype=<class 'float'>)`

Returns an array of random values over the interval `[0, 1)`.

This is a variant of `cupy.random.rand()`.

### Parameters

- **size** (*int or tuple of ints*) – The shape of the array.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

**Returns** An array of uniformly distributed random values.

**Return type** `cupy.ndarray`

See also:

`numpy.random.random_sample()`

## cupy.random.randf

`cupy.random.randf(size=None, dtype=<class 'float'>)`

Returns an array of random values over the interval `[0, 1)`.

This is a variant of `cupy.random.rand()`.

### Parameters

- **size** (*int or tuple of ints*) – The shape of the array.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

**Returns** An array of uniformly distributed random values.

**Return type** `cupy.ndarray`

See also:

`numpy.random.random_sample()`

## cupy.random.rayleigh

`cupy.random.rayleigh(scale=1.0, size=None, dtype=<class 'float'>)`  
Rayleigh distribution.

Returns an array of samples drawn from the rayleigh distribution. Its probability density function is defined as

$$f(x) = \frac{x}{\sigma^2} e^{\frac{-x^2}{2\sigma^2}}, x \geq 0.$$

### Parameters

- **scale** (*array*) – Parameter of the rayleigh distribution  $\sigma$ .
- **size** (*int or tuple of ints*) – The shape of the array. If *None*, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

**Returns** Samples drawn from the rayleigh distribution.

**Return type** *cupy.ndarray*

See also:

`numpy.random.rayleigh()`

## cupy.random.sample

`cupy.random.sample(size=None, dtype=<class 'float'>)`  
Returns an array of random values over the interval `[0, 1)`.

This is a variant of `cupy.random.rand()`.

### Parameters

- **size** (*int or tuple of ints*) – The shape of the array.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

**Returns** An array of uniformly distributed random values.

**Return type** *cupy.ndarray*

See also:

`numpy.random.random_sample()`

## cupy.random.seed

`cupy.random.seed(seed=None)`  
Resets the state of the random number generator with a seed.

This function resets the state of the global random number generator for the current device. Be careful that generators for other devices are not affected.

**Parameters** **seed** (*None or int*) – Seed for the random number generator. If *None*, it uses `os.urandom()` if available or `time.time()` otherwise. Note that this function does not support seeding by an integer array.

### cupy.random.shuffle

`cupy.random.shuffle(a)`

Shuffles an array.

**Parameters** `a` (`cupy.ndarray`) – The array to be shuffled.

**See also:**

`numpy.random.shuffle()`

### cupy.random.standard\_cauchy

`cupy.random.standard_cauchy(size=None, dtype=<class 'float'>)`

Standard cauchy distribution.

Returns an array of samples drawn from the standard cauchy distribution. Its probability density function is defined as

$$f(x) = \frac{1}{\pi(1+x^2)}.$$

**Parameters**

- **size** (`int` or *tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

**Returns** Samples drawn from the standard cauchy distribution.

**Return type** `cupy.ndarray`

**See also:**

`numpy.random.standard_cauchy()`

### cupy.random.standard\_exponential

`cupy.random.standard_exponential(size=None, dtype=<class 'float'>)`

Standard exponential distribution.

Returns an array of samples drawn from the standard exponential distribution. Its probability density function is defined as

$$f(x) = e^{-x}.$$

**Parameters**

- **size** (`int` or *tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

**Returns** Samples drawn from the standard exponential distribution.

**Return type** *cupy.ndarray*

**See also:**

`numpy.random.standard_exponential()`

### `cupy.random.standard_gamma`

`cupy.random.standard_gamma(shape, size=None, dtype=<class 'float'>)`

Standard gamma distribution.

Returns an array of samples drawn from the standard gamma distribution. Its probability density function is defined as

$$f(x) = \frac{1}{\Gamma(k)} x^{k-1} e^{-x}.$$

#### Parameters

- **shape** (*array*) – Parameter of the gamma distribution  $k$ .
- **size** (*int or tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

**Returns** Samples drawn from the standard gamma distribution.

**Return type** *cupy.ndarray*

**See also:**

`numpy.random.standard_gamma()`

### `cupy.random.standard_normal`

`cupy.random.standard_normal(size=None, dtype=<class 'float'>)`

Returns an array of samples drawn from the standard normal distribution.

This is a variant of `cupy.random.randn()`.

#### Parameters

- **size** (*int or tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier.

**Returns** Samples drawn from the standard normal distribution.

**Return type** *cupy.ndarray*

**See also:**

`numpy.random.standard_normal()`



### cupy.random.standard\_t

`cupy.random.standard_t(df, size=None, dtype=<class 'float'>)`

Standard Student's t distribution.

Returns an array of samples drawn from the standard Student's t distribution. Its probability density function is defined as

$$f(x) = \frac{\Gamma(\frac{\nu+1}{2})}{\sqrt{\nu\pi}\Gamma(\frac{\nu}{2})} \left(1 + \frac{x^2}{\nu}\right)^{-\left(\frac{\nu+1}{2}\right)}.$$

#### Parameters

- **df** (*float* or *array\_like of floats*) – Degree of freedom  $\nu$ .
- **size** (*int* or *tuple of ints*) – The shape of the array. If *None*, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

**Returns** Samples drawn from the standard Student's t distribution.

**Return type** *cupy.ndarray*

See also:

`numpy.random.standard_t()`

### cupy.random.triangular

`cupy.random.triangular(left, mode, right, size=None, dtype=<class 'float'>)`

Triangular distribution.

Returns an array of samples drawn from the triangular distribution. Its probability density function is defined as

$$f(x) = \begin{cases} \frac{2(x-l)}{(r-l)(m-l)} & \text{for } l \leq x \leq m, \\ \frac{2(r-x)}{(r-l)(r-m)} & \text{for } m \leq x \leq r, \\ 0 & \text{otherwise.} \end{cases}$$

#### Parameters

- **left** (*float*) – Lower limit  $l$ .
- **mode** (*float*) – The value where the peak of the distribution occurs.  $m$ .
- **right** (*float*) – Higher Limit  $r$ .
- **size** (*int* or *tuple of ints*) – The shape of the array. If *None*, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

**Returns** Samples drawn from the triangular distribution.

**Return type** *cupy.ndarray*

See also:

`numpy.random.triangular()`

### cupy.random.uniform

`cupy.random.uniform(low=0.0, high=1.0, size=None, dtype=<class 'float'>)`

Returns an array of uniformly-distributed samples over an interval.

Samples are drawn from a uniform distribution over the half-open interval `[low, high)`. The samples may contain the `high` limit due to floating-point rounding.

#### Parameters

- **low** (*float*) – Lower end of the interval.
- **high** (*float*) – Upper end of the interval.
- **size** (*int or tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier.

**Returns** Samples drawn from the uniform distribution.

**Return type** *cupy.ndarray*

See also:

`numpy.random.uniform()`

### cupy.random.vonmises

`cupy.random.vonmises(mu, kappa, size=None, dtype=<class 'float'>)`

von Mises distribution.

Returns an array of samples drawn from the von Mises distribution. Its probability density function is defined as

$$f(x) = \frac{e^{\kappa \cos(x-\mu)}}{2\pi I_0(\kappa)}.$$

#### Parameters

- **mu** (*float*) – Parameter of the von Mises distribution  $\mu$ .
- **kappa** (*float*) – Parameter of the von Mises distribution  $\kappa$ .
- **size** (*int or tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

**Returns** Samples drawn from the von Mises distribution.

**Return type** *cupy.ndarray*

See also:

`numpy.random.vonmises()`

**cupy.random.wald**

`cupy.random.wald(mean, scale, size=None, dtype=<class 'float'>)`  
 Wald distribution.

Returns an array of samples drawn from the Wald distribution. Its probability density function is defined as

$$f(x) = \sqrt{\frac{\lambda}{2\pi x^3}} e^{-\frac{\lambda(x-\mu)^2}{2\mu^2 x}}.$$

**Parameters**

- **mean** (*float*) – Parameter of the wald distribution  $\mu$ .
- **scale** (*float*) – Parameter of the wald distribution  $\lambda$ .
- **size** (*int or tuple of ints*) – The shape of the array. If *None*, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

**Returns** Samples drawn from the wald distribution.

**Return type** *cupy.ndarray*

**See also:**

`numpy.random.wald()`

**cupy.random.weibull**

`cupy.random.weibull(a, size=None, dtype=<class 'float'>)`  
 weibull distribution.

Returns an array of samples drawn from the weibull distribution. Its probability density function is defined as

$$f(x) = ax^{(a-1)}e^{-x^a}.$$

**Parameters**

- **a** (*float*) – Parameter of the weibull distribution  $a$ .
- **size** (*int or tuple of ints*) – The shape of the array. If *None*, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

**Returns** Samples drawn from the weibull distribution.

**Return type** *cupy.ndarray*

**See also:**

`numpy.random.weibull()`

## cupy.random.zipf

`cupy.random.zipf(a, size=None, dtype=<class 'int'>)`  
Zipf distribution.

Returns an array of samples drawn from the Zipf distribution. Its probability mass function is defined as

$$f(x) = \frac{x^{-a}}{\zeta(a)},$$

where  $\zeta$  is the Riemann Zeta function.

### Parameters

- **a** (*float*) – Parameter of the beta distribution  $a$ .
- **size** (*int or tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.int32` and `numpy.int64` types are allowed.

**Returns** Samples drawn from the Zipf distribution.

**Return type** *cupy.ndarray*

**See also:**

*numpy.random.zipf()*

CuPy does not provide `cupy.random.get_state` nor `cupy.random.set_state` at this time. Use the following CuPy-specific APIs instead. Note that these functions use *cupy.random.RandomState* instance to represent the internal state, which cannot be serialized.

<i>get_random_state()</i>	Gets the state of the random number generator for the current device.
<i>set_random_state(rs)</i>	Sets the state of the random number generator for the current device.

## cupy.random.get\_random\_state

`cupy.random.get_random_state()`

Gets the state of the random number generator for the current device.

If the state for the current device is not created yet, this function creates a new one, initializes it, and stores it as the state for the current device.

**Returns** The state of the random number generator for the device.

**Return type** *RandomState*

**cupy.random.set\_random\_state**

`cupy.random.set_random_state(rs)`

Sets the state of the random number generator for the current device.

**Parameters** `state` ([RandomState](#)) – Random state to set for the current device.

**5.3.16 Set routines**


---

**Hint:** [NumPy API Reference: Set routines](#)

---

**Making proper sets**


---

<code>unique(ar[, return_index, return_inverse, ...])</code>	Find the unique elements of an array.
--	---------------------------------------

---

**Boolean operations**


---

<code>in1d(ar1, ar2[, assume_unique, invert])</code>	Tests whether each element of a 1-D array is also present in a second array.
<code>isin(element, test_elements[, ...])</code>	Calculates element in <code>test_elements</code> , broadcasting over <code>element</code> only.

---

**cupy.in1d**

`cupy.in1d(ar1, ar2, assume_unique=False, invert=False)`

Tests whether each element of a 1-D array is also present in a second array.

Returns a boolean array the same length as `ar1` that is `True` where an element of `ar1` is in `ar2` and `False` otherwise.

**Parameters**

- `ar1` ([cupy.ndarray](#)) – Input array.
- `ar2` ([cupy.ndarray](#)) – The values against which to test each value of `ar1`.
- `assume_unique` (*bool, optional*) – Ignored
- `invert` (*bool, optional*) – If `True`, the values in the returned array are inverted (that is, `False` where an element of `ar1` is in `ar2` and `True` otherwise). Default is `False`.

**Returns** The values `ar1[in1d]` are in `ar2`.

**Return type** [cupy.ndarray](#), `bool`

## cupy.isin

`cupy.isin(element, test_elements, assume_unique=False, invert=False)`

Calculates element in `test_elements`, broadcasting over `element` only. Returns a boolean array of the same shape as `element` that is `True` where an element of `element` is in `test_elements` and `False` otherwise.

### Parameters

- **element** (`cupy.ndarray`) – Input array.
- **test\_elements** (`cupy.ndarray`) – The values against which to test each value of `element`. This argument is flattened if it is an array or array\_like.
- **assume\_unique** (*bool, optional*) – Ignored
- **invert** (*bool, optional*) – If `True`, the values in the returned array are inverted, as if calculating element not in `test_elements`. Default is `False`.

**Returns** Has the same shape as `element`. The values `element[isin]` are in `test_elements`.

**Return type** `cupy.ndarray, bool`

## 5.3.17 Sorting, searching, and counting

---

**Hint:** [NumPy API Reference: Sorting, searching, and counting](#)

---

### Sorting

<code>sort(a[, axis])</code>	Returns a sorted copy of an array with a stable sorting algorithm.
<code>lexsort(keys)</code>	Perform an indirect sort using an array of keys.
<code>argsort(a[, axis])</code>	Returns the indices that would sort an array with a stable sorting.
<code>msort(a)</code>	Returns a copy of an array sorted along the first axis.
<code>sort_complex(a)</code>	Sort a complex array using the real part first, then the imaginary part.
<code>partition(a, kth[, axis])</code>	Returns a partitioned copy of an array.
<code>argpartition(a, kth[, axis])</code>	Returns the indices that would partially sort an array.

## cupy.sort

`cupy.sort(a, axis=-1)`

Returns a sorted copy of an array with a stable sorting algorithm.

### Parameters

- **a** (`cupy.ndarray`) – Array to be sorted.
- **axis** (*int or None*) – Axis along which to sort. Default is `-1`, which means sort along the last axis. If `None` is supplied, the array is flattened before sorting.

**Returns** Array of the same type and shape as `a`.

**Return type** `cupy.ndarray`

---

**Note:** For its implementation reason, `cupy.sort` currently does not support `kind` and `order` parameters that `numpy.sort` does support.

---

**See also:**

`numpy.sort()`

## `cupy.lexsort`

`cupy.lexsort(keys)`

Perform an indirect sort using an array of keys.

**Parameters** **keys** (`cupy.ndarray`) – (`k`, `N`) array containing `k` (`N`,)-shaped arrays. The `k` different “rows” to be sorted. The last row is the primary sort key.

**Returns** Array of indices that sort the keys.

**Return type** `cupy.ndarray`

---

**Note:** For its implementation reason, `cupy.lexsort` currently supports only keys with their rank of one or two and does not support `axis` parameter that `numpy.lexsort` supports.

---

**See also:**

`numpy.lexsort()`

## `cupy.argsort`

`cupy.argsort(a, axis=-1)`

Returns the indices that would sort an array with a stable sorting.

**Parameters**

- **a** (`cupy.ndarray`) – Array to sort.
- **axis** (`int` or `None`) – Axis along which to sort. Default is -1, which means sort along the last axis. If `None` is supplied, the array is flattened before sorting.

**Returns** Array of indices that sort `a`.

**Return type** `cupy.ndarray`

---

**Note:** For its implementation reason, `cupy.argsort` does not support `kind` and `order` parameters.

---

**See also:**

`numpy.argsort()`

## cupy.msort

`cupy.msort(a)`

Returns a copy of an array sorted along the first axis.

**Parameters** `a` (`cupy.ndarray`) – Array to be sorted.

**Returns** Array of the same type and shape as `a`.

**Return type** `cupy.ndarray`

See also:

`numpy.msort()`

## cupy.sort\_complex

`cupy.sort_complex(a)`

Sort a complex array using the real part first, then the imaginary part.

**Parameters** `a` (`cupy.ndarray`) – Array to be sorted.

**Returns** sorted complex array.

**Return type** `cupy.ndarray`

See also:

`numpy.sort_complex()`

## cupy.partition

`cupy.partition(a, kth, axis=-1)`

Returns a partitioned copy of an array.

Creates a copy of the array whose elements are rearranged such that the value of the element in `k`-th position would occur in that position in a sorted array. All of the elements before the new `k`-th element are less than or equal to the elements after the new `k`-th element.

### Parameters

- `a` (`cupy.ndarray`) – Array to be sorted.
- `kth` (`int` or *sequence of ints*) – Element index to partition by. If supplied with a sequence of `k`-th it will partition all elements indexed by `k`-th of them into their sorted position at once.
- `axis` (`int` or `None`) – Axis along which to sort. Default is `-1`, which means sort along the last axis. If `None` is supplied, the array is flattened before sorting.

**Returns** Array of the same type and shape as `a`.

**Return type** `cupy.ndarray`

See also:

`numpy.partition()`



## cupy.argmaxpartition

`cupy.argmaxpartition(a, kth, axis=-1)`

Returns the indices that would partially sort an array.

### Parameters

- **a** (`cupy.ndarray`) – Array to be sorted.
- **kth** (`int` or *sequence of ints*) – Element index to partition by. If supplied with a sequence of k-th it will partition all elements indexed by k-th of them into their sorted position at once.
- **axis** (`int` or `None`) – Axis along which to sort. Default is -1, which means sort along the last axis. If `None` is supplied, the array is flattened before sorting.

**Returns** Array of the same type and shape as a.

**Return type** `cupy.ndarray`

---

**Note:** For its implementation reason, `cupy.argmaxpartition` fully sorts the given array as `cupy.argsort` does. It also does not support `kind` and `order` parameters that `numpy.argmaxpartition` supports.

---

See also:

`numpy.argmaxpartition()`

See also:

`cupy.ndarray.sort()`

## Searching

<code>argmax(a[, axis, dtype, out, keepdims])</code>	Returns the indices of the maximum along an axis.
<code>nanargmax(a[, axis, dtype, out, keepdims])</code>	Return the indices of the maximum values in the specified axis ignoring NaNs.
<code>argmin(a[, axis, dtype, out, keepdims])</code>	Returns the indices of the minimum along an axis.
<code>nanargmin(a[, axis, dtype, out, keepdims])</code>	Return the indices of the minimum values in the specified axis ignoring NaNs.
<code>argwhere(a)</code>	Return the indices of the elements that are non-zero.
<code>nonzero(a)</code>	Return the indices of the elements that are non-zero.
<code>flatnonzero(a)</code>	Return indices that are non-zero in the flattened version of a.
<code>where(condition[, x, y])</code>	Return elements, either from x or y, depending on condition.
<code>searchsorted(a, v[, side, sorter])</code>	Finds indices where elements should be inserted to maintain order.
<code>extract(condition, a)</code>	Return the elements of an array that satisfy some condition.

## cupy.argmax

`cupy.argmax(a, axis=None, dtype=None, out=None, keepdims=False)`

Returns the indices of the maximum along an axis.

### Parameters

- **a** (`cupy.ndarray`) – Array to take argmax.
- **axis** (`int`) – Along which axis to find the maximum. `a` is flattened by default.
- **dtype** – Data type specifier.
- **out** (`cupy.ndarray`) – Output array.
- **keepdims** (`bool`) – If `True`, the axis `axis` is preserved as an axis of length one.

**Returns** The indices of the maximum of `a` along an axis.

**Return type** `cupy.ndarray`

---

**Note:** `dtype` and `keepdim` arguments are specific to CuPy. They are not in NumPy.

---

---

**Note:** `axis` argument accepts a tuple of ints, but this is specific to CuPy. NumPy does not support it.

---

**See also:**

`numpy.argmax()`

## cupy.nanargmax

`cupy.nanargmax(a, axis=None, dtype=None, out=None, keepdims=False)`

Return the indices of the maximum values in the specified axis ignoring NaNs. For all-NaN slice -1 is returned. Subclass cannot be passed yet, `subok=True` still unsupported

### Parameters

- **a** (`cupy.ndarray`) – Array to take nanargmax.
- **axis** (`int`) – Along which axis to find the maximum. `a` is flattened by default.

**Returns** The indices of the maximum of `a` along an axis ignoring NaN values.

**Return type** `cupy.ndarray`

---

**Note:** For performance reasons, `cupy.nanargmax` returns `out of range` values for all-NaN slice whereas `numpy.nanargmax` raises `ValueError`

---

**See also:**

`numpy.nanargmax()`

## cupy.argmaxin

`cupy.argmaxin(a, axis=None, dtype=None, out=None, keepdims=False)`

Returns the indices of the minimum along an axis.

### Parameters

- **a** (`cupy.ndarray`) – Array to take argmin.
- **axis** (`int`) – Along which axis to find the minimum. `a` is flattened by default.
- **dtype** – Data type specifier.
- **out** (`cupy.ndarray`) – Output array.
- **keepdims** (`bool`) – If `True`, the axis `axis` is preserved as an axis of length one.

**Returns** The indices of the minimum of `a` along an axis.

**Return type** `cupy.ndarray`

---

**Note:** `dtype` and `keepdim` arguments are specific to CuPy. They are not in NumPy.

---



---

**Note:** `axis` argument accepts a tuple of ints, but this is specific to CuPy. NumPy does not support it.

---

**See also:**

`numpy.argmaxin()`

## cupy.nanargmin

`cupy.nanargmin(a, axis=None, dtype=None, out=None, keepdims=False)`

Return the indices of the minimum values in the specified axis ignoring NaNs. For all-NaN slice -1 is returned. Subclass cannot be passed yet, `subok=True` still unsupported

### Parameters

- **a** (`cupy.ndarray`) – Array to take nanargmin.
- **axis** (`int`) – Along which axis to find the minimum. `a` is flattened by default.

**Returns** The indices of the minimum of `a` along an axis ignoring NaN values.

**Return type** `cupy.ndarray`

---

**Note:** For performance reasons, `cupy.nanargmin` returns `out of range` values for all-NaN slice whereas `numpy.nanargmin` raises `ValueError`

---

**See also:**

`numpy.nanargmin()`

## cupy.argmaxwhere

`cupy.argmaxwhere(a)`

Return the indices of the elements that are non-zero.

Returns a  $(N, ndim)$  dimensional array containing the indices of the non-zero elements. Where  $N$  is number of non-zero elements and  $ndim$  is dimension of the given array.

**Parameters** `a` (`cupy.ndarray`) – array

**Returns** Indices of elements that are non-zero.

**Return type** `cupy.ndarray`

**See also:**

`numpy.argmaxwhere()`

## cupy.flatnonzero

`cupy.flatnonzero(a)`

Return indices that are non-zero in the flattened version of `a`.

This is equivalent to `a.ravel().nonzero()[0]`.

**Parameters** `a` (`cupy.ndarray`) – input array

**Returns** Output array, containing the indices of the elements of `a.ravel()` that are non-zero.

**Return type** `cupy.ndarray`

**Warning:** This function may synchronize the device.

**See also:**

`numpy.flatnonzero()`

## cupy.searchsorted

`cupy.searchsorted(a, v, side='left', sorter=None)`

Finds indices where elements should be inserted to maintain order.

Find the indices into a sorted array `a` such that, if the corresponding elements in `v` were inserted before the indices, the order of `a` would be preserved.

**Parameters**

- `a` (`cupy.ndarray`) – Input array. If `sorter` is `None`, then it must be sorted in ascending order, otherwise `sorter` must be an array of indices that sort it.
- `v` (`cupy.ndarray`) – Values to insert into `a`.
- `side` – { 'left', 'right' } If `left`, return the index of the first suitable location found. If `right`, return the last such index. If there is no suitable index, return either 0 or length of `a`.
- `sorter` – 1-D array\_like Optional array of integer indices that sort array `a` into ascending order. They are typically the result of `argsort()`.

**Returns** Array of insertion points with the same shape as `v`.

**Return type** `cupy.ndarray`

---

**Note:** When `a` is not in ascending order, behavior is undefined.

---

**See also:**

`numpy.searchsorted()`

## cupy.extract

`cupy.extract(condition, a)`

Return the elements of an array that satisfy some condition.

This is equivalent to `np.compress(ravel(condition), ravel(arr))`. If `condition` is boolean, `np.extract` is equivalent to `arr[condition]`.

### Parameters

- **condition** (*int or array\_like*) – An array whose nonzero or True entries indicate the elements of array to extract.
- **a** (`cupy.ndarray`) – Input array of the same size as `condition`.

**Returns** Rank 1 array of values from `arr` where `condition` is True.

**Return type** `cupy.ndarray`

**Warning:** This function may synchronize the device.

**See also:**

`numpy.extract()`

## Counting

---

`count_nonzero(a[, axis])`

Counts the number of non-zero values in the array.

---

## cupy.count\_nonzero

`cupy.count_nonzero(a, axis=None)`

Counts the number of non-zero values in the array.

---

**Note:** `numpy.count_nonzero()` returns *int* value when `axis=None`, but `cupy.count_nonzero()` returns zero-dimensional array to reduce CPU-GPU synchronization.

---

### Parameters

- **a** (`cupy.ndarray`) – The array for which to count non-zeros.
- **axis** (*int or tuple, optional*) – Axis or tuple of axes along which to count non-zeros. Default is None, meaning that non-zeros will be counted along a flattened version of `a`.

**Returns** Number of non-zero values in the array along a given axis. Otherwise, the total number of non-zero values in the array is returned.

**Return type** `cupy.ndarray` of `int`

### 5.3.18 Statistics

---

**Hint:** [NumPy API Reference: Statistics](#)

---

#### Order statistics

<code>amin(a[, axis, out, keepdims])</code>	Returns the minimum of an array or the minimum along an axis.
<code>amax(a[, axis, out, keepdims])</code>	Returns the maximum of an array or the maximum along an axis.
<code>nanmin(a[, axis, out, keepdims])</code>	Returns the minimum of an array along an axis ignoring NaN.
<code>nanmax(a[, axis, out, keepdims])</code>	Returns the maximum of an array along an axis ignoring NaN.
<code>ptp(a[, axis, out, keepdims])</code>	Returns the range of values (maximum - minimum) along an axis.
<code>percentile(a, q[, axis, out, interpolation, ...])</code>	Computes the q-th percentile of the data along the specified axis.
<code>quantile(a, q[, axis, out, interpolation, ...])</code>	Computes the q-th quantile of the data along the specified axis.

#### `cupy.amin`

`cupy.amin(a, axis=None, out=None, keepdims=False)`  
Returns the minimum of an array or the minimum along an axis.

---

**Note:** When at least one element is NaN, the corresponding min value will be NaN.

---

##### Parameters

- **a** (`cupy.ndarray`) – Array to take the minimum.
- **axis** (`int`) – Along which axis to take the minimum. The flattened array is used by default.
- **out** (`cupy.ndarray`) – Output array.
- **keepdims** (`bool`) – If `True`, the axis is remained as an axis of size one.

**Returns** The minimum of `a`, along the axis if specified.

**Return type** `cupy.ndarray`

---

**Note:** When cuTENSOR accelerator is used, the output value might be collapsed for reduction axes that have

one or more NaN elements.

See also:

`numpy.amin()`

## cupy.amax

`cupy.amax(a, axis=None, out=None, keepdims=False)`

Returns the maximum of an array or the maximum along an axis.

---

**Note:** When at least one element is NaN, the corresponding min value will be NaN.

---

### Parameters

- **a** (`cupy.ndarray`) – Array to take the maximum.
- **axis** (`int`) – Along which axis to take the maximum. The flattened array is used by default.
- **out** (`cupy.ndarray`) – Output array.
- **keepdims** (`bool`) – If `True`, the axis is remained as an axis of size one.

**Returns** The maximum of `a`, along the axis if specified.

**Return type** `cupy.ndarray`

---

**Note:** When cuTENSOR accelerator is used, the output value might be collapsed for reduction axes that have one or more NaN elements.

---

See also:

`numpy.amax()`

## cupy.nanmin

`cupy.nanmin(a, axis=None, out=None, keepdims=False)`

Returns the minimum of an array along an axis ignoring NaN.

When there is a slice whose elements are all NaN, a `RuntimeWarning` is raised and NaN is returned.

### Parameters

- **a** (`cupy.ndarray`) – Array to take the minimum.
- **axis** (`int`) – Along which axis to take the minimum. The flattened array is used by default.
- **out** (`cupy.ndarray`) – Output array.
- **keepdims** (`bool`) – If `True`, the axis is remained as an axis of size one.

**Returns** The minimum of `a`, along the axis if specified.

**Return type** `cupy.ndarray`

**Warning:** This function may synchronize the device.

See also:

`numpy.nanmin()`

## `cupy.nanmax`

`cupy.nanmax(a, axis=None, out=None, keepdims=False)`

Returns the maximum of an array along an axis ignoring NaN.

When there is a slice whose elements are all NaN, a `RuntimeWarning` is raised and NaN is returned.

### Parameters

- **a** (`cupy.ndarray`) – Array to take the maximum.
- **axis** (`int`) – Along which axis to take the maximum. The flattened array is used by default.
- **out** (`cupy.ndarray`) – Output array.
- **keepdims** (`bool`) – If True, the axis is remained as an axis of size one.

**Returns** The maximum of a, along the axis if specified.

**Return type** `cupy.ndarray`

**Warning:** This function may synchronize the device.

See also:

`numpy.nanmax()`

## `cupy.ptp`

`cupy.ptp(a, axis=None, out=None, keepdims=False)`

Returns the range of values (maximum - minimum) along an axis.

---

**Note:** The name of the function comes from the acronym for ‘peak to peak’.

When at least one element is NaN, the corresponding ptp value will be NaN.

---

### Parameters

- **a** (`cupy.ndarray`) – Array over which to take the range.
- **axis** (`int`) – Axis along which to take the minimum. The flattened array is used by default.
- **out** (`cupy.ndarray`) – Output array.
- **keepdims** (`bool`) – If True, the axis is retained as an axis of size one.

**Returns** The minimum of a, along the axis if specified.

**Return type** `cupy.ndarray`



---

**Note:** When cuTENSOR accelerator is used, the output value might be collapsed for reduction axes that have one or more NaN elements.

---

See also:

`numpy.amin()`

## cupy.percentile

`cupy.percentile(a, q, axis=None, out=None, interpolation='linear', keepdims=False)`

Computes the q-th percentile of the data along the specified axis.

### Parameters

- **a** (`cupy.ndarray`) – Array for which to compute percentiles.
- **q** (`float`, *tuple of floats* or `cupy.ndarray`) – Percentiles to compute in the range between 0 and 100 inclusive.
- **axis** (`int` or *tuple of ints*) – Along which axis or axes to compute the percentiles. The flattened array is used by default.
- **out** (`cupy.ndarray`) – Output array.
- **interpolation** (`str`) – Interpolation method when a quantile lies between two data points. linear interpolation is used by default. Supported interpolations are ``lower``, `higher`, `midpoint`, `nearest` and `linear`.
- **keepdims** (`bool`) – If True, the axis is remained as an axis of size one.

**Returns** The percentiles of `a`, along the axis if specified.

**Return type** `cupy.ndarray`

See also:

`numpy.percentile()`

## cupy.quantile

`cupy.quantile(a, q, axis=None, out=None, interpolation='linear', keepdims=False)`

Computes the q-th quantile of the data along the specified axis.

### Parameters

- **a** (`cupy.ndarray`) – Array for which to compute quantiles.
- **q** (`float`, *tuple of floats* or `cupy.ndarray`) – Quantiles to compute in the range between 0 and 1 inclusive.
- **axis** (`int` or *tuple of ints*) – Along which axis or axes to compute the quantiles. The flattened array is used by default.
- **out** (`cupy.ndarray`) – Output array.
- **interpolation** (`str`) – Interpolation method when a quantile lies between two data points. linear interpolation is used by default. Supported interpolations are ``lower``, `higher`, `midpoint`, `nearest` and `linear`.
- **keepdims** (`bool`) – If True, the axis is remained as an axis of size one.

**Returns** The quantiles of `a`, along the axis if specified.

**Return type** `cupy.ndarray`

See also:

`numpy.quantile()`

## Averages and variances

<code>median(a[, axis, out, overwrite_input, keepdims])</code>	Compute the median along the specified axis.
<code>average(a[, axis, weights, returned])</code>	Returns the weighted average along an axis.
<code>mean(a[, axis, dtype, out, keepdims])</code>	Returns the arithmetic mean along an axis.
<code>std(a[, axis, dtype, out, ddof, keepdims])</code>	Returns the standard deviation along an axis.
<code>var(a[, axis, dtype, out, ddof, keepdims])</code>	Returns the variance along an axis.
<code>nanmedian(a[, axis, out, overwrite_input, ...])</code>	Compute the median along the specified axis, while ignoring NaNs.
<code>nanmean(a[, axis, dtype, out, keepdims])</code>	Returns the arithmetic mean along an axis ignoring NaN values.
<code>nanstd(a[, axis, dtype, out, ddof, keepdims])</code>	Returns the standard deviation along an axis ignoring NaN values.
<code>nanvar(a[, axis, dtype, out, ddof, keepdims])</code>	Returns the variance along an axis ignoring NaN values.

## `cupy.median`

`cupy.median(a, axis=None, out=None, overwrite_input=False, keepdims=False)`

Compute the median along the specified axis.

Returns the median of the array elements.

### Parameters

- **a** (`cupy.ndarray`) – Array to compute the median.
- **axis** (`int`, *sequence of int or None*) – Axis along which the medians are computed. The flattened array is used by default.
- **out** (`cupy.ndarray`) – Output array.
- **overwrite\_input** (`bool`) – If `True`, then allow use of memory of input array `a` for calculations. The input array will be modified by the call to `median`. This will save memory when you do not need to preserve the contents of the input array. Treat the input as undefined, but it will probably be fully or partially sorted. Default is `False`. If `overwrite_input` is `True` and `a` is not already an ndarray, an error will be raised.
- **keepdims** (`bool`) – If `True`, the axis is remained as an axis of size one.

**Returns** The median of `a`, along the axis if specified.

**Return type** `cupy.ndarray`

See also:

`numpy.median()`

**cupy.average**

`cupy.average(a, axis=None, weights=None, returned=False)`

Returns the weighted average along an axis.

**Parameters**

- **a** (`cupy.ndarray`) – Array to compute average.
- **axis** (`int`) – Along which axis to compute average. The flattened array is used by default.
- **weights** (`cupy.ndarray`) – Array of weights where each element corresponds to the value in a. If `None`, all the values in a have a weight equal to one.
- **returned** (`bool`) – If `True`, a tuple of the average and the sum of weights is returned, otherwise only the average is returned.

**Returns** The average of the input array along the axis and the sum of weights.

**Return type** `cupy.ndarray` or tuple of `cupy.ndarray`

**Warning:** This function may synchronize the device if `weight` is given.

See also:

`numpy.average()`

**cupy.mean**

`cupy.mean(a, axis=None, dtype=None, out=None, keepdims=False)`

Returns the arithmetic mean along an axis.

**Parameters**

- **a** (`cupy.ndarray`) – Array to compute mean.
- **axis** (`int`, *sequence of int or None*) – Along which axis to compute mean. The flattened array is used by default.
- **dtype** – Data type specifier.
- **out** (`cupy.ndarray`) – Output array.
- **keepdims** (`bool`) – If `True`, the axis is remained as an axis of size one.

**Returns** The mean of the input array along the axis.

**Return type** `cupy.ndarray`

See also:

`numpy.mean()`

## cupy.std

`cupy.std(a, axis=None, dtype=None, out=None, ddof=0, keepdims=False)`

Returns the standard deviation along an axis.

### Parameters

- **a** (`cupy.ndarray`) – Array to compute standard deviation.
- **axis** (`int`) – Along which axis to compute standard deviation. The flattened array is used by default.
- **dtype** – Data type specifier.
- **out** (`cupy.ndarray`) – Output array.
- **keepdims** (`bool`) – If `True`, the axis is remained as an axis of size one.

**Returns** The standard deviation of the input array along the axis.

**Return type** `cupy.ndarray`

See also:

`numpy.std()`

## cupy.var

`cupy.var(a, axis=None, dtype=None, out=None, ddof=0, keepdims=False)`

Returns the variance along an axis.

### Parameters

- **a** (`cupy.ndarray`) – Array to compute variance.
- **axis** (`int`) – Along which axis to compute variance. The flattened array is used by default.
- **dtype** – Data type specifier.
- **out** (`cupy.ndarray`) – Output array.
- **keepdims** (`bool`) – If `True`, the axis is remained as an axis of size one.

**Returns** The variance of the input array along the axis.

**Return type** `cupy.ndarray`

See also:

`numpy.var()`

## cupy.nanmedian

`cupy.nanmedian(a, axis=None, out=None, overwrite_input=False, keepdims=False)`

Compute the median along the specified axis, while ignoring NaNs.

Returns the median of the array elements.

### Parameters

- **a** (`cupy.ndarray`) – Array to compute the median.
- **axis** (`int`, *sequence of int or None*) – Axis along which the medians are computed. The flattened array is used by default.

- **out** (`cupy.ndarray`) – Output array.
- **overwrite\_input** (`bool`) – If `True`, then allow use of memory of input array `a` for calculations. The input array will be modified by the call to `median`. This will save memory when you do not need to preserve the contents of the input array. Treat the input as undefined, but it will probably be fully or partially sorted. Default is `False`. If `overwrite_input` is `True` and `a` is not already an ndarray, an error will be raised.
- **keepdims** (`bool`) – If `True`, the axis is remained as an axis of size one.

**Returns** The median of `a`, along the axis if specified.

**Return type** `cupy.ndarray`

See also:

`numpy.nanmedian()`

### `cupy.nanmean`

`cupy.nanmean(a, axis=None, dtype=None, out=None, keepdims=False)`

Returns the arithmetic mean along an axis ignoring NaN values.

#### Parameters

- **a** (`cupy.ndarray`) – Array to compute mean.
- **axis** (`int`, *sequence of int or None*) – Along which axis to compute mean. The flattened array is used by default.
- **dtype** – Data type specifier.
- **out** (`cupy.ndarray`) – Output array.
- **keepdims** (`bool`) – If `True`, the axis is remained as an axis of size one.

**Returns** The mean of the input array along the axis ignoring NaNs.

**Return type** `cupy.ndarray`

See also:

`numpy.nanmean()`

### `cupy.nanstd`

`cupy.nanstd(a, axis=None, dtype=None, out=None, ddof=0, keepdims=False)`

Returns the standard deviation along an axis ignoring NaN values.

#### Parameters

- **a** (`cupy.ndarray`) – Array to compute standard deviation.
- **axis** (`int`) – Along which axis to compute standard deviation. The flattened array is used by default.
- **dtype** – Data type specifier.
- **out** (`cupy.ndarray`) – Output array.
- **keepdims** (`bool`) – If `True`, the axis is remained as an axis of size one.

**Returns** The standard deviation of the input array along the axis.

**Return type** *cupy.ndarray*

**See also:**

*numpy.nanstd()*

## **cupy.nanvar**

**cupy.nanvar**(*a*, *axis=None*, *dtype=None*, *out=None*, *ddof=0*, *keepdims=False*)

Returns the variance along an axis ignoring NaN values.

### **Parameters**

- **a** (*cupy.ndarray*) – Array to compute variance.
- **axis** (*int*) – Along which axis to compute variance. The flattened array is used by default.
- **dtype** – Data type specifier.
- **out** (*cupy.ndarray*) – Output array.
- **keepdims** (*bool*) – If *True*, the axis is remained as an axis of size one.

**Returns** The variance of the input array along the axis.

**Return type** *cupy.ndarray*

**See also:**

*numpy.nanvar()*

## **Correlations**

<i>corrcoef</i> ( <i>a</i> [, <i>y</i> , <i>rowvar</i> , <i>bias</i> , <i>ddof</i> ])	Returns the Pearson product-moment correlation coefficients of an array.
<i>correlate</i> ( <i>a</i> , <i>v</i> [, <i>mode</i> ])	Returns the cross-correlation of two 1-dimensional sequences.
<i>cov</i> ( <i>a</i> [, <i>y</i> , <i>rowvar</i> , <i>bias</i> , <i>ddof</i> ])	Returns the covariance matrix of an array.

## **cupy.corrcoef**

**cupy.corrcoef**(*a*, *y=None*, *rowvar=True*, *bias=None*, *ddof=None*)

Returns the Pearson product-moment correlation coefficients of an array.

### **Parameters**

- **a** (*cupy.ndarray*) – Array to compute the Pearson product-moment correlation coefficients.
- **y** (*cupy.ndarray*) – An additional set of variables and observations.
- **rowvar** (*bool*) – If *True*, then each row represents a variable, with observations in the columns. Otherwise, the relationship is transposed.
- **bias** (*None*) – Has no effect, do not use.
- **ddof** (*None*) – Has no effect, do not use.

**Returns** The Pearson product-moment correlation coefficients of the input array.

**Return type** *cupy.ndarray*

See also:

`numpy.corrcoef()`

### `cupy.correlate`

`cupy.correlate(a, v, mode='valid')`

Returns the cross-correlation of two 1-dimensional sequences.

#### Parameters

- **a** (`cupy.ndarray`) – first 1-dimensional input.
- **v** (`cupy.ndarray`) – second 1-dimensional input.
- **mode** (`str`, optional) – *valid*, *same*, *full*

**Returns** Discrete cross-correlation of a and v.

**Return type** `cupy.ndarray`

See also:

`numpy.correlate()`

### `cupy.cov`

`cupy.cov(a, y=None, rowvar=True, bias=False, ddof=None)`

Returns the covariance matrix of an array.

This function currently does not support `fweights` and `aweights` options.

#### Parameters

- **a** (`cupy.ndarray`) – Array to compute covariance matrix.
- **y** (`cupy.ndarray`) – An additional set of variables and observations.
- **rowvar** (`bool`) – If `True`, then each row represents a variable, with observations in the columns. Otherwise, the relationship is transposed.
- **bias** (`bool`) – If `False`, normalization is by  $(N - 1)$ , where  $N$  is the number of observations given (unbiased estimate). If `True`, then normalization is by  $N$ .
- **ddof** (`int`) – If not `None` the default value implied by `bias` is overridden. Note that `ddof=1` will return the unbiased estimate and `ddof=0` will return the simple average.

**Returns** The covariance matrix of the input array.

**Return type** `cupy.ndarray`

See also:

`numpy.cov()`

## Histograms

<code>histogram(x[, bins, range, weights, density])</code>	Computes the histogram of a set of data.
<code>histogram2d(x, y[, bins, range, weights, ...])</code>	Compute the bi-dimensional histogram of two data samples.
<code>histogramdd(sample[, bins, range, weights, ...])</code>	Compute the multidimensional histogram of some data.
<code>bincount(x[, weights, minlength])</code>	Count number of occurrences of each value in array of non-negative ints.
<code>digitize(x, bins[, right])</code>	Finds the indices of the bins to which each value in input array belongs.

### cupy.histogram

`cupy.histogram(x, bins=10, range=None, weights=None, density=False)`

Computes the histogram of a set of data.

#### Parameters

- **x** (`cupy.ndarray`) – Input array.
- **bins** (`int` or `cupy.ndarray`) – If bins is an int, it represents the number of bins. If bins is an `ndarray`, it represents a bin edges.
- **range** (*2-tuple of float, optional*) – The lower and upper range of the bins. If not provided, range is simply `(x.min(), x.max())`. Values outside the range are ignored. The first element of the range must be less than or equal to the second. *range* affects the automatic bin computation as well. While bin width is computed to be optimal based on the actual data within *range*, the bin count will fill the entire range including portions containing no data.
- **density** (`bool`, *optional*) – If False, the default, returns the number of samples in each bin. If True, returns the probability *density* function at the bin, `bin_count / sample_count / bin_volume`.
- **weights** (`cupy.ndarray`, *optional*) – An array of weights, of the same shape as *x*. Each value in *x* only contributes its associated weight towards the bin count (instead of 1).

**Returns** (`hist`, `bin_edges`) where `hist` is a `cupy.ndarray` storing the values of the histogram, and `bin_edges` is a `cupy.ndarray` storing the bin edges.

**Return type** `tuple`

**Warning:** This function may synchronize the device.

See also:

`numpy.histogram()`



## cupy.histogram2d

`cupy.histogram2d(x, y, bins=10, range=None, weights=None, density=None)`

Compute the bi-dimensional histogram of two data samples.

### Parameters

- **x** (`cupy.ndarray`) – The first array of samples to be histogrammed.
- **y** (`cupy.ndarray`) – The second array of samples to be histogrammed.
- **bins** (`int` or *tuple of int* or `cupy.ndarray`) – The bin specification:
  - A sequence of arrays describing the monotonically increasing bin edges along each dimension.
  - The number of bins for each dimension (nx, ny)
  - The number of bins for all dimensions (nx=ny=bins).
- **range** (*sequence*, *optional*) – A sequence of length two, each an optional (lower, upper) tuple giving the outer bin edges to be used if the edges are not given explicitly in *bins*. An entry of *None* in the sequence results in the minimum and maximum values being used for the corresponding dimension. The default, *None*, is equivalent to passing a tuple of two *None* values.
- **weights** (`cupy.ndarray`) – An array of values  $w_i$  weighing each sample  $(x_i, y_i)$ . The values of the returned histogram are equal to the sum of the weights belonging to the samples falling into each bin.
- **density** (*bool*, *optional*) – If *False*, the default, returns the number of samples in each bin. If *True*, returns the probability *density* function at the bin,  $\text{bin\_count} / \text{sample\_count} / \text{bin\_volume}$ .

### Returns

**H** (`cupy.ndarray`): The multidimensional histogram of sample x. See *normed* and *weights* for the different possible semantics.

**edges0** (*tuple of cupy.ndarray*): A list of D arrays describing the bin edges for the first dimension.

**edges1** (*tuple of cupy.ndarray*): A list of D arrays describing the bin edges for the second dimension.

**Return type** `tuple`

**Warning:** This function may synchronize the device.

See also:

`numpy.histogram2d()`

## cupy.histogramdd

`cupy.histogramdd(sample, bins=10, range=None, weights=None, density=False)`

Compute the multidimensional histogram of some data.

### Parameters

- **sample** (`cupy.ndarray`) – The data to be histogrammed. (N, D) or (D, N) array  
Note the unusual interpretation of sample when an array\_like:
  - When an array, each row is a coordinate in a D-dimensional space - such as `histogramdd(cupy.array([p1, p2, p3]))`.
  - When an array\_like, each element is the list of values for single coordinate - such as `histogramdd((X, Y, Z))`.The first form should be preferred.
- **bins** (*int or tuple of int or cupy.ndarray*) – The bin specification:
  - A sequence of arrays describing the monotonically increasing bin edges along each dimension.
  - The number of bins for each dimension (`nx, ny, ... =bins`)
  - The number of bins for all dimensions (`nx=ny=...=bins`).
- **range** (*sequence, optional*) – A sequence of length D, each an optional (lower, upper) tuple giving the outer bin edges to be used if the edges are not given explicitly in *bins*. An entry of None in the sequence results in the minimum and maximum values being used for the corresponding dimension. The default, None, is equivalent to passing a tuple of D None values.
- **weights** (`cupy.ndarray`) – An array of values  $w_i$  weighing each sample ( $x_i, y_i, z_i, \dots$ ). The values of the returned histogram are equal to the sum of the weights belonging to the samples falling into each bin.
- **density** (*bool, optional*) – If False, the default, returns the number of samples in each bin. If True, returns the probability *density* function at the bin, `bin_count / sample_count / bin_volume`.

### Returns

**H** (`cupy.ndarray`): The multidimensional histogram of sample x. See *normed* and *weights* for the different possible semantics.

**edges** (*list of cupy.ndarray*): A list of D arrays describing the bin edges for each dimension.

Return type `tuple`

**Warning:** This function may synchronize the device.

See also:

`numpy.histogramdd()`

## cupy.bincount

`cupy.bincount(x, weights=None, minlength=None)`

Count number of occurrences of each value in array of non-negative ints.

### Parameters

- **x** (`cupy.ndarray`) – Input array.
- **weights** (`cupy.ndarray`) – Weights array which has the same shape as **x**.
- **minlength** (`int`) – A minimum number of bins for the output array.

**Returns** The result of binning the input array. The length of output is equal to `max(cupy.max(x) + 1, minlength)`.

**Return type** `cupy.ndarray`

**Warning:** This function may synchronize the device.

See also:

`numpy.bincount()`

## cupy.digitize

`cupy.digitize(x, bins, right=False)`

Finds the indices of the bins to which each value in input array belongs.

---

**Note:** In order to avoid device synchronization, `digitize` does not raise an exception when the array is not monotonic

---

### Parameters

- **x** (`cupy.ndarray`) – Input array.
- **bins** (`cupy.ndarray`) – Array of bins. It has to be 1-dimensional and monotonic increasing or decreasing.
- **right** (`bool`) – Indicates whether the intervals include the right or the left bin edge.

**Returns** Output array of indices, of same shape as **x**.

**Return type** `cupy.ndarray`

See also:

`numpy.digitize()`

### 5.3.19 Test support (cupy.testing)

---

**Hint:** NumPy API Reference: Test support ([numpy.testing](#))

---

#### Asserts

---

**Hint:** These APIs can accept both `numpy.ndarray` and `cupy.ndarray`.

---

<code>assert_array_almost_equal(x, y[, decimal, ...])</code>	Raises an AssertionError if objects are not equal up to desired precision.
<code>assert_allclose(actual, desired[, rtol, ...])</code>	Raises an AssertionError if objects are not equal up to desired tolerance.
<code>assert_array_almost_equal_nulp(x, y[, nulp])</code>	Compare two arrays relatively to their spacing.
<code>assert_array_max_ulp(a, b[, maxulp, dtype])</code>	Check that all items of arrays differ in at most N Units in the Last Place.
<code>assert_array_equal(x, y[, err_msg, verbose, ...])</code>	Raises an AssertionError if two array_like objects are not equal.
<code>assert_array_less(x, y[, err_msg, verbose])</code>	Raises an AssertionError if array_like objects are not ordered by less than.

#### cupy.testing.assert\_array\_almost\_equal

`cupy.testing.assert_array_almost_equal(x, y, decimal=6, err_msg="", verbose=True)`  
Raises an AssertionError if objects are not equal up to desired precision.

##### Parameters

- **x** (`numpy.ndarray` or `cupy.ndarray`) – The actual object to check.
- **y** (`numpy.ndarray` or `cupy.ndarray`) – The desired, expected object.
- **decimal** (`int`) – Desired precision.
- **err\_msg** (`str`) – The error message to be printed in case of failure.
- **verbose** (`bool`) – If True, the conflicting values are appended to the error message.

See also:

`numpy.testing.assert_array_almost_equal()`

### cupy.testing.assert\_allclose

`cupy.testing.assert_allclose(actual, desired, rtol=1e-07, atol=0, err_msg="", verbose=True)`

Raises an AssertionError if objects are not equal up to desired tolerance.

#### Parameters

- **actual** (*numpy.ndarray* or *cupy.ndarray*) – The actual object to check.
- **desired** (*numpy.ndarray* or *cupy.ndarray*) – The desired, expected object.
- **rtol** (*float*) – Relative tolerance.
- **atol** (*float*) – Absolute tolerance.
- **err\_msg** (*str*) – The error message to be printed in case of failure.
- **verbose** (*bool*) – If True, the conflicting values are appended to the error message.

See also:

`numpy.testing.assert_allclose()`

### cupy.testing.assert\_array\_almost\_equal\_nulp

`cupy.testing.assert_array_almost_equal_nulp(x, y, nulp=1)`

Compare two arrays relatively to their spacing.

#### Parameters

- **x** (*numpy.ndarray* or *cupy.ndarray*) – The actual object to check.
- **y** (*numpy.ndarray* or *cupy.ndarray*) – The desired, expected object.
- **nulp** (*int*) – The maximum number of unit in the last place for tolerance.

See also:

`numpy.testing.assert_array_almost_equal_nulp()`

### cupy.testing.assert\_array\_max\_ulp

`cupy.testing.assert_array_max_ulp(a, b, maxulp=1, dtype=None)`

Check that all items of arrays differ in at most N Units in the Last Place.

#### Parameters

- **a** (*numpy.ndarray* or *cupy.ndarray*) – The actual object to check.
- **b** (*numpy.ndarray* or *cupy.ndarray*) – The desired, expected object.
- **maxulp** (*int*) – The maximum number of units in the last place that elements of *a* and *b* can differ.
- **dtype** (*numpy.dtype*) – Data-type to convert *a* and *b* to if given.

See also:

`numpy.testing.assert_array_max_ulp()`

### cupy.testing.assert\_array\_equal

`cupy.testing.assert_array_equal(x, y, err_msg="", verbose=True, strides_check=False)`  
Raises an `AssertionError` if two `array_like` objects are not equal.

#### Parameters

- **x** (*numpy.ndarray* or *cupy.ndarray*) – The actual object to check.
- **y** (*numpy.ndarray* or *cupy.ndarray*) – The desired, expected object.
- **strides\_check** (*bool*) – If `True`, consistency of strides is also checked.
- **err\_msg** (*str*) – The error message to be printed in case of failure.
- **verbose** (*bool*) – If `True`, the conflicting values are appended to the error message.

See also:

`numpy.testing.assert_array_equal()`

### cupy.testing.assert\_array\_less

`cupy.testing.assert_array_less(x, y, err_msg="", verbose=True)`  
Raises an `AssertionError` if `array_like` objects are not ordered by less than.

#### Parameters

- **x** (*numpy.ndarray* or *cupy.ndarray*) – The smaller object to check.
- **y** (*numpy.ndarray* or *cupy.ndarray*) – The larger object to compare.
- **err\_msg** (*str*) – The error message to be printed in case of failure.
- **verbose** (*bool*) – If `True`, the conflicting values are appended to the error message.

See also:

`numpy.testing.assert_array_less()`

## CuPy-specific APIs

### Asserts

---

<code>assert_array_list_equal(xlist, ylist[, ...])</code>	Compares lists of arrays pairwise with <code>assert_array_equal</code> .
---	--

---

### cupy.testing.assert\_array\_list\_equal

`cupy.testing.assert_array_list_equal(xlist, ylist, err_msg="", verbose=True)`  
Compares lists of arrays pairwise with `assert_array_equal`.

#### Parameters

- **x** (*array\_like*) – Array of the actual objects.
- **y** (*array\_like*) – Array of the desired, expected objects.
- **err\_msg** (*str*) – The error message to be printed in case of failure.

- **verbose** (*bool*) – If `True`, the conflicting values are appended to the error message.

Each element of `x` and `y` must be either `numpy.ndarray` or `cupy.ndarray`. `x` and `y` must have same length. Otherwise, this function raises `AssertionError`. It compares elements of `x` and `y` pairwise with `assert_array_equal()` and raises error if at least one pair is not equal.

See also:

`numpy.testing.assert_array_equal()`

## NumPy-CuPy Consistency Check

The following decorators are for testing consistency between CuPy's functions and corresponding NumPy's ones.

<code>numpy_cupy_allclose([rtol, atol, err_msg, ...])</code>	Decorator that checks NumPy results and CuPy ones are close.
<code>numpy_cupy_array_almost_equal([decimal, ...])</code>	Decorator that checks NumPy results and CuPy ones are almost equal.
<code>numpy_cupy_array_almost_equal_nulp([nulp, ...])</code>	Decorator that checks results of NumPy and CuPy are equal w.r.t.
<code>numpy_cupy_array_max_ulp([maxulp, dtype, ...])</code>	Decorator that checks results of NumPy and CuPy ones are equal w.r.t.
<code>numpy_cupy_array_equal([err_msg, verbose, ...])</code>	Decorator that checks NumPy results and CuPy ones are equal.
<code>numpy_cupy_array_list_equal([err_msg, ...])</code>	Decorator that checks the resulting lists of NumPy and CuPy's one are equal.
<code>numpy_cupy_array_less([err_msg, verbose, ...])</code>	Decorator that checks the CuPy result is less than NumPy result.

## cupy.testing.numpy\_cupy\_allclose

```
cupy.testing.numpy_cupy_allclose(rtol=1e-07, atol=0, err_msg="", verbose=True, name='xp',
                                type_check=True, accept_error=False, sp_name=None,
                                scipy_name=None, contiguous_check=True, *,
                                _check_sparse_format=True)
```

Decorator that checks NumPy results and CuPy ones are close.

### Parameters

- **rtol** (*float* or *dict*) – Relative tolerance. Besides a float value, a dictionary that maps a dtypes to a float value can be supplied to adjust tolerance per dtype. If the dictionary has 'default' string as its key, its value is used as the default tolerance in case any dtype keys do not match.
- **atol** (*float* or *dict*) – Absolute tolerance. Besides a float value, a dictionary can be supplied as `rtol`.
- **err\_msg** (*str*) – The error message to be printed in case of failure.
- **verbose** (*bool*) – If `True`, the conflicting values are appended to the error message.
- **name** (*str*) – Argument name whose value is either `numpy` or `cupy` module.
- **type\_check** (*bool*) – If `True`, consistency of dtype is also checked.

- **accept\_error** (*bool*, *Exception* or *tuple of Exception*) – Specify acceptable errors. When both NumPy test and CuPy test raises the same type of errors, and the type of the errors is specified with this argument, the errors are ignored and not raised. If it is `True` all error types are acceptable. If it is `False` no error is acceptable.
- **sp\_name** (*str* or *None*) – Argument name whose value is either `scipy.sparse` or `cupyx.scipy.sparse` module. If `None`, no argument is given for the modules.
- **scipy\_name** (*str* or *None*) – Argument name whose value is either `scipy` or `cupyx.scipy` module. If `None`, no argument is given for the modules.
- **contiguous\_check** (*bool*) – If `True`, consistency of contiguity is also checked.

Decorated test fixture is required to return the arrays whose values are close between `numpy` case and `cupy` case. For example, this test case checks `numpy.zeros` and `cupy.zeros` should return same value.

```
>>> import unittest
>>> from cupy import testing
>>> @testing.gpu
... class TestFoo(unittest.TestCase):
...
...     @testing.numpy_cupy_allclose()
...     def test_foo(self, xp):
...         # ...
...         # Prepare data with xp
...         # ...
...
...         xp_result = xp.zeros(10)
...         return xp_result
```

See also:

`cupy.testing.assert_allclose()`

### `cupy.testing.numpy_cupy_array_almost_equal`

`cupy.testing.numpy_cupy_array_almost_equal(decimal=6, err_msg="", verbose=True, name='xp', type_check=True, accept_error=False, sp_name=None, scipy_name=None)`

Decorator that checks NumPy results and CuPy ones are almost equal.

#### Parameters

- **decimal** (*int*) – Desired precision.
- **err\_msg** (*str*) – The error message to be printed in case of failure.
- **verbose** (*bool*) – If `True`, the conflicting values are appended to the error message.
- **name** (*str*) – Argument name whose value is either `numpy` or `cupy` module.
- **type\_check** (*bool*) – If `True`, consistency of dtype is also checked.
- **accept\_error** (*bool*, *Exception* or *tuple of Exception*) – Specify acceptable errors. When both NumPy test and CuPy test raises the same type of errors, and the type of the errors is specified with this argument, the errors are ignored and not raised. If it is `True` all error types are acceptable. If it is `False` no error is acceptable.
- **sp\_name** (*str* or *None*) – Argument name whose value is either `scipy.sparse` or `cupyx.scipy.sparse` module. If `None`, no argument is given for the modules.



- **scipy\_name** (*str* or *None*) – Argument name whose value is either `scipy` or `cupyx.scipy` module. If *None*, no argument is given for the modules.

Decorated test fixture is required to return the same arrays in the sense of `cupy.testing.assert_array_almost_equal()` (except the type of array module) even if `xp` is `numpy` or `cupy`.

See also:

`cupy.testing.assert_array_almost_equal()`

### `cupy.testing.numpy_cupy_array_almost_equal_nulp`

`cupy.testing.numpy_cupy_array_almost_equal_nulp(nulp=1, name='xp', type_check=True, accept_error=False, sp_name=None, scipy_name=None)`

Decorator that checks results of NumPy and CuPy are equal w.r.t. spacing.

#### Parameters

- **nulp** (*int*) – The maximum number of unit in the last place for tolerance.
- **name** (*str*) – Argument name whose value is either `numpy` or `cupy` module.
- **type\_check** (*bool*) – If *True*, consistency of dtype is also checked.
- **accept\_error** (*bool*, *Exception* or *tuple of Exception*) – Specify acceptable errors. When both NumPy test and CuPy test raises the same type of errors, and the type of the errors is specified with this argument, the errors are ignored and not raised. If it is *True*, all error types are acceptable. If it is *False*, no error is acceptable.
- **sp\_name** (*str* or *None*) – Argument name whose value is either `scipy.sparse` or `cupyx.scipy.sparse` module. If *None*, no argument is given for the modules.
- **scipy\_name** (*str* or *None*) – Argument name whose value is either `scipy` or `cupyx.scipy` module. If *None*, no argument is given for the modules.

Decorated test fixture is required to return the same arrays in the sense of `cupy.testing.assert_array_almost_equal_nulp()` (except the type of array module) even if `xp` is `numpy` or `cupy`.

See also:

`cupy.testing.assert_array_almost_equal_nulp()`

### `cupy.testing.numpy_cupy_array_max_ulp`

`cupy.testing.numpy_cupy_array_max_ulp(maxulp=1, dtype=None, name='xp', type_check=True, accept_error=False, sp_name=None, scipy_name=None)`

Decorator that checks results of NumPy and CuPy ones are equal w.r.t. ulp.

#### Parameters

- **maxulp** (*int*) – The maximum number of units in the last place that elements of resulting two arrays can differ.
- **dtype** (*numpy.dtype*) – Data-type to convert the resulting two array to if given.
- **name** (*str*) – Argument name whose value is either `numpy` or `cupy` module.
- **type\_check** (*bool*) – If *True*, consistency of dtype is also checked.

- **accept\_error** (*bool*, *Exception* or *tuple of Exception*) – Specify acceptable errors. When both NumPy test and CuPy test raises the same type of errors, and the type of the errors is specified with this argument, the errors are ignored and not raised. If it is `True` all error types are acceptable. If it is `False` no error is acceptable.
- **sp\_name** (*str* or *None*) – Argument name whose value is either `scipy.sparse` or `cupyx.scipy.sparse` module. If `None`, no argument is given for the modules.
- **scipy\_name** (*str* or *None*) – Argument name whose value is either `scipy` or `cupyx.scipy` module. If `None`, no argument is given for the modules.

Decorated test fixture is required to return the same arrays in the sense of `assert_array_max_ulp()` (except the type of array module) even if `xp` is `numpy` or `cupy`.

See also:

`cupy.testing.assert_array_max_ulp()`

### `cupy.testing.numpy_cupy_array_equal`

`cupy.testing.numpy_cupy_array_equal(err_msg='', verbose=True, name='xp', type_check=True, accept_error=False, sp_name=None, scipy_name=None, strides_check=False)`

Decorator that checks NumPy results and CuPy ones are equal.

#### Parameters

- **err\_msg** (*str*) – The error message to be printed in case of failure.
- **verbose** (*bool*) – If `True`, the conflicting values are appended to the error message.
- **name** (*str*) – Argument name whose value is either `numpy` or `cupy` module.
- **type\_check** (*bool*) – If `True`, consistency of dtype is also checked.
- **accept\_error** (*bool*, *Exception* or *tuple of Exception*) – Specify acceptable errors. When both NumPy test and CuPy test raises the same type of errors, and the type of the errors is specified with this argument, the errors are ignored and not raised. If it is `True` all error types are acceptable. If it is `False` no error is acceptable.
- **sp\_name** (*str* or *None*) – Argument name whose value is either `scipy.sparse` or `cupyx.scipy.sparse` module. If `None`, no argument is given for the modules.
- **scipy\_name** (*str* or *None*) – Argument name whose value is either `scipy` or `cupyx.scipy` module. If `None`, no argument is given for the modules.
- **strides\_check** (*bool*) – If `True`, consistency of strides is also checked.

Decorated test fixture is required to return the same arrays in the sense of `numpy_cupy_array_equal()` (except the type of array module) even if `xp` is `numpy` or `cupy`.

See also:

`cupy.testing.assert_array_equal()`

### cupy.testing.numpy\_cupy\_array\_list\_equal

```
cupy.testing.numpy_cupy_array_list_equal(err_msg="", verbose=True, name='xp', sp_name=None,
                                         scipy_name=None)
```

Decorator that checks the resulting lists of NumPy and CuPy's one are equal.

#### Parameters

- **err\_msg** (*str*) – The error message to be printed in case of failure.
- **verbose** (*bool*) – If True, the conflicting values are appended to the error message.
- **name** (*str*) – Argument name whose value is either `numpy` or `cupy` module.
- **sp\_name** (*str* or *None*) – Argument name whose value is either `scipy.sparse` or `cupyx.scipy.sparse` module. If *None*, no argument is given for the modules.
- **scipy\_name** (*str* or *None*) – Argument name whose value is either `scipy` or `cupyx.scipy` module. If *None*, no argument is given for the modules.

Decorated test fixture is required to return the same list of arrays (except the type of array module) even if `xp` is `numpy` or `cupy`.

See also:

[`cupy.testing.assert\_array\_list\_equal\(\)`](#)

### cupy.testing.numpy\_cupy\_array\_less

```
cupy.testing.numpy_cupy_array_less(err_msg="", verbose=True, name='xp', type_check=True,
                                   accept_error=False, sp_name=None, scipy_name=None)
```

Decorator that checks the CuPy result is less than NumPy result.

#### Parameters

- **err\_msg** (*str*) – The error message to be printed in case of failure.
- **verbose** (*bool*) – If True, the conflicting values are appended to the error message.
- **name** (*str*) – Argument name whose value is either `numpy` or `cupy` module.
- **type\_check** (*bool*) – If True, consistency of dtype is also checked.
- **accept\_error** (*bool*, *Exception* or *tuple of Exception*) – Specify acceptable errors. When both NumPy test and CuPy test raises the same type of errors, and the type of the errors is specified with this argument, the errors are ignored and not raised. If it is True all error types are acceptable. If it is False no error is acceptable.
- **sp\_name** (*str* or *None*) – Argument name whose value is either `scipy.sparse` or `cupyx.scipy.sparse` module. If *None*, no argument is given for the modules.
- **scipy\_name** (*str* or *None*) – Argument name whose value is either `scipy` or `cupyx.scipy` module. If *None*, no argument is given for the modules.

Decorated test fixture is required to return the smaller array when `xp` is `cupy` than the one when `xp` is `numpy`.

See also:

[`cupy.testing.assert\_array\_less\(\)`](#)

## Parameterized dtype Test

The following decorators offer the standard way for parameterized test with respect to single or the combination of dtype(s).

<code>for_dtypes(dtypes[, name])</code>	Decorator for parameterized dtype test.
<code>for_all_dtypes([name, no_float16, no_bool, ...])</code>	Decorator that checks the fixture with all dtypes.
<code>for_float_dtypes([name, no_float16])</code>	Decorator that checks the fixture with float dtypes.
<code>for_signed_dtypes([name])</code>	Decorator that checks the fixture with signed dtypes.
<code>for_unsigned_dtypes([name])</code>	Decorator that checks the fixture with unsigned dtypes.
<code>for_int_dtypes([name, no_bool])</code>	Decorator that checks the fixture with integer and optionally bool dtypes.
<code>for_complex_dtypes([name])</code>	Decorator that checks the fixture with complex dtypes.
<code>for_dtypes_combination(types[, names, full])</code>	Decorator that checks the fixture with a product set of dtypes.
<code>for_all_dtypes_combination([names, ...])</code>	Decorator that checks the fixture with a product set of all dtypes.
<code>for_signed_dtypes_combination([names, full])</code>	Decorator for parameterized test w.r.t.
<code>for_unsigned_dtypes_combination([names, full])</code>	Decorator for parameterized test w.r.t.
<code>for_int_dtypes_combination([names, no_bool, ...])</code>	Decorator for parameterized test w.r.t.

## cupy.testing.for\_dtypes

`cupy.testing.for_dtypes(dtypes, name='dtype')`

Decorator for parameterized dtype test.

### Parameters

- **dtypes** (*list of dtypes*) – dtypes to be tested.
- **name** (*str*) – Argument name to which specified dtypes are passed.

This decorator adds a keyword argument specified by `name` to the test fixture. Then, it runs the fixtures in parallel by passing the each element of `dtypes` to the named argument.

## cupy.testing.for\_all\_dtypes

`cupy.testing.for_all_dtypes(name='dtype', no_float16=False, no_bool=False, no_complex=False)`

Decorator that checks the fixture with all dtypes.

### Parameters

- **name** (*str*) – Argument name to which specified dtypes are passed.
- **no\_float16** (*bool*) – If True, `numpy.float16` is omitted from candidate dtypes.
- **no\_bool** (*bool*) – If True, `numpy.bool_` is omitted from candidate dtypes.
- **no\_complex** (*bool*) – If True, `numpy.complex64` and `numpy.complex128` are omitted from candidate dtypes.

dtypes to be tested: `numpy.complex64` (optional), `numpy.complex128` (optional), `numpy.float16` (optional), `numpy.float32`, `numpy.float64`, `numpy.dtype('b')`, `numpy.dtype('h')`, `numpy.dtype('i')`, `numpy.dtype('l')`, `numpy.dtype('q')`, `numpy.dtype('B')`, `numpy.dtype('H')`, `numpy.dtype('I')`, `numpy.dtype('L')`, `numpy.dtype('Q')`, and `numpy.bool_` (optional).

The usage is as follows. This test fixture checks if `cPickle` successfully reconstructs `cupy.ndarray` for various dtypes. `dtype` is an argument inserted by the decorator.

```
>>> import unittest
>>> from cupy import testing
>>> @testing.gpu
... class TestNpz(unittest.TestCase):
...
...     @testing.for_all_dtypes()
...     def test_pickle(self, dtype):
...         a = testing.shaped_arange((2, 3, 4), dtype=dtype)
...         s = pickle.dumps(a)
...         b = pickle.loads(s)
...         testing.assert_array_equal(a, b)
```

Typically, we use this decorator in combination with decorators that check consistency between NumPy and CuPy like `cupy.testing.numpy_cupy_allclose()`. The following is such an example.

```
>>> import unittest
>>> from cupy import testing
>>> @testing.gpu
... class TestMean(unittest.TestCase):
...
...     @testing.for_all_dtypes()
...     @testing.numpy_cupy_allclose()
...     def test_mean_all(self, xp, dtype):
...         a = testing.shaped_arange((2, 3), xp, dtype)
...         return a.mean()
```

See also:

`cupy.testing.for_dtypes()`

### `cupy.testing.for_float_dtypes`

`cupy.testing.for_float_dtypes(name='dtype', no_float16=False)`

Decorator that checks the fixture with float dtypes.

#### Parameters

- **name** (*str*) – Argument name to which specified dtypes are passed.
- **no\_float16** (*bool*) – If True, `numpy.float16` is omitted from candidate dtypes.

dtypes to be tested are `numpy.float16` (optional), `numpy.float32`, and `numpy.float64`.

See also:

`cupy.testing.for_dtypes()`, `cupy.testing.for_all_dtypes()`

### `cupy.testing.for_signed_dtypes`

`cupy.testing.for_signed_dtypes(name='dtype')`  
Decorator that checks the fixture with signed dtypes.

**Parameters** `name` (*str*) – Argument name to which specified dtypes are passed.

dtypes to be tested are `numpy.dtype('b')`, `numpy.dtype('h')`, `numpy.dtype('i')`, `numpy.dtype('l')`, and `numpy.dtype('q')`.

**See also:**

[`cupy.testing.for\_dtypes\(\)`](#), [`cupy.testing.for\_all\_dtypes\(\)`](#)

### `cupy.testing.for_unsigned_dtypes`

`cupy.testing.for_unsigned_dtypes(name='dtype')`  
Decorator that checks the fixture with unsigned dtypes.

**Parameters** `name` (*str*) – Argument name to which specified dtypes are passed.

dtypes to be tested are `numpy.dtype('B')`, `numpy.dtype('H')`,  
`numpy.dtype('I')`, `numpy.dtype('L')`, and `numpy.dtype('Q')`.

**See also:**

[`cupy.testing.for\_dtypes\(\)`](#), [`cupy.testing.for\_all\_dtypes\(\)`](#)

### `cupy.testing.for_int_dtypes`

`cupy.testing.for_int_dtypes(name='dtype', no_bool=False)`  
Decorator that checks the fixture with integer and optionally bool dtypes.

**Parameters**

- **name** (*str*) – Argument name to which specified dtypes are passed.
- **no\_bool** (*bool*) – If True, `numpy.bool_` is omitted from candidate dtypes.

dtypes to be tested are `numpy.dtype('b')`, `numpy.dtype('h')`, `numpy.dtype('i')`, `numpy.dtype('l')`, `numpy.dtype('q')`, `numpy.dtype('B')`, `numpy.dtype('H')`, `numpy.dtype('I')`, `numpy.dtype('L')`, `numpy.dtype('Q')`, and `numpy.bool_` (optional).

**See also:**

[`cupy.testing.for\_dtypes\(\)`](#), [`cupy.testing.for\_all\_dtypes\(\)`](#)

### `cupy.testing.for_complex_dtypes`

`cupy.testing.for_complex_dtypes(name='dtype')`  
Decorator that checks the fixture with complex dtypes.

**Parameters** `name` (*str*) – Argument name to which specified dtypes are passed.

dtypes to be tested are `numpy.complex64` and `numpy.complex128`.

**See also:**

[`cupy.testing.for\_dtypes\(\)`](#), [`cupy.testing.for\_all\_dtypes\(\)`](#)

### `cupy.testing.for_dtypes_combination`

`cupy.testing.for_dtypes_combination(types, names=('dtype',), full=None)`

Decorator that checks the fixture with a product set of dtypes.

#### Parameters

- **types** (*list of dtypes*) – dtypes to be tested.
- **names** (*list of str*) – Argument names to which dtypes are passed.
- **full** (*bool*) – If True, then all combinations of dtypes will be tested. Otherwise, the subset of combinations will be tested (see the description below).

Decorator adds the keyword arguments specified by `names` to the test fixture. Then, it runs the fixtures in parallel with passing (possibly a subset of) the product set of dtypes. The range of dtypes is specified by `types`.

The combination of dtypes to be tested changes depending on the option `full`. If `full` is True, all combinations of `types` are tested. Sometimes, such an exhaustive test can be costly. So, if `full` is False, only a subset of possible combinations is randomly sampled. If `full` is None, the behavior is determined by an environment variable `CUPY_TEST_FULL_COMBINATION`. If the value is set to '1', it behaves as if `full=True`, and otherwise `full=False`.

### `cupy.testing.for_all_dtypes_combination`

`cupy.testing.for_all_dtypes_combination(names=('dtyes',), no_float16=False, no_bool=False, full=None, no_complex=False)`

Decorator that checks the fixture with a product set of all dtypes.

#### Parameters

- **names** (*list of str*) – Argument names to which dtypes are passed.
- **no\_float16** (*bool*) – If True, `numpy.float16` is omitted from candidate dtypes.
- **no\_bool** (*bool*) – If True, `numpy.bool_` is omitted from candidate dtypes.
- **full** (*bool*) – If True, then all combinations of dtypes will be tested. Otherwise, the subset of combinations will be tested (see description in [`cupy.testing.for\_dtypes\_combination\(\)`](#)).
- **no\_complex** (*bool*) – If, True, `numpy.complex64` and `numpy.complex128` are omitted from candidate dtypes.

See also:

[`cupy.testing.for\_dtypes\_combination\(\)`](#)

### `cupy.testing.for_signed_dtypes_combination`

`cupy.testing.for_signed_dtypes_combination(names=('dtype',), full=None)`

Decorator for parameterized test w.r.t. the product set of signed dtypes.

#### Parameters

- **names** (*list of str*) – Argument names to which dtypes are passed.
- **full** (*bool*) – If True, then all combinations of dtypes will be tested. Otherwise, the subset of combinations will be tested (see description in [`cupy.testing.for\_dtypes\_combination\(\)`](#)).

See also:

[`cupy.testing.for\_dtypes\_combination\(\)`](#)

### `cupy.testing.for_unsigned_dtypes_combination`

`cupy.testing.for_unsigned_dtypes_combination(names=('dtype',), full=None)`

Decorator for parameterized test w.r.t. the product set of unsigned dtypes.

#### Parameters

- **names** (*list of str*) – Argument names to which dtypes are passed.
- **full** (*bool*) – If True, then all combinations of dtypes will be tested. Otherwise, the subset of combinations will be tested (see description in [`cupy.testing.for\_dtypes\_combination\(\)`](#)).

See also:

[`cupy.testing.for\_dtypes\_combination\(\)`](#)

### `cupy.testing.for_int_dtypes_combination`

`cupy.testing.for_int_dtypes_combination(names=('dtype',), no_bool=False, full=None)`

Decorator for parameterized test w.r.t. the product set of int and boolean.

#### Parameters

- **names** (*list of str*) – Argument names to which dtypes are passed.
- **no\_bool** (*bool*) – If True, `numpy.bool_` is omitted from candidate dtypes.
- **full** (*bool*) – If True, then all combinations of dtypes will be tested. Otherwise, the subset of combinations will be tested (see description in [`cupy.testing.for\_dtypes\_combination\(\)`](#)).

See also:

[`cupy.testing.for\_dtypes\_combination\(\)`](#)

## Parameterized order Test

The following decorators offer the standard way to parameterize tests with orders.

<a href="#"><code>for_orders</code></a> (orders[, name])	Decorator to parameterize tests with order.
<a href="#"><code>for_CF_orders</code></a> ([name])	Decorator that checks the fixture with orders ‘C’ and ‘F’.



### cupy.testing.for\_orders

`cupy.testing.for_orders`(*orders*, *name*='order')

Decorator to parameterize tests with order.

#### Parameters

- **orders** (*list of order*) – orders to be tested.
- **name** (*str*) – Argument name to which the specified order is passed.

This decorator adds a keyword argument specified by *name* to the test fixtures. Then, the fixtures run by passing each element of *orders* to the named argument.

### cupy.testing.for\_CF\_orders

`cupy.testing.for_CF_orders`(*name*='order')

Decorator that checks the fixture with orders 'C' and 'F'.

**Parameters** *name* (*str*) – Argument name to which the specified order is passed.

See also:

[`cupy.testing.for\_all\_dtypes\(\)`](#)

## 5.3.20 Window functions

---

**Hint:** [NumPy API Reference: Window functions](#)

---

### Various windows

<a href="#"><code>bartlett</code></a> ( <i>M</i> )	Returns the Bartlett window.
<a href="#"><code>blackman</code></a> ( <i>M</i> )	Returns the Blackman window.
<a href="#"><code>hamming</code></a> ( <i>M</i> )	Returns the Hamming window.
<a href="#"><code>hanning</code></a> ( <i>M</i> )	Returns the Hanning window.
<a href="#"><code>kaiser</code></a> ( <i>M</i> , <i>beta</i> )	Return the Kaiser window.

### cupy.bartlett

`cupy.bartlett`(*M*)

Returns the Bartlett window.

The Bartlett window is defined as

$$w(n) = \frac{2}{M-1} \left( \frac{M-1}{2} - \left| n - \frac{M-1}{2} \right| \right)$$

**Parameters** *M* (*int*) – Number of points in the output window. If zero or less, an empty array is returned.

**Returns** Output ndarray.

**Return type** *ndarray*

See also:

`numpy.bartlett()`

## `cupy.blackman`

`cupy.blackman(M)`

Returns the Blackman window.

The Blackman window is defined as

$$w(n) = 0.42 - 0.5 \cos\left(\frac{2\pi n}{M-1}\right) + 0.08 \cos\left(\frac{4\pi n}{M-1}\right) \quad 0 \leq n \leq M-1$$

**Parameters** **M** (`int`) – Number of points in the output window. If zero or less, an empty array is returned.

**Returns** Output ndarray.

**Return type** `ndarray`

See also:

`numpy.blackman()`

## `cupy.hamming`

`cupy.hamming(M)`

Returns the Hamming window.

The Hamming window is defined as

$$w(n) = 0.54 - 0.46 \cos\left(\frac{2\pi n}{M-1}\right) \quad 0 \leq n \leq M-1$$

**Parameters** **M** (`int`) – Number of points in the output window. If zero or less, an empty array is returned.

**Returns** Output ndarray.

**Return type** `ndarray`

See also:

`numpy.hamming()`

## `cupy.hanning`

`cupy.hanning(M)`

Returns the Hanning window.

The Hanning window is defined as

$$w(n) = 0.5 - 0.5 \cos\left(\frac{2\pi n}{M-1}\right) \quad 0 \leq n \leq M-1$$

**Parameters** **M** (`int`) – Number of points in the output window. If zero or less, an empty array is returned.

**Returns** Output ndarray.

**Return type** *ndarray*

**See also:**

`numpy.hanning()`

## cupy.kaiser

`cupy.kaiser(M, beta)`

Return the Kaiser window. The Kaiser window is a taper formed by using a Bessel function.

$$w(n) = I_0 \left( \beta \sqrt{1 - \frac{4n^2}{(M-1)^2}} \right) / I_0(\beta)$$

with

$$-\frac{M-1}{2} \leq n \leq \frac{M-1}{2}$$

where  $I_0$  is the modified zeroth-order Bessel function.

**Args:**

**M (int):** Number of points in the output window. If zero or less, an empty array is returned.

**beta (float):** Shape parameter for window

**Returns** The window, with the maximum value normalized to one (the value one appears only if the number of samples is odd).

**Return type** *ndarray*

**See also:**

`numpy.kaiser()`

## 5.3.21 CUB/cuTENSOR backend for some CuPy routines

Some CuPy reduction routines, including `sum()`, `amin()`, `amax()`, `argmin()`, `argmax()`, and other functions built on top of them, can be accelerated by switching to the **CUB** or **cuTENSOR** backend. These backends can be enabled by setting the `CUPY_ACCELERATORS` environment variable as documented [here](#). Note that while in general the accelerated reductions are faster, there could be exceptions depending on the data layout. In particular, the CUB reduction only supports reduction over contiguous axes.

CUB also accelerates other routines, such as inclusive scans (ex: `cumsum()`), histograms, sparse matrix-vector multiplications (not applicable in CUDA 11), and `cupy.ReductionKernel`.

In any case, we recommend users to perform some benchmarks to determine whether CUB/cuTENSOR offers better performance or not.

## 5.4 Routines (SciPy)

The following pages describe SciPy-compatible routines. These functions cover a subset of [SciPy routines](#).

### 5.4.1 Discrete Fourier transforms (`cupyx.scipy.fft`)

---

**Hint:** [SciPy API Reference: Discrete Fourier transforms \(scipy.fft\)](#)

---

**See also:**

*Fast Fourier Transform with CuPy*

#### Fast Fourier Transforms (FFTs)

<code>fft(x[, n, axis, norm, overwrite_x, plan])</code>	Compute the one-dimensional FFT.
<code>ifft(x[, n, axis, norm, overwrite_x, plan])</code>	Compute the one-dimensional inverse FFT.
<code>fft2(x[, s, axes, norm, overwrite_x, plan])</code>	Compute the two-dimensional FFT.
<code>ifft2(x[, s, axes, norm, overwrite_x, plan])</code>	Compute the two-dimensional inverse FFT.
<code>fftn(x[, s, axes, norm, overwrite_x, plan])</code>	Compute the N-dimensional FFT.
<code>ifftn(x[, s, axes, norm, overwrite_x, plan])</code>	Compute the N-dimensional inverse FFT.
<code>rfft(x[, n, axis, norm, overwrite_x, plan])</code>	Compute the one-dimensional FFT for real input.
<code>irfft(x[, n, axis, norm, overwrite_x, plan])</code>	Compute the one-dimensional inverse FFT for real input.
<code>rfft2(x[, s, axes, norm, overwrite_x, plan])</code>	Compute the two-dimensional FFT for real input.
<code>irfft2(x[, s, axes, norm, overwrite_x, plan])</code>	Compute the two-dimensional inverse FFT for real input.
<code>rfftn(x[, s, axes, norm, overwrite_x, plan])</code>	Compute the N-dimensional FFT for real input.
<code>irfftn(x[, s, axes, norm, overwrite_x, plan])</code>	Compute the N-dimensional inverse FFT for real input.
<code>hfft(x[, n, axis, norm, overwrite_x, plan])</code>	Compute the FFT of a signal that has Hermitian symmetry.
<code>ihfft(x[, n, axis, norm, overwrite_x, plan])</code>	Compute the FFT of a signal that has Hermitian symmetry.

#### `cupyx.scipy.fft.fft`

`cupyx.scipy.fft.fft(x, n=None, axis=-1, norm=None, overwrite_x=False, *, plan=None)`

Compute the one-dimensional FFT.

##### Parameters

- **x** (`cupy.ndarray`) – Array to be transformed.
- **n** (`None` or `int`) – Length of the transformed axis of the output. If `n` is not given, the length of the input along the axis specified by `axis` is used.
- **axis** (`int`) – Axis over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is `None`, which is an alias of "backward".
- **overwrite\_x** (`bool`) – If `True`, the contents of `x` can be destroyed.
- **plan** (`cupy.cuda.cufft.Plan1d` or `None`) – a cuFFT plan for transforming `x` over `axis`, which can be obtained using:

```
plan = cupyx.scipy.fftpack.get_fft_plan(x, n, axis)
```

Note that `plan` is defaulted to `None`, meaning CuPy will use an auto-generated plan behind the scene.

**Returns** The transformed array which shape is specified by `n` and type will convert to complex if that of the input is another.

**Return type** `cupy.ndarray`

See also:

`scipy.fft.fft()`

## cupyx.scipy.fft.iff

`cupyx.scipy.fft.iff(x, n=None, axis=-1, norm=None, overwrite_x=False, *, plan=None)`

Compute the one-dimensional inverse FFT.

### Parameters

- **x** (`cupy.ndarray`) – Array to be transformed.
- **n** (`None` or `int`) – Length of the transformed axis of the output. If `n` is not given, the length of the input along the axis specified by `axis` is used.
- **axis** (`int`) – Axis over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is `None`, which is an alias of "backward".
- **overwrite\_x** (`bool`) – If `True`, the contents of `x` can be destroyed.
- **plan** (`cupy.cuda.cufft.Plan1d` or `None`) – a cuFFT plan for transforming `x` over `axis`, which can be obtained using:

```
plan = cupyx.scipy.fftpack.get_fft_plan(x, n, axis)
```

Note that `plan` is defaulted to `None`, meaning CuPy will use an auto-generated plan behind the scene.

**Returns** The transformed array which shape is specified by `n` and type will convert to complex if that of the input is another.

**Return type** `cupy.ndarray`

See also:

`scipy.fft.iff()`

## cupyx.scipy.fft.fft2

`cupyx.scipy.fft.fft2(x, s=None, axes=(-2, -1), norm=None, overwrite_x=False, *, plan=None)`  
Compute the two-dimensional FFT.

### Parameters

- **x** (`cupy.ndarray`) – Array to be transformed.
- **s** (*None or tuple of ints*) – Shape of the transformed axes of the output. If **s** is not given, the lengths of the input along the axes specified by **axes** are used.
- **axes** (*tuple of ints*) – Axes over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is `None`, which is an alias of "backward".
- **overwrite\_x** (*bool*) – If True, the contents of **x** can be destroyed.
- **plan** (`cupy.cuda.cufft.PlanNd` or `None`) – a cuFFT plan for transforming **x** over **axes**, which can be obtained using:

```
plan = cupyx.scipy.fftpack.get_fft_plan(x, s, axes)
```

Note that **plan** is defaulted to `None`, meaning CuPy will use an auto-generated plan behind the scene.

**Returns** The transformed array which shape is specified by **s** and type will convert to complex if that of the input is another.

**Return type** `cupy.ndarray`

See also:

`scipy.fft.fft2()`

## cupyx.scipy.fft.ifft2

`cupyx.scipy.fft.ifft2(x, s=None, axes=(-2, -1), norm=None, overwrite_x=False, *, plan=None)`  
Compute the two-dimensional inverse FFT.

### Parameters

- **x** (`cupy.ndarray`) – Array to be transformed.
- **s** (*None or tuple of ints*) – Shape of the transformed axes of the output. If **s** is not given, the lengths of the input along the axes specified by **axes** are used.
- **axes** (*tuple of ints*) – Axes over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is `None`, which is an alias of "backward".
- **overwrite\_x** (*bool*) – If True, the contents of **x** can be destroyed.
- **plan** (`cupy.cuda.cufft.PlanNd` or `None`) – a cuFFT plan for transforming **x** over **axes**, which can be obtained using:

```
plan = cupyx.scipy.fftpack.get_fft_plan(x, s, axes)
```

Note that **plan** is defaulted to `None`, meaning CuPy will use an auto-generated plan behind the scene.

**Returns** The transformed array which shape is specified by `s` and type will convert to complex if that of the input is another.

**Return type** `cupy.ndarray`

See also:

`scipy.fft.ifft2()`

### `cupyx.scipy.fft.fftn`

`cupyx.scipy.fft.fftn(x, s=None, axes=None, norm=None, overwrite_x=False, *, plan=None)`

Compute the N-dimensional FFT.

#### Parameters

- **x** (`cupy.ndarray`) – Array to be transformed.
- **s** (*None or tuple of ints*) – Shape of the transformed axes of the output. If `s` is not given, the lengths of the input along the axes specified by `axes` are used.
- **axes** (*tuple of ints*) – Axes over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is `None`, which is an alias of "backward".
- **overwrite\_x** (*bool*) – If `True`, the contents of `x` can be destroyed.
- **plan** (`cupy.cuda.cufft.PlanNd` or `None`) – a cuFFT plan for transforming `x` over `axes`, which can be obtained using:

```
plan = cupyx.scipy.fftpack.get_fftn_plan(x, s, axes)
```

Note that `plan` is defaulted to `None`, meaning CuPy will use an auto-generated plan behind the scene.

**Returns** The transformed array which shape is specified by `s` and type will convert to complex if that of the input is another.

**Return type** `cupy.ndarray`

See also:

`scipy.fft.fftn()`

### `cupyx.scipy.fft.ifftn`

`cupyx.scipy.fft.ifftn(x, s=None, axes=None, norm=None, overwrite_x=False, *, plan=None)`

Compute the N-dimensional inverse FFT.

#### Parameters

- **x** (`cupy.ndarray`) – Array to be transformed.
- **s** (*None or tuple of ints*) – Shape of the transformed axes of the output. If `s` is not given, the lengths of the input along the axes specified by `axes` are used.
- **axes** (*tuple of ints*) – Axes over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is `None`, which is an alias of "backward".

- **overwrite\_x** (*bool*) – If True, the contents of **x** can be destroyed.
- **plan** (`cupy.cuda.cufft.PlanNd` or `None`) – a cuFFT plan for transforming **x** over **axes**, which can be obtained using:

```
plan = cupyx.scipy.fftpack.get_fft_plan(x, s, axes)
```

Note that **plan** is defaulted to `None`, meaning CuPy will use an auto-generated plan behind the scene.

**Returns** The transformed array which shape is specified by **s** and type will convert to complex if that of the input is another.

**Return type** *cupy.ndarray*

See also:

`scipy.fft.ifftn()`

## cupyx.scipy.fft.rfft

`cupyx.scipy.fft.rfft(x, n=None, axis=-1, norm=None, overwrite_x=False, *, plan=None)`

Compute the one-dimensional FFT for real input.

The returned array contains the positive frequency components of the corresponding `fft()`, up to and including the Nyquist frequency.

### Parameters

- **x** (*cupy.ndarray*) – Array to be transformed.
- **n** (*None* or *int*) – Length of the transformed axis of the output. If **n** is not given, the length of the input along the axis specified by **axis** is used.
- **axis** (*int*) – Axis over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is `None`, which is an alias of "backward".
- **overwrite\_x** (*bool*) – If True, the contents of **x** can be destroyed.
- **plan** (`cupy.cuda.cufft.Plan1d` or `None`) – a cuFFT plan for transforming **x** over **axis**, which can be obtained using:

```
plan = cupyx.scipy.fftpack.get_fft_plan(x, n, axis,
                                         value_type='R2C')
```

Note that **plan** is defaulted to `None`, meaning CuPy will use an auto-generated plan behind the scene.

**Returns** The transformed array.

**Return type** *cupy.ndarray*

See also:

`scipy.fft.rfft()`



**cupyx.scipy.fft.irfft**

`cupyx.scipy.fft.irfft(x, n=None, axis=-1, norm=None, overwrite_x=False, *, plan=None)`  
 Compute the one-dimensional inverse FFT for real input.

**Parameters**

- **x** (`cupy.ndarray`) – Array to be transformed.
- **n** (`None` or `int`) – Length of the transformed axis of the output. If `n` is not given, the length of the input along the axis specified by `axis` is used.
- **axis** (`int`) – Axis over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is `None`, which is an alias of "backward".
- **overwrite\_x** (`bool`) – If `True`, the contents of `x` can be destroyed.
- **plan** (`cupy.cuda.cufft.Plan1d` or `None`) – a cuFFT plan for transforming `x` over `axis`, which can be obtained using:

```
plan = cupyx.scipy.fftpack.get_fft_plan(x, n, axis,
                                       value_type='C2R')
```

Note that `plan` is defaulted to `None`, meaning CuPy will use an auto-generated plan behind the scene.

**Returns** The transformed array.

**Return type** `cupy.ndarray`

See also:

`scipy.fft.irfft()`

**cupyx.scipy.fft.rfft2**

`cupyx.scipy.fft.rfft2(x, s=None, axes=(-2, -1), norm=None, overwrite_x=False, *, plan=None)`  
 Compute the two-dimensional FFT for real input.

**Parameters**

- **a** (`cupy.ndarray`) – Array to be transform.
- **s** (`None` or `tuple of ints`) – Shape to use from the input. If `s` is not given, the lengths of the input along the axes specified by `axes` are used.
- **axes** (`tuple of ints`) – Axes over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is `None`, which is an alias of "backward".
- **overwrite\_x** (`bool`) – If `True`, the contents of `x` can be destroyed.
- **plan** (`cupy.cuda.cufft.PlanNd` or `None`) – a cuFFT plan for transforming `x` over `axes`, which can be obtained using:

```
plan = cupyx.scipy.fftpack.get_fft_plan(x, s, axes,
                                       value_type='R2C')
```

Note that `plan` is defaulted to `None`, meaning CuPy will use an auto-generated plan behind the scene.

**Returns** The transformed array which shape is specified by `s` and type will convert to complex if the input is other. The length of the last axis transformed will be `s[-1]//2+1`.

**Return type** `cupy.ndarray`

See also:

`scipy.fft.rfft2()`

### `cupyx.scipy.fft.irfft2`

`cupyx.scipy.fft.irfft2(x, s=None, axes=(-2, -1), norm=None, overwrite_x=False, *, plan=None)`

Compute the two-dimensional inverse FFT for real input.

#### Parameters

- **a** (`cupy.ndarray`) – Array to be transform.
- **s** (`None` or `tuple of ints`) – Shape of the output. If `s` is not given, they are determined from the lengths of the input along the axes specified by `axes`.
- **axes** (`tuple of ints`) – Axes over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is `None`, which is an alias of "backward".
- **overwrite\_x** (`bool`) – If True, the contents of `x` can be destroyed.
- **plan** (`cupy.cuda.cufft.PlanNd` or `None`) – a cuFFT plan for transforming `x` over `axes`, which can be obtained using:

```
plan = cupyx.scipy.fftpack.get_fft_plan(x, s, axes,
                                       value_type='C2R')
```

Note that `plan` is defaulted to `None`, meaning CuPy will use an auto-generated plan behind the scene.

**Returns** The transformed array which shape is specified by `s` and type will convert to complex if the input is other. If `s` is not given, the length of final transformed axis of output will be  $2^{*(m-1)}$  where  $m$  is the length of the final transformed axis of the input.

**Return type** `cupy.ndarray`

See also:

`scipy.fft.irfft2()`

### `cupyx.scipy.fft.rfftn`

`cupyx.scipy.fft.rfftn(x, s=None, axes=None, norm=None, overwrite_x=False, *, plan=None)`

Compute the N-dimensional FFT for real input.

#### Parameters

- **a** (`cupy.ndarray`) – Array to be transform.
- **s** (`None` or `tuple of ints`) – Shape to use from the input. If `s` is not given, the lengths of the input along the axes specified by `axes` are used.

- **axes** (*tuple of ints*) – Axes over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is `None`, which is an alias of "backward".
- **overwrite\_x** (*bool*) – If `True`, the contents of `x` can be destroyed.
- **plan** (`cupy.cuda.cufft.PlanNd` or `None`) – a cuFFT plan for transforming `x` over `axes`, which can be obtained using:

```
plan = cupyx.scipy.fftpack.get_fft_plan(x, s, axes,
                                       value_type='R2C')
```

Note that `plan` is defaulted to `None`, meaning CuPy will use an auto-generated plan behind the scene.

**Returns** The transformed array which shape is specified by `s` and type will convert to complex if the input is other. The length of the last axis transformed will be `s[-1]//2+1`.

**Return type** `cupy.ndarray`

See also:

`scipy.fft.rfftn()`

## cupyx.scipy.fft.irfftn

`cupyx.scipy.fft.irfftn(x, s=None, axes=None, norm=None, overwrite_x=False, *, plan=None)`

Compute the N-dimensional inverse FFT for real input.

### Parameters

- **a** (`cupy.ndarray`) – Array to be transform.
- **s** (*None or tuple of ints*) – Shape of the output. If `s` is not given, they are determined from the lengths of the input along the axes specified by `axes`.
- **axes** (*tuple of ints*) – Axes over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is `None`, which is an alias of "backward".
- **overwrite\_x** (*bool*) – If `True`, the contents of `x` can be destroyed.
- **plan** (`cupy.cuda.cufft.PlanNd` or `None`) – a cuFFT plan for transforming `x` over `axes`, which can be obtained using:

```
plan = cupyx.scipy.fftpack.get_fft_plan(x, s, axes,
                                       value_type='C2R')
```

Note that `plan` is defaulted to `None`, meaning CuPy will use an auto-generated plan behind the scene.

**Returns** The transformed array which shape is specified by `s` and type will convert to complex if the input is other. If `s` is not given, the length of final transformed axis of output will be  $2 \cdot (m-1)$  where  $m$  is the length of the final transformed axis of the input.

**Return type** `cupy.ndarray`

See also:

`scipy.fft.irfftn()`

### cupyx.scipy.fft.hfft

`cupyx.scipy.fft.hfft(x, n=None, axis=-1, norm=None, overwrite_x=False, *, plan=None)`

Compute the FFT of a signal that has Hermitian symmetry.

#### Parameters

- **a** (`cupy.ndarray`) – Array to be transform.
- **n** (`None` or `int`) – Length of the transformed axis of the output. For `n` output points, `n//2+1` input points are necessary. If `n` is not given, it is determined from the length of the input along the axis specified by `axis`.
- **axis** (`int`) – Axis over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is `None`, which is an alias of "backward".
- **overwrite\_x** (`bool`) – If True, the contents of `x` can be destroyed.
- **plan** (`None`) – This argument is currently not supported.

**Returns** The transformed array which shape is specified by `n` and type will convert to complex if the input is other. If `n` is not given, the length of the transformed axis is  $2*(m-1)$  where  $m$  is the length of the transformed axis of the input.

**Return type** `cupy.ndarray`

See also:

`scipy.fft.hfft()`

### cupyx.scipy.fft.ihfft

`cupyx.scipy.fft.ihfft(x, n=None, axis=-1, norm=None, overwrite_x=False, *, plan=None)`

Compute the FFT of a signal that has Hermitian symmetry.

#### Parameters

- **a** (`cupy.ndarray`) – Array to be transform.
- **n** (`None` or `int`) – Number of points along transformation axis in the input to use. If `n` is not given, the length of the input along the axis specified by `axis` is used.
- **axis** (`int`) – Axis over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is `None`, which is an alias of "backward".
- **overwrite\_x** (`bool`) – If True, the contents of `x` can be destroyed.
- **plan** (`None`) – This argument is currently not supported.

**Returns** The transformed array which shape is specified by `n` and type will convert to complex if the input is other. The length of the transformed axis is `n//2+1`.

**Return type** `cupy.ndarray`

See also:

`scipy.fft.ihfft()`

## Helper functions

<code>fftshift(x[, axes])</code>	Shift the zero-frequency component to the center of the spectrum.
<code>ifftshift(x[, axes])</code>	The inverse of <code>fftshift()</code> .
<code>fftfreq(n[, d])</code>	Return the FFT sample frequencies.
<code>rfftfreq(n[, d])</code>	Return the FFT sample frequencies for real input.
<code>next_fast_len(target[, real])</code>	Find the next fast size to <code>fft</code> .

### cupyx.scipy.fft.fftshift

`cupyx.scipy.fft.fftshift(x, axes=None)`

Shift the zero-frequency component to the center of the spectrum.

#### Parameters

- **x** (`cupy.ndarray`) – Input array.
- **axes** (`int` or `tuple of ints`) – Axes over which to shift. Default is `None`, which shifts all axes.

**Returns** The shifted array.

**Return type** `cupy.ndarray`

See also:

`numpy.fft.fftshift()`

### cupyx.scipy.fft.ifftshift

`cupyx.scipy.fft.ifftshift(x, axes=None)`

The inverse of `fftshift()`.

#### Parameters

- **x** (`cupy.ndarray`) – Input array.
- **axes** (`int` or `tuple of ints`) – Axes over which to shift. Default is `None`, which shifts all axes.

**Returns** The shifted array.

**Return type** `cupy.ndarray`

See also:

`numpy.fft.ifftshift()`

### cupyx.scipy.fft.fftfreq

`cupyx.scipy.fft.fftfreq(n, d=1.0)`  
Return the FFT sample frequencies.

**Parameters**

- **n** (*int*) – Window length.
- **d** (*scalar*) – Sample spacing.

**Returns** Array of length *n* containing the sample frequencies.

**Return type** *cupy.ndarray*

See also:

`numpy.fft.fftfreq()`

### cupyx.scipy.fft.rfftfreq

`cupyx.scipy.fft.rfftfreq(n, d=1.0)`  
Return the FFT sample frequencies for real input.

**Parameters**

- **n** (*int*) – Window length.
- **d** (*scalar*) – Sample spacing.

**Returns** Array of length  $n//2+1$  containing the sample frequencies.

**Return type** *cupy.ndarray*

See also:

`numpy.fft.rfftfreq()`

### cupyx.scipy.fft.next\_fast\_len

`cupyx.scipy.fft.next_fast_len(target, real=False)`  
Find the next fast size to `fft`.

**Parameters**

- **target** (*int*) – The size of input array.
- **real** (*bool*) – True if the FFT involves real input or output. This parameter is of no use, and only for compatibility to SciPy's interface.

**Returns** The smallest fast length greater than or equal to the input value.

**Return type** *int*

See also:

`scipy.fft.next_fast_len()`

---

**Note:** It may return a different value to `scipy.fft.next_fast_len()` as `pocketfft`'s prime factors are different from `cuFFT`'s factors. For details, see the `cuFFT` documentation.

---

## Code compatibility features

1. As with other FFT modules in CuPy, FFT functions in this module can take advantage of an existing cuFFT plan (returned by `get_fft_plan()`) to accelerate the computation. The plan can be either passed in explicitly via the keyword-only `plan` argument or used as a context manager.
2. The boolean switch `cupy.fft.config.enable_nd_planning` also affects the FFT functions in this module, see *Discrete Fourier Transform (cupy.fft)*. This switch is neglected when planning manually using `get_fft_plan()`.
3. Like in `scipy.fft`, all FFT functions in this module have an optional argument `overwrite_x` (default is `False`), which has the same semantics as in `scipy.fft`: when it is set to `True`, the input array `x` *can* (not *will*) be overwritten arbitrarily. For this reason, when an in-place FFT is desired, the user should always reassign the input in the following manner: `x = cupyx.scipy.fftpack.fft(x, ..., overwrite_x=True, ...)`.
4. The `cupyx.scipy.fft` module can also be used as a backend for `scipy.fft` e.g. by installing with `scipy.fft.set_backend(cupyx.scipy.fft)`. This can allow `scipy.fft` to work with both `numpy` and `cupy` arrays. For more information, see *SciPy FFT backend*.
5. The boolean switch `cupy.fft.config.use_multi_gpus` also affects the FFT functions in this module, see *Discrete Fourier Transform (cupy.fft)*. Moreover, this switch is *honored* when planning manually using `get_fft_plan()`.

## 5.4.2 Legacy discrete fourier transforms (cupyx.scipy.fftpack)

---

**Note:** As of SciPy version 1.4.0, `scipy.fft` is recommended over `scipy.fftpack`. Consider using `cupyx.scipy.fft` instead.

---



---

**Hint:** SciPy API Reference: Legacy discrete Fourier transforms (`scipy.fftpack`)

---

## Fast Fourier Transforms (FFTs)

<code>fft(x[, n, axis, overwrite_x, plan])</code>	Compute the one-dimensional FFT.
<code>ifft(x[, n, axis, overwrite_x, plan])</code>	Compute the one-dimensional inverse FFT.
<code>fft2(x[, shape, axes, overwrite_x, plan])</code>	Compute the two-dimensional FFT.
<code>ifft2(x[, shape, axes, overwrite_x, plan])</code>	Compute the two-dimensional inverse FFT.
<code>fftn(x[, shape, axes, overwrite_x, plan])</code>	Compute the N-dimensional FFT.
<code>ifftn(x[, shape, axes, overwrite_x, plan])</code>	Compute the N-dimensional inverse FFT.
<code>rfft(x[, n, axis, overwrite_x, plan])</code>	Compute the one-dimensional FFT for real input.
<code>irfft(x[, n, axis, overwrite_x])</code>	Compute the one-dimensional inverse FFT for real input.
<code>get_fft_plan(a[, shape, axes, value_type])</code>	Generate a CUDA FFT plan for transforming up to three axes.

### cupyx.scipy.fftpack.fft

`cupyx.scipy.fftpack.fft(x, n=None, axis=-1, overwrite_x=False, plan=None)`  
Compute the one-dimensional FFT.

#### Parameters

- **x** (`cupy.ndarray`) – Array to be transformed.
- **n** (`None` or `int`) – Length of the transformed axis of the output. If `n` is not given, the length of the input along the axis specified by `axis` is used.
- **axis** (`int`) – Axis over which to compute the FFT.
- **overwrite\_x** (`bool`) – If True, the contents of `x` can be destroyed.
- **plan** (`cupy.cuda.cufft.Plan1d` or `None`) – a cuFFT plan for transforming `x` over `axis`, which can be obtained using:

```
plan = cupyx.scipy.fftpack.get_fft_plan(x, axis)
```

Note that *plan* is defaulted to `None`, meaning CuPy will use an auto-generated plan behind the scene.

**Returns** The transformed array which shape is specified by `n` and type will convert to complex if that of the input is another.

**Return type** `cupy.ndarray`

---

**Note:** The argument *plan* is currently experimental and the interface may be changed in the future version.

---

See also:

`scipy.fftpack.fft()`

### cupyx.scipy.fftpack.ifft

`cupyx.scipy.fftpack.ifft(x, n=None, axis=-1, overwrite_x=False, plan=None)`  
Compute the one-dimensional inverse FFT.

#### Parameters

- **x** (`cupy.ndarray`) – Array to be transformed.
- **n** (`None` or `int`) – Length of the transformed axis of the output. If `n` is not given, the length of the input along the axis specified by `axis` is used.
- **axis** (`int`) – Axis over which to compute the FFT.
- **overwrite\_x** (`bool`) – If True, the contents of `x` can be destroyed.
- **plan** (`cupy.cuda.cufft.Plan1d` or `None`) – a cuFFT plan for transforming `x` over `axis`, which can be obtained using:

```
plan = cupyx.scipy.fftpack.get_fft_plan(x, axis)
```

Note that *plan* is defaulted to `None`, meaning CuPy will use an auto-generated plan behind the scene.



**Returns** The transformed array which shape is specified by `n` and type will convert to complex if that of the input is another.

**Return type** `cupy.ndarray`

---

**Note:** The argument `plan` is currently experimental and the interface may be changed in the future version.

---

**See also:**

`scipy.fftpack.ifft()`

## `cupyx.scipy.fftpack.fft2`

`cupyx.scipy.fftpack.fft2(x, shape=None, axes=(-2, -1), overwrite_x=False, plan=None)`

Compute the two-dimensional FFT.

### Parameters

- **x** (`cupy.ndarray`) – Array to be transformed.
- **shape** (`None` or *tuple of ints*) – Shape of the transformed axes of the output. If `shape` is not given, the lengths of the input along the axes specified by `axes` are used.
- **axes** (*tuple of ints*) – Axes over which to compute the FFT.
- **overwrite\_x** (`bool`) – If True, the contents of `x` can be destroyed.
- **plan** (`cupy.cuda.cufft.PlanNd` or `None`) – a cuFFT plan for transforming `x` over `axes`, which can be obtained using:

```
plan = cupyx.scipy.fftpack.get_fft_plan(x, axes)
```

Note that `plan` is defaulted to `None`, meaning CuPy will either use an auto-generated plan behind the scene if `cupy.fft.config.enable_nd_planning = True`, or use no cuFFT plan if it is set to `False`.

**Returns** The transformed array which shape is specified by `shape` and type will convert to complex if that of the input is another.

**Return type** `cupy.ndarray`

**See also:**

`scipy.fftpack.fft2()`

---

**Note:** The argument `plan` is currently experimental and the interface may be changed in the future version.

---

### cupyx.scipy.fftpack.iff2

`cupyx.scipy.fftpack.iff2(x, shape=None, axes=(-2, -1), overwrite_x=False, plan=None)`  
Compute the two-dimensional inverse FFT.

#### Parameters

- **x** (`cupy.ndarray`) – Array to be transformed.
- **shape** (*None or tuple of ints*) – Shape of the transformed axes of the output. If **shape** is not given, the lengths of the input along the axes specified by **axes** are used.
- **axes** (*tuple of ints*) – Axes over which to compute the FFT.
- **overwrite\_x** (*bool*) – If True, the contents of **x** can be destroyed.
- **plan** (`cupy.cuda.cufft.PlanNd` or *None*) – a cuFFT plan for transforming **x** over **axes**, which can be obtained using:

```
plan = cupyx.scipy.fftpack.get_fft_plan(x, axes)
```

Note that *plan* is defaulted to *None*, meaning CuPy will either use an auto-generated plan behind the scene if `cupy.fft.config.enable_nd_planning = True`, or use no cuFFT plan if it is set to *False*.

**Returns** The transformed array which shape is specified by **shape** and type will convert to complex if that of the input is another.

**Return type** `cupy.ndarray`

See also:

`scipy.fftpack.iff2()`

---

**Note:** The argument *plan* is currently experimental and the interface may be changed in the future version.

---

### cupyx.scipy.fftpack.fftn

`cupyx.scipy.fftpack.fftn(x, shape=None, axes=None, overwrite_x=False, plan=None)`  
Compute the N-dimensional FFT.

#### Parameters

- **x** (`cupy.ndarray`) – Array to be transformed.
- **shape** (*None or tuple of ints*) – Shape of the transformed axes of the output. If **shape** is not given, the lengths of the input along the axes specified by **axes** are used.
- **axes** (*tuple of ints*) – Axes over which to compute the FFT.
- **overwrite\_x** (*bool*) – If True, the contents of **x** can be destroyed.
- **plan** (`cupy.cuda.cufft.PlanNd` or *None*) – a cuFFT plan for transforming **x** over **axes**, which can be obtained using:

```
plan = cupyx.scipy.fftpack.get_fft_plan(x, axes)
```

Note that *plan* is defaulted to *None*, meaning CuPy will either use an auto-generated plan behind the scene if `cupy.fft.config.enable_nd_planning = True`, or use no cuFFT plan if it is set to *False*.

**Returns** The transformed array which shape is specified by `shape` and type will convert to complex if that of the input is another.

**Return type** `cupy.ndarray`

See also:

`scipy.fftpack.fftn()`

---

**Note:** The argument `plan` is currently experimental and the interface may be changed in the future version.

---

## `cupyx.scipy.fftpack.ifftn`

`cupyx.scipy.fftpack.ifftn(x, shape=None, axes=None, overwrite_x=False, plan=None)`

Compute the N-dimensional inverse FFT.

### Parameters

- **x** (`cupy.ndarray`) – Array to be transformed.
- **shape** (`None` or `tuple of ints`) – Shape of the transformed axes of the output. If `shape` is not given, the lengths of the input along the axes specified by `axes` are used.
- **axes** (`tuple of ints`) – Axes over which to compute the FFT.
- **overwrite\_x** (`bool`) – If True, the contents of `x` can be destroyed.
- **plan** (`cupy.cuda.cufft.PlanNd` or `None`) – a cuFFT plan for transforming `x` over `axes`, which can be obtained using:

```
plan = cupyx.scipy.fftpack.get_fft_plan(x, axes)
```

Note that `plan` is defaulted to `None`, meaning CuPy will either use an auto-generated plan behind the scene if `cupy.fft.config.enable_nd_planning = True`, or use no cuFFT plan if it is set to `False`.

**Returns** The transformed array which shape is specified by `shape` and type will convert to complex if that of the input is another.

**Return type** `cupy.ndarray`

See also:

`scipy.fftpack.ifftn()`

---

**Note:** The argument `plan` is currently experimental and the interface may be changed in the future version.

---

### cupyx.scipy.fftpack.rfft

`cupyx.scipy.fftpack.rfft(x, n=None, axis=-1, overwrite_x=False, plan=None)`

Compute the one-dimensional FFT for real input.

The returned real array contains

```
[y(0), Re(y(1)), Im(y(1)), ..., Re(y(n/2))] # if n is even
[y(0), Re(y(1)), Im(y(1)), ..., Re(y(n/2)), Im(y(n/2))] # if n is odd
```

#### Parameters

- **x** (`cupy.ndarray`) – Array to be transformed.
- **n** (`None` or `int`) – Length of the transformed axis of the output. If `n` is not given, the length of the input along the axis specified by `axis` is used.
- **axis** (`int`) – Axis over which to compute the FFT.
- **overwrite\_x** (`bool`) – If True, the contents of `x` can be destroyed.
- **plan** (`cupy.cuda.cufft.Plan1d` or `None`) – a cuFFT plan for transforming `x` over `axis`, which can be obtained using:

```
plan = cupyx.scipy.fftpack.get_fft_plan(
    x, axes, value_type='R2C')
```

Note that `plan` is defaulted to `None`, meaning CuPy will either use an auto-generated plan behind the scene if `cupy.fft.config.enable_nd_planning = True`, or use no cuFFT plan if it is set to `False`.

**Returns** The transformed array.

**Return type** `cupy.ndarray`

See also:

`scipy.fftpack.rfft()`

---

**Note:** The argument `plan` is currently experimental and the interface may be changed in the future version.

---

### cupyx.scipy.fftpack.irfft

`cupyx.scipy.fftpack.irfft(x, n=None, axis=-1, overwrite_x=False)`

Compute the one-dimensional inverse FFT for real input.

#### Parameters

- **x** (`cupy.ndarray`) – Array to be transformed.
- **n** (`None` or `int`) – Length of the transformed axis of the output. If `n` is not given, the length of the input along the axis specified by `axis` is used.
- **axis** (`int`) – Axis over which to compute the FFT.
- **overwrite\_x** (`bool`) – If True, the contents of `x` can be destroyed.

**Returns** The transformed array.

**Return type** `cupy.ndarray`

**See also:**

`scipy.fftpack.irfft()`

---

**Note:** This function does not support a precomputed *plan*. If you need this capability, please consider using `cupy.fft.irfft()` or `:func:`cupyx.scipy.fft.irfft``.

---

## `cupyx.scipy.fftpack.get_fft_plan`

`cupyx.scipy.fftpack.get_fft_plan(a, shape=None, axes=None, value_type='C2C')`

Generate a CUDA FFT plan for transforming up to three axes.

### Parameters

- **a** (`cupy.ndarray`) – Array to be transform, assumed to be either C- or F- contiguous.
- **shape** (*None* or *tuple of ints*) – Shape of the transformed axes of the output. If *shape* is not given, the lengths of the input along the axes specified by *axes* are used.
- **axes** (*None* or *int* or *tuple of int*) – The axes of the array to transform. If *None*, it is assumed that all axes are transformed.

Currently, for performing N-D transform these must be a set of up to three adjacent axes, and must include either the first or the last axis of the array.

- **value\_type** (*str*) – The FFT type to perform. Acceptable values are:
  - 'C2C': complex-to-complex transform (default)
  - 'R2C': real-to-complex transform
  - 'C2R': complex-to-real transform

**Returns** a cuFFT plan for either 1D transform (`cupy.cuda.cufft.Plan1d`) or N-D transform (`cupy.cuda.cufft.PlanNd`).

---

**Note:** The returned plan can not only be passed as one of the arguments of the functions in `cupyx.scipy.fftpack`, but also be used as a context manager for both `cupy.fft` and `cupyx.scipy.fftpack` functions:

```
x = cupy.random.random(16).reshape(4, 4).astype(complex)
plan = cupyx.scipy.fftpack.get_fft_plan(x)
with plan:
    y = cupy.fft.fftn(x)
    # alternatively:
    y = cupyx.scipy.fftpack.fftn(x) # no explicit plan is given!
# alternatively:
y = cupyx.scipy.fftpack.fftn(x, plan=plan) # pass plan explicitly
```

In the first case, no cuFFT plan will be generated automatically, even if `cupy.fft.config.enable_nd_planning = True` is set.

---

**Note:** If this function is called under the context of `set_cufft_callbacks()`, the generated plan will have callbacks enabled.

---

**Warning:** This API is a deviation from SciPy's, is currently experimental, and may be changed in the future version.

## Code compatibility features

1. As with other FFT modules in CuPy, FFT functions in this module can take advantage of an existing cuFFT plan (returned by `get_fft_plan()`) to accelerate the computation. The plan can be either passed in explicitly via the `plan` argument or used as a context manager. The argument `plan` is currently experimental and the interface may be changed in the future version. The `get_fft_plan()` function has no counterpart in `scipy.fftpack`.
2. The boolean switch `cupy.fft.config.enable_nd_planning` also affects the FFT functions in this module, see *Discrete Fourier Transform (cupy.fft)*. This switch is neglected when planning manually using `get_fft_plan()`.
3. Like in `scipy.fftpack`, all FFT functions in this module have an optional argument `overwrite_x` (default is `False`), which has the same semantics as in `scipy.fftpack`: when it is set to `True`, the input array `x` *can* (not *will*) be overwritten arbitrarily. For this reason, when an in-place FFT is desired, the user should always reassign the input in the following manner: `x = cupyx.scipy.fftpack.fft(x, ..., overwrite_x=True, ...)`.
4. The boolean switch `cupy.fft.config.use_multi_gpus` also affects the FFT functions in this module, see *Discrete Fourier Transform (cupy.fft)*. Moreover, this switch is *honored* when planning manually using `get_fft_plan()`.

### 5.4.3 Linear algebra (cupyx.scipy.linalg)

**Hint:** SciPy API Reference: Linear algebra ([scipy.linalg](#))

#### Basics

<code>solve_triangular(a, b[, trans, lower, ...])</code>	Solve the equation $a x = b$ for $x$ , assuming $a$ is a triangular matrix.
<code>tril(m[, k])</code>	Make a copy of a matrix with elements above the $k$ -th diagonal zeroed.
<code>triu(m[, k])</code>	Make a copy of a matrix with elements below the $k$ -th diagonal zeroed.

#### cupyx.scipy.linalg.solve\_triangular

`cupyx.scipy.linalg.solve_triangular(a, b, trans=0, lower=False, unit_diagonal=False, overwrite_b=False, check_finite=False)`

Solve the equation  $a x = b$  for  $x$ , assuming  $a$  is a triangular matrix.

##### Parameters

- **a** (`cupy.ndarray`) – The matrix with dimension  $(M, M)$ .
- **b** (`cupy.ndarray`) – The matrix with dimension  $(M, )$  or  $(M, N)$ .
- **lower** (`bool`) – Use only data contained in the lower triangle of  $a$ . Default is to use upper triangle.

- **trans** (`0`, `1`, `2`, `'N'`, `'T'` or `'C'`) – Type of system to solve:
  - `'0'` or `'N'` –  $ax = b$
  - `'1'` or `'T'` –  $a^T x = b$
  - `'2'` or `'C'` –  $a^H x = b$
- **unit\_diagonal** (*bool*) – If `True`, diagonal elements of `a` are assumed to be 1 and will not be referenced.
- **overwrite\_b** (*bool*) – Allow overwriting data in `b` (may enhance performance)
- **check\_finite** (*bool*) – Whether to check that the input matrices contain only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs.

**Returns** The matrix with dimension  $(M,)$  or  $(M, N)$ .

**Return type** *cupy.ndarray*

See also:

`scipy.linalg.solve_triangular()`

## cupyx.scipy.linalg.tril

`cupyx.scipy.linalg.tril(m, k=0)`

Make a copy of a matrix with elements above the `k`-th diagonal zeroed.

### Parameters

- **m** (*cupy.ndarray*) – Matrix whose elements to return
- **k** (*int*, *optional*) – Diagonal above which to zero elements. `k == 0` is the main diagonal, `k < 0` subdiagonal and `k > 0` superdiagonal.

**Returns** Return is the same shape and type as `m`.

**Return type** (*cupy.ndarray*)

See also:

`scipy.linalg.tril()`

## cupyx.scipy.linalg.triu

`cupyx.scipy.linalg.triu(m, k=0)`

Make a copy of a matrix with elements below the `k`-th diagonal zeroed.

### Parameters

- **m** (*cupy.ndarray*) – Matrix whose elements to return
- **k** (*int*, *optional*) – Diagonal above which to zero elements. `k == 0` is the main diagonal, `k < 0` subdiagonal and `k > 0` superdiagonal.

**Returns** Return matrix with zeroed elements below the `k`th diagonal and has same shape and type as `m`.

**Return type** (*cupy.ndarray*)

See also:

`scipy.linalg.triu()`

## Decompositions

<code>lu(a[, permute_l, overwrite_a, check_finite])</code>	LU decomposition.
<code>lu_factor(a[, overwrite_a, check_finite])</code>	LU decomposition.
<code>lu_solve(lu_and_piv, b[, trans, ...])</code>	Solve an equation system, $a * x = b$ , given the LU factorization of $a$

### cupyx.scipy.linalg.lu

`cupyx.scipy.linalg.lu(a, permute_l=False, overwrite_a=False, check_finite=True)`  
LU decomposition.

Decomposes a given two-dimensional matrix into  $P @ L @ U$ , where  $P$  is a permutation matrix,  $L$  is a lower triangular or trapezoidal matrix with unit diagonal, and  $U$  is a upper triangular or trapezoidal matrix.

#### Parameters

- **a** (`cupy.ndarray`) – The input matrix with dimension (M, N).
- **permute\_l** (`bool`) – If True, perform the multiplication  $P @ L$ .
- **overwrite\_a** (`bool`) – Allow overwriting data in a (may enhance performance)
- **check\_finite** (`bool`) – Whether to check that the input matrices contain only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs.

**Returns** ( $P$ ,  $L$ ,  $U$ ) if `permute_l == False`, otherwise ( $PL$ ,  $U$ ).  $P$  is a `cupy.ndarray` storing permutation matrix with dimension (M, M).  $L$  is a `cupy.ndarray` storing lower triangular or trapezoidal matrix with unit diagonal with dimension (M, K) where  $K = \min(M, N)$ .  $U$  is a `cupy.ndarray` storing upper triangular or trapezoidal matrix with dimension (K, N).  $PL$  is a `cupy.ndarray` storing permuted  $L$  matrix with dimension (M, K).

**Return type** `tuple`

See also:

`scipy.linalg.lu()`

### cupyx.scipy.linalg.lu\_factor

`cupyx.scipy.linalg.lu_factor(a, overwrite_a=False, check_finite=True)`  
LU decomposition.

Decompose a given two-dimensional square matrix into  $P * L * U$ , where  $P$  is a permutation matrix,  $L$  lower-triangular with unit diagonal elements, and  $U$  upper-triangular matrix.

#### Parameters

- **a** (`cupy.ndarray`) – The input matrix with dimension (M, N)
- **overwrite\_a** (`bool`) – Allow overwriting data in a (may enhance performance)



- **check\_finite** (*bool*) – Whether to check that the input matrices contain only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs.

**Returns** (lu, piv) where lu is a [cupy.ndarray](#) storing U in its upper triangle, and L without unit diagonal elements in its lower triangle, and piv is a [cupy.ndarray](#) storing pivot indices representing permutation matrix P. For  $0 \leq i < \min(M, N)$ , row i of the matrix was interchanged with row piv[i]

**Return type** tuple

See also:

`scipy.linalg.lu_factor()`

## cupyx.scipy.linalg.lu\_solve

`cupyx.scipy.linalg.lu_solve(lu_and_piv, b, trans=0, overwrite_b=False, check_finite=True)`

Solve an equation system,  $a * x = b$ , given the LU factorization of a

**Parameters**

- **lu\_and\_piv** (*tuple*) – LU factorization of matrix a ((M, M)) together with pivot indices.
- **b** ([cupy.ndarray](#)) – The matrix with dimension (M,) or (M, N).
- **trans** ({0, 1, 2}) – Type of system to solve:

trans	system
0	$a x = b$
1	$a^T x = b$
2	$a^H x = b$

- **overwrite\_b** (*bool*) – Allow overwriting data in b (may enhance performance)
- **check\_finite** (*bool*) – Whether to check that the input matrices contain only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs.

**Returns** The matrix with dimension (M,) or (M, N).

**Return type** [cupy.ndarray](#)

See also:

`scipy.linalg.lu_solve()`

## Special Matrices

<a href="#">block_diag</a> (*arrs)	Create a block diagonal matrix from provided arrays.
<a href="#">circulant</a> (c)	Construct a circulant matrix.
<a href="#">companion</a> (a)	Create a companion matrix.
<a href="#">convolution_matrix</a> (a, n[, mode])	Construct a convolution matrix.
<a href="#">dft</a> (n[, scale])	Discrete Fourier transform matrix.
<a href="#">fiedler</a> (a)	Returns a symmetric Fiedler matrix
<a href="#">fiedler_companion</a> (a)	Returns a Fiedler companion matrix

continues on next page

Table 98 – continued from previous page

<code>hadamard(n[, dtype])</code>	Construct an Hadamard matrix.
<code>hankel(c[, r])</code>	Construct a Hankel matrix.
<code>helmert(n[, full])</code>	Create an Helmert matrix of order <code>n</code> .
<code>hilbert(n)</code>	Create a Hilbert matrix of order <code>n</code> .
<code>kron(a, b)</code>	Kronecker product.
<code>leslie(f, s)</code>	Create a Leslie matrix.
<code>toeplitz(c[, r])</code>	Construct a Toeplitz matrix.
<code>tri(N[, M, k, dtype])</code>	Construct (N, M) matrix filled with ones at and below the <code>k</code> -th diagonal.

**cupyx.scipy.linalg.block\_diag****cupyx.scipy.linalg.block\_diag**(\*arrs)

Create a block diagonal matrix from provided arrays.

Given the inputs A, B, and C, the output will have these arrays arranged on the diagonal:

```
[A, 0, 0]
[0, B, 0]
[0, 0, C]
```

**Parameters**

- **A** (`cupy.ndarray`) – Input arrays. A 1-D array of length `n` is treated as a 2-D array with shape `(1,n)`.
- **B** (`cupy.ndarray`) – Input arrays. A 1-D array of length `n` is treated as a 2-D array with shape `(1,n)`.
- **C** (`cupy.ndarray`) – Input arrays. A 1-D array of length `n` is treated as a 2-D array with shape `(1,n)`.
- ... (`cupy.ndarray`) – Input arrays. A 1-D array of length `n` is treated as a 2-D array with shape `(1,n)`.

**Returns** Array with A, B, C, ... on the diagonal. Output has the same dtype as A.**Return type** (`cupy.ndarray`)

See also:

`scipy.linalg.block_diag()`**cupyx.scipy.linalg.circulant****cupyx.scipy.linalg.circulant**(c)

Construct a circulant matrix.

**Parameters** `c` (`cupy.ndarray`) – 1-D array, the first column of the matrix.**Returns** A circulant matrix whose first column is `c`.**Return type** `cupy.ndarray`

See also:

`cupyx.scipy.linalg.toeplitz()`

See also:

`cupyx.scipy.linalg.hankel()`

See also:

`cupyx.scipy.linalg.solve_circulant()`

See also:

`cupyx.scipy.linalg.fiedler()`

See also:

`scipy.linalg.circulant()`

### `cupyx.scipy.linalg.companion`

`cupyx.scipy.linalg.companion(a)`

Create a companion matrix.

Create the companion matrix associated with the polynomial whose coefficients are given in `a`.

**Parameters** `a` (`cupy.ndarray`) – 1-D array of polynomial coefficients. The length of `a` must be at least two, and `a[0]` must not be zero.

**Returns** The first row of the output is  $-a[1:]/a[0]$ , and the first sub-diagonal is all ones. The data-type of the array is the same as the data-type of  $-a[1:]/a[0]$ .

**Return type** (`cupy.ndarray`)

See also:

`cupyx.scipy.linalg.fiedler_companion()`

See also:

`scipy.linalg.companion()`

### `cupyx.scipy.linalg.convolution_matrix`

`cupyx.scipy.linalg.convolution_matrix(a, n, mode='full')`

Construct a convolution matrix.

Constructs the Toeplitz matrix representing one-dimensional convolution.

**Parameters**

- `a` (`cupy.ndarray`) – The 1-D array to convolve.
- `n` (`int`) – The number of columns in the resulting matrix. It gives the length of the input to be convolved with `a`. This is analogous to the length of `v` in `numpy.convolve(a, v)`.
- `mode` (`str`) – This must be one of ('full', 'valid', 'same'). This is analogous to `mode` in `numpy.convolve(v, a, mode)`.

**Returns**

The convolution matrix whose row count `k` depends on `mode`:

mode	k
'full'	$m + n - 1$
'same'	$\max(m, n)$
'valid'	$\max(m, n) - \min(m, n) + 1$

**Return type** *cupy.ndarray*

**See also:**

*cupyx.scipy.linalg.toeplitz()*

**See also:**

*scipy.linalg.convolution\_matrix()*

## cupyx.scipy.linalg.dft

*cupyx.scipy.linalg.dft*(*n*, *scale=None*)

Discrete Fourier transform matrix.

Create the matrix that computes the discrete Fourier transform of a sequence. The *n*th primitive root of unity used to generate the matrix is  $\exp(-2\pi i/n)$ , where  $i = \sqrt{-1}$ .

**Parameters**

- **n** (*int*) – Size the matrix to create.
- **scale** (*str*, *optional*) – Must be None, 'sqtrn', or 'n'. If *scale* is 'sqtrn', the matrix is divided by  $\sqrt{n}$ . If *scale* is 'n', the matrix is divided by *n*. If *scale* is None (default), the matrix is not normalized, and the return value is simply the Vandermonde matrix of the roots of unity.

**Returns** The DFT matrix.

**Return type** (*cupy.ndarray*)

### Notes

When *scale* is None, multiplying a vector by the matrix returned by *dft* is mathematically equivalent to (but much less efficient than) the calculation performed by *scipy.fft.fft*.

**See also:**

*scipy.linalg.dft()*

## cupyx.scipy.linalg.fiedler

*cupyx.scipy.linalg.fiedler*(*a*)

Returns a symmetric Fiedler matrix

Given an sequence of numbers *a*, Fiedler matrices have the structure  $F[i, j] = \text{np.abs}(a[i] - a[j])$ , and hence zero diagonals and nonnegative entries. A Fiedler matrix has a dominant positive eigenvalue and other eigenvalues are negative. Although not valid generally, for certain inputs, the inverse and the determinant can be derived explicitly.

**Parameters** *a* (*cupy.ndarray*) – coefficient array

**Returns** the symmetric Fiedler matrix

**Return type** *cupy.ndarray*

**See also:**

*cupyx.scipy.linalg.circulant()*

**See also:**

*cupyx.scipy.linalg.toeplitz()*

**See also:**

*scipy.linalg.fiedler()*

### cupyx.scipy.linalg.fiedler\_companion

`cupyx.scipy.linalg.fiedler_companion(a)`

Returns a Fiedler companion matrix

Given a polynomial coefficient array *a*, this function forms a pentadiagonal matrix with a special structure whose eigenvalues coincides with the roots of *a*.

**Parameters** *a* (*cupy.ndarray*) – 1-D array of polynomial coefficients in descending order with a nonzero leading coefficient. For  $N < 2$ , an empty array is returned.

**Returns** Resulting companion matrix

**Return type** *cupy.ndarray*

### Notes

Similar to `companion` the leading coefficient should be nonzero. In the case the leading coefficient is not 1, other coefficients are rescaled before the array generation. To avoid numerical issues, it is best to provide a monic polynomial.

**See also:**

*cupyx.scipy.linalg.companion()*

**See also:**

*scipy.linalg.fiedler\_companion()*

### cupyx.scipy.linalg.hadamard

`cupyx.scipy.linalg.hadamard(n, dtype=<class 'int'>)`

Construct an Hadamard matrix.

Constructs an *n*-by-*n* Hadamard matrix, using Sylvester's construction. *n* must be a power of 2.

**Parameters**

- **n** (*int*) – The order of the matrix. *n* must be a power of 2.
- **dtype** (*dtype*, *optional*) – The data type of the array to be constructed.

**Returns** The Hadamard matrix.

**Return type** (*cupy.ndarray*)

See also:

`scipy.linalg.hadamard()`

### `cupyx.scipy.linalg.hankel`

`cupyx.scipy.linalg.hankel(c, r=None)`

Construct a Hankel matrix.

The Hankel matrix has constant anti-diagonals, with `c` as its first column and `r` as its last row. If `r` is not given, then `r = zeros_like(c)` is assumed.

#### Parameters

- **c** (`cupy.ndarray`) – First column of the matrix. Whatever the actual shape of `c`, it will be converted to a 1-D array.
- **r** (`cupy.ndarray`, *optional*) – Last row of the matrix. If `None`, `r = zeros_like(c)` is assumed. `r[0]` is ignored; the last row of the returned matrix is `[c[-1], r[1:]]`. Whatever the actual shape of `r`, it will be converted to a 1-D array.

**Returns** The Hankel matrix. Dtype is the same as `(c[0] + r[0]).dtype`.

**Return type** `cupy.ndarray`

See also:

`cupyx.scipy.linalg.toeplitz()`

See also:

`cupyx.scipy.linalg.circulant()`

See also:

`scipy.linalg.hankel()`

### `cupyx.scipy.linalg.helmert`

`cupyx.scipy.linalg.helmert(n, full=False)`

Create an Helmert matrix of order `n`.

This has applications in statistics, compositional or simplicial analysis, and in Aitchison geometry.

#### Parameters

- **n** (`int`) – The size of the array to create.
- **full** (`bool`, *optional*) – If `True` the `(n, n)` ndarray will be returned. Otherwise, the default, the submatrix that does not include the first row will be returned.

**Returns** The Helmert matrix. The shape is `(n, n)` or `(n-1, n)` depending on the `full` argument.

**Return type** `cupy.ndarray`

See also:

`scipy.linalg.helmert()`

**cupyx.scipy.linalg.hilbert****cupyx.scipy.linalg.hilbert**(*n*)Create a Hilbert matrix of order *n*.Returns the *n* by *n* array with entries  $h[i, j] = 1 / (i + j + 1)$ .**Parameters** *n* (*int*) – The size of the array to create.**Returns** The Hilbert matrix.**Return type** *cupy.ndarray***See also:**`scipy.linalg.hilbert()`**cupyx.scipy.linalg.kron****cupyx.scipy.linalg.kron**(*a*, *b*)

Kronecker product.

**The result is the block matrix::**  $a[0,0]*b \ a[0,1]*b \ \dots \ a[0,-1]*b \ a[1,0]*b \ a[1,1]*b \ \dots \ a[1,-1]*b \ \dots \ a[-1,0]*b$   
 $a[-1,1]*b \ \dots \ a[-1,-1]*b$ **Parameters**

- **a** (*cupy.ndarray*) – Input array
- **b** (*cupy.ndarray*) – Input array

**Returns** Kronecker product of *a* and *b*.**Return type** *cupy.ndarray***See also:**`scipy.linalg.kron()`**cupyx.scipy.linalg.leslie****cupyx.scipy.linalg.leslie**(*f*, *s*)

Create a Leslie matrix.

Given the length *n* array of fecundity coefficients *f* and the length *n*-1 array of survival coefficients *s*, return the associated Leslie matrix.**Parameters**

- **f** (*cupy.ndarray*) – The “fecundity” coefficients.
- **s** (*cupy.ndarray*) – The “survival” coefficients, has to be 1-D. The length of *s* must be one less than the length of *f*, and it must be at least 1.

**Returns** The array is zero except for the first row, which is *f*, and the first sub-diagonal, which is *s*. The data-type of the array will be the data-type of `f[0]+s[0]`.**Return type** *cupy.ndarray***See also:**`scipy.linalg.leslie()`

### cupyx.scipy.linalg.toeplitz

cupyx.scipy.linalg.toeplitz(*c*, *r=None*)

Construct a Toeplitz matrix.

The Toeplitz matrix has constant diagonals, with *c* as its first column and *r* as its first row. If *r* is not given, *r* == conjugate(*c*) is assumed.

#### Parameters

- ***c*** ([cupy.ndarray](#)) – First column of the matrix. Whatever the actual shape of *c*, it will be converted to a 1-D array.
- ***r*** ([cupy.ndarray](#), *optional*) – First row of the matrix. If *None*, *r* = conjugate(*c*) is assumed; in this case, if *c*[0] is real, the result is a Hermitian matrix. *r*[0] is ignored; the first row of the returned matrix is [*c*[0], *r*[1:]]. Whatever the actual shape of *r*, it will be converted to a 1-D array.

**Returns** The Toeplitz matrix. Dtype is the same as (*c*[0] + *r*[0]).dtype.

**Return type** [cupy.ndarray](#)

See also:

[cupyx.scipy.linalg.circulant\(\)](#)

See also:

[cupyx.scipy.linalg.hankel\(\)](#)

See also:

[cupyx.scipy.linalg.solve\\_toeplitz\(\)](#)

See also:

[cupyx.scipy.linalg.fiedler\(\)](#)

See also:

[scipy.linalg.toeplitz\(\)](#)

### cupyx.scipy.linalg.tri

cupyx.scipy.linalg.tri(*N*, *M=None*, *k=0*, *dtype=None*)

Construct (*N*, *M*) matrix filled with ones at and below the *k*-th diagonal. The matrix has  $A[i, j] == 1$  for  $i \leq j + k$ .

#### Parameters

- ***N*** (*int*) – The size of the first dimension of the matrix.
- ***M*** (*int*, *optional*) – The size of the second dimension of the matrix. If *M* is *None*, *M* = *N* is assumed.
- ***k*** (*int*, *optional*) – Number of subdiagonal below which matrix is filled with ones. *k* = 0 is the main diagonal, *k* < 0 subdiagonal and *k* > 0 superdiagonal.
- ***dtype*** (*dtype*, *optional*) – Data type of the matrix.

**Returns** Tri matrix.

**Return type** [cupy.ndarray](#)



See also:

`scipy.linalg.tri()`

## 5.4.4 Multidimensional image processing (`cupyx.scipy.ndimage`)

---

**Hint:** SciPy API Reference: Multidimensional image processing (`scipy.ndimage`)

---

### Filters

<code>convolve</code> (input, weights[, output, mode, ...])	Multi-dimensional convolution.
<code>convolve1d</code> (input, weights[, axis, output, ...])	One-dimensional convolution.
<code>correlate</code> (input, weights[, output, mode, ...])	Multi-dimensional correlate.
<code>correlate1d</code> (input, weights[, axis, output, ...])	One-dimensional correlate.
<code>gaussian_filter</code> (input, sigma[, order, ...])	Multi-dimensional Gaussian filter.
<code>gaussian_filter1d</code> (input, sigma[, axis, ...])	One-dimensional Gaussian filter along the given axis.
<code>gaussian_gradient_magnitude</code> (input, sigma[, ...])	Multi-dimensional gradient magnitude using Gaussian derivatives.
<code>gaussian_laplace</code> (input, sigma[, output, ...])	Multi-dimensional Laplace filter using Gaussian second derivatives.
<code>generic_filter</code> (input, function[, size, ...])	Compute a multi-dimensional filter using the provided raw kernel or reduction kernel.
<code>generic_filter1d</code> (input, function, filter_size)	Compute a 1D filter along the given axis using the provided raw kernel.
<code>generic_gradient_magnitude</code> (input, derivative)	Multi-dimensional gradient magnitude filter using a provided derivative function.
<code>generic_laplace</code> (input, derivative2[, ...])	Multi-dimensional Laplace filter using a provided second derivative function.
<code>laplace</code> (input[, output, mode, cval])	Multi-dimensional Laplace filter based on approximate second derivatives.
<code>maximum_filter</code> (input[, size, footprint, ...])	Multi-dimensional maximum filter.
<code>maximum_filter1d</code> (input, size[, axis, ...])	Compute the maximum filter along a single axis.
<code>median_filter</code> (input[, size, footprint, ...])	Multi-dimensional median filter.
<code>minimum_filter</code> (input[, size, footprint, ...])	Multi-dimensional minimum filter.
<code>minimum_filter1d</code> (input, size[, axis, ...])	Compute the minimum filter along a single axis.
<code>percentile_filter</code> (input, percentile[, size, ...])	Multi-dimensional percentile filter.
<code>prewitt</code> (input[, axis, output, mode, cval])	Compute a Prewitt filter along the given axis.
<code>rank_filter</code> (input, rank[, size, footprint, ...])	Multi-dimensional rank filter.
<code>sobel</code> (input[, axis, output, mode, cval])	Compute a Sobel filter along the given axis.
<code>uniform_filter</code> (input[, size, output, mode, ...])	Multi-dimensional uniform filter.
<code>uniform_filter1d</code> (input, size[, axis, ...])	One-dimensional uniform filter along the given axis.

### cupyx.scipy.ndimage.convolve

`cupyx.scipy.ndimage.convolve(input, weights, output=None, mode='reflect', cval=0.0, origin=0)`  
Multi-dimensional convolution.

The array is convolved with the given kernel.

#### Parameters

- **input** (`cupy.ndarray`) – The input array.
- **weights** (`cupy.ndarray`) – Array of weights, same number of dimensions as input
- **output** (`cupy.ndarray`, *dtype* or *None*) – The array in which to place the output.
- **mode** (*str*) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (*scalar*) – Value to fill past edges of input if mode is constant. Default is 0.0.
- **origin** (*scalar or tuple of scalar*) – The origin parameter controls the placement of the filter, relative to the center of the current element of the input. Default of 0 is equivalent to (0,)\*input.ndim.

**Returns** The result of convolution.

**Return type** `cupy.ndarray`

See also:

`scipy.ndimage.convolve()`

---

**Note:** When the output data type is integral (or when no output is provided and input is integral) the results may not perfectly match the results from SciPy due to floating-point rounding of intermediate results.

---

### cupyx.scipy.ndimage.convolve1d

`cupyx.scipy.ndimage.convolve1d(input, weights, axis=-1, output=None, mode='reflect', cval=0.0, origin=0)`  
One-dimensional convolution.

The array is convolved with the given kernel.

#### Parameters

- **input** (`cupy.ndarray`) – The input array.
- **weights** (`cupy.ndarray`) – One-dimensional array of weights
- **axis** (*int*) – The axis of input along which to calculate. Default is -1.
- **output** (`cupy.ndarray`, *dtype* or *None*) – The array in which to place the output. Default is is same dtype as the input.
- **mode** (*str*) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (*scalar*) – Value to fill past edges of input if mode is 'constant'. Default is 0.0.
- **origin** (*int*) – The origin parameter controls the placement of the filter, relative to the center of the current element of the input. Default is 0.

**Returns** The result of the 1D convolution.

**Return type** *cupy.ndarray*

**See also:**

`scipy.ndimage.convolve1d()`

---

**Note:** When the output data type is integral (or when no output is provided and input is integral) the results may not perfectly match the results from SciPy due to floating-point rounding of intermediate results.

---

### cupyx.scipy.ndimage.correlate

`cupyx.scipy.ndimage.correlate(input, weights, output=None, mode='reflect', cval=0.0, origin=0)`

Multi-dimensional correlate.

The array is correlated with the given kernel.

#### Parameters

- **input** (*cupy.ndarray*) – The input array.
- **weights** (*cupy.ndarray*) – Array of weights, same number of dimensions as input
- **output** (*cupy.ndarray*, *dtype* or *None*) – The array in which to place the output.
- **mode** (*str*) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (*scalar*) – Value to fill past edges of input if mode is constant. Default is 0.0.
- **origin** (*scalar* or *tuple of scalar*) – The origin parameter controls the placement of the filter, relative to the center of the current element of the input. Default of 0 is equivalent to (0,)\*input.ndim.

**Returns** The result of correlate.

**Return type** *cupy.ndarray*

**See also:**

`scipy.ndimage.correlate()`

---

**Note:** When the output data type is integral (or when no output is provided and input is integral) the results may not perfectly match the results from SciPy due to floating-point rounding of intermediate results.

---

### cupyx.scipy.ndimage.correlate1d

`cupyx.scipy.ndimage.correlate1d(input, weights, axis=-1, output=None, mode='reflect', cval=0.0, origin=0)`

One-dimensional correlate.

The array is correlated with the given kernel.

#### Parameters

- **input** (*cupy.ndarray*) – The input array.
- **weights** (*cupy.ndarray*) – One-dimensional array of weights
- **axis** (*int*) – The axis of input along which to calculate. Default is -1.

- **output** (`cupy.ndarray`, `dtype` or `None`) – The array in which to place the output. Default is is same dtype as the input.
- **mode** (`str`) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (`scalar`) – Value to fill past edges of input if mode is 'constant'. Default is 0.0.
- **origin** (`int`) – The origin parameter controls the placement of the filter, relative to the center of the current element of the input. Default is 0.

**Returns** The result of the 1D correlation.

**Return type** `cupy.ndarray`

See also:

`scipy.ndimage.correlate1d()`

---

**Note:** When the output data type is integral (or when no output is provided and input is integral) the results may not perfectly match the results from SciPy due to floating-point rounding of intermediate results.

---

## `cupyx.scipy.ndimage.gaussian_filter`

`cupyx.scipy.ndimage.gaussian_filter(input, sigma, order=0, output=None, mode='reflect', cval=0.0, truncate=4.0)`

Multi-dimensional Gaussian filter.

### Parameters

- **input** (`cupy.ndarray`) – The input array.
- **sigma** (`scalar` or `sequence of scalar`) – Standard deviations for each axis of Gaussian kernel. A single value applies to all axes.
- **order** (`int` or `sequence of scalar`) – An order of 0, the default, corresponds to convolution with a Gaussian kernel. A positive order corresponds to convolution with that derivative of a Gaussian. A single value applies to all axes.
- **output** (`cupy.ndarray`, `dtype` or `None`) – The array in which to place the output. Default is is same dtype as the input.
- **mode** (`str`) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (`scalar`) – Value to fill past edges of input if mode is 'constant'. Default is 0.0.
- **truncate** (`float`) – Truncate the filter at this many standard deviations. Default is 4.0.

**Returns** The result of the filtering.

**Return type** `cupy.ndarray`

See also:

`scipy.ndimage.gaussian_filter()`

---

**Note:** When the output data type is integral (or when no output is provided and input is integral) the results may not perfectly match the results from SciPy due to floating-point rounding of intermediate results.

---

### cupyx.scipy.ndimage.gaussian\_filter1d

`cupyx.scipy.ndimage.gaussian_filter1d(input, sigma, axis=-1, order=0, output=None, mode='reflect', cval=0.0, truncate=4.0)`

One-dimensional Gaussian filter along the given axis.

The lines of the array along the given axis are filtered with a Gaussian filter of the given standard deviation.

#### Parameters

- **input** (`cupy.ndarray`) – The input array.
- **sigma** (*scalar*) – Standard deviation for Gaussian kernel.
- **axis** (*int*) – The axis of input along which to calculate. Default is -1.
- **order** (*int*) – An order of 0, the default, corresponds to convolution with a Gaussian kernel. A positive order corresponds to convolution with that derivative of a Gaussian.
- **output** (`cupy.ndarray`, *dtype* or *None*) – The array in which to place the output. Default is same dtype as the input.
- **mode** (*str*) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (*scalar*) – Value to fill past edges of input if mode is 'constant'. Default is 0.0.
- **truncate** (*float*) – Truncate the filter at this many standard deviations. Default is 4.0.

**Returns** The result of the filtering.

**Return type** `cupy.ndarray`

See also:

`scipy.ndimage.gaussian_filter1d()`

---

**Note:** When the output data type is integral (or when no output is provided and input is integral) the results may not perfectly match the results from SciPy due to floating-point rounding of intermediate results.

---

### cupyx.scipy.ndimage.gaussian\_gradient\_magnitude

`cupyx.scipy.ndimage.gaussian_gradient_magnitude(input, sigma, output=None, mode='reflect', cval=0.0, **kwargs)`

Multi-dimensional gradient magnitude using Gaussian derivatives.

#### Parameters

- **input** (`cupy.ndarray`) – The input array.
- **sigma** (*scalar or sequence of scalar*) – Standard deviations for each axis of Gaussian kernel. A single value applies to all axes.
- **output** (`cupy.ndarray`, *dtype* or *None*) – The array in which to place the output. Default is same dtype as the input.
- **mode** (*str*) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (*scalar*) – Value to fill past edges of input if mode is 'constant'. Default is 0.0.

- **kwargs** (*dict, optional*) – dict of extra keyword arguments to pass `gaussian_filter()`.

**Returns** The result of the filtering.

**Return type** *cupy.ndarray*

See also:

`scipy.ndimage.gaussian_gradient_magnitude()`

---

**Note:** When the output data type is integral (or when no output is provided and input is integral) the results may not perfectly match the results from SciPy due to floating-point rounding of intermediate results.

---

### `cupyx.scipy.ndimage.gaussian_laplace`

`cupyx.scipy.ndimage.gaussian_laplace(input, sigma, output=None, mode='reflect', cval=0.0, **kwargs)`  
Multi-dimensional Laplace filter using Gaussian second derivatives.

#### Parameters

- **input** (*cupy.ndarray*) – The input array.
- **sigma** (*scalar or sequence of scalar*) – Standard deviations for each axis of Gaussian kernel. A single value applies to all axes.
- **output** (*cupy.ndarray, dtype or None*) – The array in which to place the output. Default is same dtype as the input.
- **mode** (*str*) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (*scalar*) – Value to fill past edges of input if mode is 'constant'. Default is 0.0.
- **kwargs** (*dict, optional*) – dict of extra keyword arguments to pass `gaussian_filter()`.

**Returns** The result of the filtering.

**Return type** *cupy.ndarray*

See also:

`scipy.ndimage.gaussian_laplace()`

---

**Note:** When the output data type is integral (or when no output is provided and input is integral) the results may not perfectly match the results from SciPy due to floating-point rounding of intermediate results.

---

**cupyx.scipy.ndimage.generic\_filter**

`cupyx.scipy.ndimage.generic_filter`(*input*, *function*, *size=None*, *footprint=None*, *output=None*, *mode='reflect'*, *cval=0.0*, *origin=0*)

Compute a multi-dimensional filter using the provided raw kernel or reduction kernel.

Unlike the `scipy.ndimage` function, this does not support the `extra_arguments` or `extra_keywordsdict` arguments and has significant restrictions on the `function` provided.

**Parameters**

- **input** (`cupy.ndarray`) – The input array.
- **function** (`cupy.ReductionKernel` or `cupy.RawKernel`) – The kernel or function to apply to each region.
- **size** (`int` or *sequence of int*) – One of `size` or `footprint` must be provided. If `footprint` is given, `size` is ignored. Otherwise `footprint = cupy.ones(size)` with `size` automatically made to match the number of dimensions in `input`.
- **footprint** (`cupy.ndarray`) – a boolean array which specifies which of the elements within this shape will get passed to the filter function.
- **output** (`cupy.ndarray`, *dtype* or `None`) – The array in which to place the output. Default is same dtype as the input.
- **mode** (`str`) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (*scalar*) – Value to fill past edges of input if mode is 'constant'. Default is 0.0.
- **origin** (*scalar or tuple of scalar*) – The origin parameter controls the placement of the filter, relative to the center of the current element of the input. Default of 0 is equivalent to (0,)\*input.ndim.

**Returns** The result of the filtering.

**Return type** `cupy.ndarray`

---

**Note:** If the *function* is a `cupy.RawKernel` then it must be for a function that has the following signature. Unlike most functions, this should not utilize `blockDim/blockIdx/threadIdx`:

```
__global__ void func(double *buffer, int filter_size,
                    double *return_value)
```

---

If the *function* is a `cupy.ReductionKernel` then it must be for a kernel that takes 1 array input and produces 1 'scalar' output.

---

**See also:**

`scipy.ndimage.generic_filter()`

### cupyx.scipy.ndimage.generic\_filter1d

cupyx.scipy.ndimage.**generic\_filter1d**(input, function, filter\_size, axis=-1, output=None, mode='reflect', cval=0.0, origin=0)

Compute a 1D filter along the given axis using the provided raw kernel.

Unlike the `scipy.ndimage` function, this does not support the `extra_arguments` or `extra_keywordsdict` arguments and has significant restrictions on the `function` provided.

#### Parameters

- **input** (`cupy.ndarray`) – The input array.
- **function** (`cupy.RawKernel`) – The kernel to apply along each axis.
- **filter\_size** (`int`) – Length of the filter.
- **axis** (`int`) – The axis of input along which to calculate. Default is -1.
- **output** (`cupy.ndarray`, `dtype` or `None`) – The array in which to place the output. Default is same dtype as the input.
- **mode** (`str`) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (`scalar`) – Value to fill past edges of input if mode is 'constant'. Default is 0.0.
- **origin** (`int`) – The origin parameter controls the placement of the filter, relative to the center of the current element of the input. Default is 0.

**Returns** The result of the filtering.

**Return type** `cupy.ndarray`

---

**Note:** The provided function (as a `RawKernel`) must have the following signature. Unlike most functions, this should not utilize `blockDim/blockIdx/threadIdx`:

```
__global__ void func(double *input_line, ptrdiff_t input_length,
                    double *output_line, ptrdiff_t output_length)
```

---

See also:

`scipy.ndimage.generic_filter1d()`

### cupyx.scipy.ndimage.generic\_gradient\_magnitude

cupyx.scipy.ndimage.**generic\_gradient\_magnitude**(input, derivative, output=None, mode='reflect', cval=0.0, extra\_arguments=(), extra\_keywords=None)

Multi-dimensional gradient magnitude filter using a provided derivative function.

#### Parameters

- **input** (`cupy.ndarray`) – The input array.
- **derivative** (`callable`) – Function or other callable with the following signature that is called once per axis:

```
derivative(input, axis, output, mode, cval,
          *extra_arguments, **extra_keywords)
```



where `input` and `output` are `cupy.ndarray`, `axis` is an `int` from 0 to the number of dimensions, and `mode`, `cval`, `extra_arguments`, `extra_keywords` are the values given to this function.

- **output** (`cupy.ndarray`, `dtype` or `None`) – The array in which to place the output. Default is same dtype as the input.
- **mode** (`str`) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (`scalar`) – Value to fill past edges of input if mode is 'constant'. Default is 0.0.
- **extra\_arguments** (`sequence`, `optional`) – Sequence of extra positional arguments to pass to `derivative2`.
- **extra\_keywords** (`dict`, `optional`) – dict of extra keyword arguments to pass to `derivative2`.

**Returns** The result of the filtering.

**Return type** `cupy.ndarray`

See also:

`scipy.ndimage.generic_gradient_magnitude()`

---

**Note:** When the output data type is integral (or when no output is provided and input is integral) the results may not perfectly match the results from SciPy due to floating-point rounding of intermediate results.

---

## cupyx.scipy.ndimage.generic\_laplace

`cupyx.scipy.ndimage.generic_laplace(input, derivative2, output=None, mode='reflect', cval=0.0, extra_arguments=(), extra_keywords=None)`

Multi-dimensional Laplace filter using a provided second derivative function.

### Parameters

- **input** (`cupy.ndarray`) – The input array.
- **derivative2** (`callable`) – Function or other callable with the following signature that is called once per axis:

```
derivative2(input, axis, output, mode, cval,
            *extra_arguments, **extra_keywords)
```

where `input` and `output` are `cupy.ndarray`, `axis` is an `int` from 0 to the number of dimensions, and `mode`, `cval`, `extra_arguments`, `extra_keywords` are the values given to this function.

- **output** (`cupy.ndarray`, `dtype` or `None`) – The array in which to place the output. Default is same dtype as the input.
- **mode** (`str`) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (`scalar`) – Value to fill past edges of input if mode is 'constant'. Default is 0.0.
- **extra\_arguments** (`sequence`, `optional`) – Sequence of extra positional arguments to pass to `derivative2`.

- **extra\_keywords** (*dict*, *optional*) – dict of extra keyword arguments to pass `derivative2`.

**Returns** The result of the filtering.

**Return type** *cupy.ndarray*

See also:

`scipy.ndimage.generic_laplace()`

---

**Note:** When the output data type is integral (or when no output is provided and input is integral) the results may not perfectly match the results from SciPy due to floating-point rounding of intermediate results.

---

### **cupyx.scipy.ndimage.laplace**

`cupyx.scipy.ndimage.laplace(input, output=None, mode='reflect', cval=0.0)`

Multi-dimensional Laplace filter based on approximate second derivatives.

#### **Parameters**

- **input** (*cupy.ndarray*) – The input array.
- **output** (*cupy.ndarray*, *dtype* or *None*) – The array in which to place the output. Default is same dtype as the input.
- **mode** (*str*) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (*scalar*) – Value to fill past edges of input if mode is 'constant'. Default is 0.0.

**Returns** The result of the filtering.

**Return type** *cupy.ndarray*

See also:

`scipy.ndimage.laplace()`

---

**Note:** When the output data type is integral (or when no output is provided and input is integral) the results may not perfectly match the results from SciPy due to floating-point rounding of intermediate results.

---

### **cupyx.scipy.ndimage.maximum\_filter**

`cupyx.scipy.ndimage.maximum_filter(input, size=None, footprint=None, output=None, mode='reflect', cval=0.0, origin=0)`

Multi-dimensional maximum filter.

#### **Parameters**

- **input** (*cupy.ndarray*) – The input array.
- **size** (*int* or *sequence of int*) – One of size or footprint must be provided. If footprint is given, size is ignored. Otherwise footprint = `cupy.ones(size)` with size automatically made to match the number of dimensions in input.

- **footprint** (`cupy.ndarray`) – a boolean array which specifies which of the elements within this shape will get passed to the filter function.
- **output** (`cupy.ndarray`, `dtype` or `None`) – The array in which to place the output. Default is same dtype as the input.
- **mode** (`str`) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (`scalar`) – Value to fill past edges of input if mode is 'constant'. Default is 0.0.
- **origin** (`int` or *sequence of int*) – The origin parameter controls the placement of the filter, relative to the center of the current element of the input. Default of 0 is equivalent to (0,)\*input.ndim.

**Returns** The result of the filtering.

**Return type** `cupy.ndarray`

See also:

`scipy.ndimage.maximum_filter()`

### `cupyx.scipy.ndimage.maximum_filter1d`

`cupyx.scipy.ndimage.maximum_filter1d(input, size, axis=-1, output=None, mode='reflect', cval=0.0, origin=0)`

Compute the maximum filter along a single axis.

#### Parameters

- **input** (`cupy.ndarray`) – The input array.
- **size** (`int`) – Length of the maximum filter.
- **axis** (`int`) – The axis of input along which to calculate. Default is -1.
- **output** (`cupy.ndarray`, `dtype` or `None`) – The array in which to place the output. Default is same dtype as the input.
- **mode** (`str`) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (`scalar`) – Value to fill past edges of input if mode is 'constant'. Default is 0.0.
- **origin** (`int`) – The origin parameter controls the placement of the filter, relative to the center of the current element of the input. Default is 0.

**Returns** The result of the filtering.

**Return type** `cupy.ndarray`

See also:

`scipy.ndimage.maximum_filter1d()`

### cupyx.scipy.ndimage.median\_filter

`cupyx.scipy.ndimage.median_filter(input, size=None, footprint=None, output=None, mode='reflect', cval=0.0, origin=0)`

Multi-dimensional median filter.

#### Parameters

- **input** (`cupy.ndarray`) – The input array.
- **size** (`int` or *sequence of int*) – One of size or footprint must be provided. If footprint is given, size is ignored. Otherwise `footprint = cupy.ones(size)` with size automatically made to match the number of dimensions in `input`.
- **footprint** (`cupy.ndarray`) – a boolean array which specifies which of the elements within this shape will get passed to the filter function.
- **output** (`cupy.ndarray`, `dtype` or `None`) – The array in which to place the output. Default is is same dtype as the input.
- **mode** (`str`) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (*scalar*) – Value to fill past edges of input if mode is 'constant'. Default is 0.0.
- **origin** (`int` or *sequence of int*) – The origin parameter controls the placement of the filter, relative to the center of the current element of the input. Default of 0 is equivalent to (0,)\*input.ndim.

**Returns** The result of the filtering.

**Return type** `cupy.ndarray`

See also:

`scipy.ndimage.median_filter()`

### cupyx.scipy.ndimage.minimum\_filter

`cupyx.scipy.ndimage.minimum_filter(input, size=None, footprint=None, output=None, mode='reflect', cval=0.0, origin=0)`

Multi-dimensional minimum filter.

#### Parameters

- **input** (`cupy.ndarray`) – The input array.
- **size** (`int` or *sequence of int*) – One of size or footprint must be provided. If footprint is given, size is ignored. Otherwise `footprint = cupy.ones(size)` with size automatically made to match the number of dimensions in `input`.
- **footprint** (`cupy.ndarray`) – a boolean array which specifies which of the elements within this shape will get passed to the filter function.
- **output** (`cupy.ndarray`, `dtype` or `None`) – The array in which to place the output. Default is is same dtype as the input.
- **mode** (`str`) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (*scalar*) – Value to fill past edges of input if mode is 'constant'. Default is 0.0.

- **origin** (*int* or *sequence of int*) – The origin parameter controls the placement of the filter, relative to the center of the current element of the input. Default of 0 is equivalent to `(0,)*input.ndim`.

**Returns** The result of the filtering.

**Return type** *cupy.ndarray*

See also:

`scipy.ndimage.minimum_filter()`

### cupyx.scipy.ndimage.minimum\_filter1d

`cupyx.scipy.ndimage.minimum_filter1d(input, size, axis=-1, output=None, mode='reflect', cval=0.0, origin=0)`

Compute the minimum filter along a single axis.

#### Parameters

- **input** (*cupy.ndarray*) – The input array.
- **size** (*int*) – Length of the minimum filter.
- **axis** (*int*) – The axis of input along which to calculate. Default is -1.
- **output** (*cupy.ndarray*, *dtype* or *None*) – The array in which to place the output. Default is same dtype as the input.
- **mode** (*str*) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (*scalar*) – Value to fill past edges of input if mode is 'constant'. Default is 0.0.
- **origin** (*int*) – The origin parameter controls the placement of the filter, relative to the center of the current element of the input. Default is 0.

**Returns** The result of the filtering.

**Return type** *cupy.ndarray*

See also:

`scipy.ndimage.minimum_filter1d()`

### cupyx.scipy.ndimage.percentile\_filter

`cupyx.scipy.ndimage.percentile_filter(input, percentile, size=None, footprint=None, output=None, mode='reflect', cval=0.0, origin=0)`

Multi-dimensional percentile filter.

#### Parameters

- **input** (*cupy.ndarray*) – The input array.
- **percentile** (*scalar*) – The percentile of the element to get (from 0 to 100). Can be negative, thus -20 equals 80.
- **size** (*int* or *sequence of int*) – One of size or footprint must be provided. If footprint is given, size is ignored. Otherwise footprint = `cupy.ones(size)` with size automatically made to match the number of dimensions in input.

- **footprint** (`cupy.ndarray`) – a boolean array which specifies which of the elements within this shape will get passed to the filter function.
- **output** (`cupy.ndarray`, `dtype` or `None`) – The array in which to place the output. Default is is same dtype as the input.
- **mode** (`str`) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (`scalar`) – Value to fill past edges of input if mode is 'constant'. Default is 0.0.
- **origin** (`int` or *sequence of int*) – The origin parameter controls the placement of the filter, relative to the center of the current element of the input. Default of 0 is equivalent to (0,)\*input.ndim.

**Returns** The result of the filtering.

**Return type** `cupy.ndarray`

See also:

`scipy.ndimage.percentile_filter()`

### `cupyx.scipy.ndimage.prewitt`

`cupyx.scipy.ndimage.prewitt(input, axis=-1, output=None, mode='reflect', cval=0.0)`

Compute a Prewitt filter along the given axis.

#### Parameters

- **input** (`cupy.ndarray`) – The input array.
- **axis** (`int`) – The axis of input along which to calculate. Default is -1.
- **output** (`cupy.ndarray`, `dtype` or `None`) – The array in which to place the output. Default is is same dtype as the input.
- **mode** (`str`) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (`scalar`) – Value to fill past edges of input if mode is 'constant'. Default is 0.0.

**Returns** The result of the filtering.

**Return type** `cupy.ndarray`

See also:

`scipy.ndimage.prewitt()`

---

**Note:** When the output data type is integral (or when no output is provided and input is integral) the results may not perfectly match the results from SciPy due to floating-point rounding of intermediate results.

---

**cupyx.scipy.ndimage.rank\_filter**

`cupyx.scipy.ndimage.rank_filter(input, rank, size=None, footprint=None, output=None, mode='reflect', cval=0.0, origin=0)`

Multi-dimensional rank filter.

**Parameters**

- **input** (`cupy.ndarray`) – The input array.
- **rank** (`int`) – The rank of the element to get. Can be negative to count from the largest value, e.g. -1 indicates the largest value.
- **size** (`int` or *sequence of int*) – One of `size` or `footprint` must be provided. If `footprint` is given, `size` is ignored. Otherwise `footprint = cupy.ones(size)` with `size` automatically made to match the number of dimensions in `input`.
- **footprint** (`cupy.ndarray`) – a boolean array which specifies which of the elements within this shape will get passed to the filter function.
- **output** (`cupy.ndarray`, `dtype` or `None`) – The array in which to place the output. Default is same dtype as the input.
- **mode** (`str`) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (*scalar*) – Value to fill past edges of input if mode is 'constant'. Default is 0.0.
- **origin** (`int` or *sequence of int*) – The origin parameter controls the placement of the filter, relative to the center of the current element of the input. Default of 0 is equivalent to (0,)\*input.ndim.

**Returns** The result of the filtering.

**Return type** `cupy.ndarray`

See also:

`scipy.ndimage.rank_filter()`

**cupyx.scipy.ndimage.sobel**

`cupyx.scipy.ndimage.sobel(input, axis=-1, output=None, mode='reflect', cval=0.0)`

Compute a Sobel filter along the given axis.

**Parameters**

- **input** (`cupy.ndarray`) – The input array.
- **axis** (`int`) – The axis of input along which to calculate. Default is -1.
- **output** (`cupy.ndarray`, `dtype` or `None`) – The array in which to place the output. Default is same dtype as the input.
- **mode** (`str`) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (*scalar*) – Value to fill past edges of input if mode is 'constant'. Default is 0.0.

**Returns** The result of the filtering.

**Return type** `cupy.ndarray`

See also:

`scipy.ndimage.sobel()`

---

**Note:** When the output data type is integral (or when no output is provided and input is integral) the results may not perfectly match the results from SciPy due to floating-point rounding of intermediate results.

---

### `cupyx.scipy.ndimage.uniform_filter`

`cupyx.scipy.ndimage.uniform_filter(input, size=3, output=None, mode='reflect', cval=0.0, origin=0)`  
Multi-dimensional uniform filter.

#### Parameters

- **input** (`cupy.ndarray`) – The input array.
- **size** (`int` or *sequence of int*) – Lengths of the uniform filter for each dimension. A single value applies to all axes.
- **output** (`cupy.ndarray`, `dtype` or `None`) – The array in which to place the output. Default is same dtype as the input.
- **mode** (`str`) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (*scalar*) – Value to fill past edges of input if mode is 'constant'. Default is 0.0.
- **origin** (`int` or *sequence of int*) – The origin parameter controls the placement of the filter, relative to the center of the current element of the input. Default of 0 is equivalent to (0,)\*input.ndim.

**Returns** The result of the filtering.

**Return type** `cupy.ndarray`

See also:

`scipy.ndimage.uniform_filter()`

---

**Note:** When the output data type is integral (or when no output is provided and input is integral) the results may not perfectly match the results from SciPy due to floating-point rounding of intermediate results.

---

### `cupyx.scipy.ndimage.uniform_filter1d`

`cupyx.scipy.ndimage.uniform_filter1d(input, size, axis=-1, output=None, mode='reflect', cval=0.0, origin=0)`

One-dimensional uniform filter along the given axis.

The lines of the array along the given axis are filtered with a uniform filter of the given size.

#### Parameters

- **input** (`cupy.ndarray`) – The input array.
- **size** (`int`) – Length of the uniform filter.
- **axis** (`int`) – The axis of input along which to calculate. Default is -1.



- **output** (`cupy.ndarray`, `dtype` or `None`) – The array in which to place the output. Default is same dtype as the input.
- **mode** (`str`) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (`scalar`) – Value to fill past edges of input if mode is 'constant'. Default is 0.0.
- **origin** (`int`) – The origin parameter controls the placement of the filter, relative to the center of the current element of the input. Default is 0.

**Returns** The result of the filtering.

**Return type** `cupy.ndarray`

See also:

`scipy.ndimage.uniform_filter1d()`

---

**Note:** When the output data type is integral (or when no output is provided and input is integral) the results may not perfectly match the results from SciPy due to floating-point rounding of intermediate results.

---

## Fourier filters

<code>fourier_ellipsoid</code> (input, size[, n, axis, output])	Multidimensional ellipsoid Fourier filter.
<code>fourier_gaussian</code> (input, sigma[, n, axis, output])	Multidimensional Gaussian shift filter.
<code>fourier_shift</code> (input, shift[, n, axis, output])	Multidimensional Fourier shift filter.
<code>fourier_uniform</code> (input, size[, n, axis, output])	Multidimensional uniform shift filter.

## `cupyx.scipy.ndimage.fourier_ellipsoid`

`cupyx.scipy.ndimage.fourier_ellipsoid`(input, size, n=-1, axis=-1, output=None)

Multidimensional ellipsoid Fourier filter.

The array is multiplied with the fourier transform of a ellipsoid of given sizes.

### Parameters

- **input** (`cupy.ndarray`) – The input array.
- **size** (`float` or `sequence of float`) – The size of the box used for filtering. If a float, *size* is the same for all axes. If a sequence, *size* has to contain one value for each axis.
- **n** (`int`, *optional*) – If *n* is negative (default), then the input is assumed to be the result of a complex fft. If *n* is larger than or equal to zero, the input is assumed to be the result of a real fft, and *n* gives the length of the array before transformation along the real transform direction.
- **axis** (`int`, *optional*) – The axis of the real transform (only used when *n* > -1).
- **output** (`cupy.ndarray`, *optional*) – If given, the result of shifting the input is placed in this array.

**Returns** The filtered output.

**Return type** output (`cupy.ndarray`)

### cupyx.scipy.ndimage.fourier\_gaussian

cupyx.scipy.ndimage.fourier\_gaussian(input, sigma, n=-1, axis=-1, output=None)  
Multidimensional Gaussian shift filter.

The array is multiplied with the Fourier transform of a (separable) Gaussian kernel.

#### Parameters

- **input** (`cupy.ndarray`) – The input array.
- **sigma** (`float` or *sequence of float*) – The sigma of the Gaussian kernel. If a float, *sigma* is the same for all axes. If a sequence, *sigma* has to contain one value for each axis.
- **n** (`int`, *optional*) – If *n* is negative (default), then the input is assumed to be the result of a complex fft. If *n* is larger than or equal to zero, the input is assumed to be the result of a real fft, and *n* gives the length of the array before transformation along the real transform direction.
- **axis** (`int`, *optional*) – The axis of the real transform (only used when *n* > -1).
- **output** (`cupy.ndarray`, *optional*) – If given, the result of shifting the input is placed in this array.

**Returns** The filtered output.

**Return type** output (`cupy.ndarray`)

### cupyx.scipy.ndimage.fourier\_shift

cupyx.scipy.ndimage.fourier\_shift(input, shift, n=-1, axis=-1, output=None)  
Multidimensional Fourier shift filter.

The array is multiplied with the Fourier transform of a shift operation.

#### Parameters

- **input** (`cupy.ndarray`) – The input array. This should be in the Fourier domain.
- **shift** (`float` or *sequence of float*) – The size of shift. If a float, *shift* is the same for all axes. If a sequence, *shift* has to contain one value for each axis.
- **n** (`int`, *optional*) – If *n* is negative (default), then the input is assumed to be the result of a complex fft. If *n* is larger than or equal to zero, the input is assumed to be the result of a real fft, and *n* gives the length of the array before transformation along the real transform direction.
- **axis** (`int`, *optional*) – The axis of the real transform (only used when *n* > -1).
- **output** (`cupy.ndarray`, *optional*) – If given, the result of shifting the input is placed in this array.

**Returns** The shifted output (in the Fourier domain).

**Return type** output (`cupy.ndarray`)

## cupyx.scipy.ndimage.fourier\_uniform

cupyx.scipy.ndimage.fourier\_uniform(*input*, *size*, *n*=-1, *axis*=-1, *output*=None)

Multidimensional uniform shift filter.

The array is multiplied with the Fourier transform of a box of given size.

### Parameters

- **input** (`cupy.ndarray`) – The input array.
- **size** (`float` or *sequence of float*) – The sigma of the box used for filtering. If a float, *size* is the same for all axes. If a sequence, *size* has to contain one value for each axis.
- **n** (`int`, *optional*) – If *n* is negative (default), then the input is assumed to be the result of a complex fft. If *n* is larger than or equal to zero, the input is assumed to be the result of a real fft, and *n* gives the length of the array before transformation along the real transform direction.
- **axis** (`int`, *optional*) – The axis of the real transform (only used when *n* > -1).
- **output** (`cupy.ndarray`, *optional*) – If given, the result of shifting the input is placed in this array.

**Returns** The filtered output.

**Return type** `output` (`cupy.ndarray`)

## Interpolation

<code>affine_transform</code> ( <i>input</i> , <i>matrix</i> [, <i>offset</i> , ...])	Apply an affine transformation.
<code>map_coordinates</code> ( <i>input</i> , <i>coordinates</i> [, ...])	Map the input array to new coordinates by interpolation.
<code>rotate</code> ( <i>input</i> , <i>angle</i> [, <i>axes</i> , <i>reshape</i> , ...])	Rotate an array.
<code>shift</code> ( <i>input</i> , <i>shift</i> [, <i>output</i> , <i>order</i> , <i>mode</i> , ...])	Shift an array.
<code>spline_filter</code> ( <i>input</i> [, <i>order</i> , <i>output</i> , <i>mode</i> ])	Multidimensional spline filter.
<code>spline_filter1d</code> ( <i>input</i> [, <i>order</i> , <i>axis</i> , ...])	Calculate a 1-D spline filter along the given axis.
<code>zoom</code> ( <i>input</i> , <i>zoom</i> [, <i>output</i> , <i>order</i> , <i>mode</i> , ...])	Zoom an array.

## cupyx.scipy.ndimage.affine\_transform

cupyx.scipy.ndimage.affine\_transform(*input*, *matrix*, *offset*=0.0, *output\_shape*=None, *output*=None, *order*=3, *mode*='constant', *cval*=0.0, *prefilter*=True)

Apply an affine transformation.

Given an output image pixel index vector *o*, the pixel value is determined from the input image at position `cupy.dot(matrix, o) + offset`.

### Parameters

- **input** (`cupy.ndarray`) – The input array.
- **matrix** (`cupy.ndarray`) – The inverse coordinate transformation matrix, mapping output coordinates to input coordinates. If *ndim* is the number of dimensions of *input*, the given matrix must have one of the following shapes:
  - (*ndim*, *ndim*): the linear transformation matrix for each output coordinate.

- `(ndim,)`: assume that the 2D transformation matrix is diagonal, with the diagonal specified by the given value.
- `(ndim + 1, ndim + 1)`: assume that the transformation is specified using homogeneous coordinates. In this case, any value passed to `offset` is ignored.
- `(ndim, ndim + 1)`: as above, but the bottom row of a homogeneous transformation matrix is always `[0, 0, ..., 1]`, and may be omitted.
- **offset** (*float or sequence*) – The offset into the array where the transform is applied. If a float, `offset` is the same for each axis. If a sequence, `offset` should contain one value for each axis.
- **output\_shape** (*tuple of ints*) – Shape tuple.
- **output** (*cupy.ndarray or dtype*) – The array in which to place the output, or the dtype of the returned array.
- **order** (*int*) – The order of the spline interpolation, default is 3. Must be in the range 0-5.
- **mode** (*str*) – Points outside the boundaries of the input are filled according to the given mode ('constant', 'nearest', 'mirror', 'reflect', 'wrap', 'grid-mirror', 'grid-wrap', 'grid-constant' or 'opencv').
- **cval** (*scalar*) – Value used for points outside the boundaries of the input if mode='constant' or mode='opencv'. Default is 0.0
- **prefilter** (*bool*) – It is not used yet. It just exists for compatibility with `scipy.ndimage`.

**Returns** The transformed input. If `output` is given as a parameter, `None` is returned.

**Return type** `cupy.ndarray` or `None`

See also:

`scipy.ndimage.affine_transform()`

## cupyx.scipy.ndimage.map\_coordinates

`cupyx.scipy.ndimage.map_coordinates(input, coordinates, output=None, order=3, mode='constant', cval=0.0, prefilter=True)`

Map the input array to new coordinates by interpolation.

The array of coordinates is used to find, for each point in the output, the corresponding coordinates in the input. The value of the input at those coordinates is determined by spline interpolation of the requested order.

The shape of the output is derived from that of the coordinate array by dropping the first axis. The values of the array along the first axis are the coordinates in the input array at which the output value is found.

### Parameters

- **input** (*cupy.ndarray*) – The input array.
- **coordinates** (*array\_like*) – The coordinates at which `input` is evaluated.
- **output** (*cupy.ndarray or dtype*) – The array in which to place the output, or the dtype of the returned array.
- **order** (*int*) – The order of the spline interpolation, default is 3. Must be in the range 0-5.
- **mode** (*str*) – Points outside the boundaries of the input are filled according to the given mode ('constant', 'nearest', 'mirror', 'reflect', 'wrap', 'grid-mirror', 'grid-wrap', 'grid-constant' or 'opencv').

- **cval** (*scalar*) – Value used for points outside the boundaries of the input if `mode='constant'` or `mode='opencv'`. Default is 0.0
- **prefilter** (*bool*) – It is not used yet. It just exists for compatibility with `scipy.ndimage`.

**Returns** The result of transforming the input. The shape of the output is derived from that of coordinates by dropping the first axis.

**Return type** `cupy.ndarray`

See also:

`scipy.ndimage.map_coordinates()`

### `cupyx.scipy.ndimage.rotate`

`cupyx.scipy.ndimage.rotate(input, angle, axes=(1, 0), reshape=True, output=None, order=3, mode='constant', cval=0.0, prefilter=True)`

Rotate an array.

The array is rotated in the plane defined by the two axes given by the `axes` parameter using spline interpolation of the requested order.

#### Parameters

- **input** (`cupy.ndarray`) – The input array.
- **angle** (*float*) – The rotation angle in degrees.
- **axes** (*tuple of 2 ints*) – The two axes that define the plane of rotation. Default is the first two axes.
- **reshape** (*bool*) – If `reshape` is `True`, the output shape is adapted so that the input array is contained completely in the output. Default is `True`.
- **output** (`cupy.ndarray` or *dtype*) – The array in which to place the output, or the dtype of the returned array.
- **order** (*int*) – The order of the spline interpolation, default is 3. Must be in the range 0-5.
- **mode** (*str*) – Points outside the boundaries of the input are filled according to the given mode ('constant', 'nearest', 'mirror', 'reflect', 'wrap', 'grid-mirror', 'grid-wrap', 'grid-constant' or 'opencv').
- **cval** (*scalar*) – Value used for points outside the boundaries of the input if `mode='constant'` or `mode='opencv'`. Default is 0.0
- **prefilter** (*bool*) – It is not used yet. It just exists for compatibility with `scipy.ndimage`.

**Returns** The rotated input.

**Return type** `cupy.ndarray` or `None`

See also:

`scipy.ndimage.rotate()`

### cupyx.scipy.ndimage.shift

`cupyx.scipy.ndimage.shift`(*input*, *shift*, *output=None*, *order=3*, *mode='constant'*, *cval=0.0*, *prefilter=True*)  
Shift an array.

The array is shifted using spline interpolation of the requested order. Points outside the boundaries of the input are filled according to the given mode.

#### Parameters

- **input** (`cupy.ndarray`) – The input array.
- **shift** (`float` or `sequence`) – The shift along the axes. If a float, `shift` is the same for each axis. If a sequence, `shift` should contain one value for each axis.
- **output** (`cupy.ndarray` or `dtype`) – The array in which to place the output, or the dtype of the returned array.
- **order** (`int`) – The order of the spline interpolation, default is 3. Must be in the range 0-5.
- **mode** (`str`) – Points outside the boundaries of the input are filled according to the given mode ('constant', 'nearest', 'mirror', 'reflect', 'wrap', 'grid-mirror', 'grid-wrap', 'grid-constant' or 'opencv').
- **cval** (`scalar`) – Value used for points outside the boundaries of the input if mode='constant' or mode='opencv'. Default is 0.0
- **prefilter** (`bool`) – It is not used yet. It just exists for compatibility with `scipy.ndimage`.

**Returns** The shifted input.

**Return type** `cupy.ndarray` or `None`

See also:

`scipy.ndimage.shift()`

### cupyx.scipy.ndimage.spline\_filter

`cupyx.scipy.ndimage.spline_filter`(*input*, *order=3*, *output=<class 'numpy.float64'>*, *mode='mirror'*)  
Multidimensional spline filter.

#### Parameters

- **input** (`cupy.ndarray`) – The input array.
- **order** (`int`) – The order of the spline interpolation, default is 3. Must be in the range 0-5.
- **output** (`cupy.ndarray` or `dtype`, *optional*) – The array in which to place the output, or the dtype of the returned array. Default is `numpy.float64`.
- **mode** (`str`) – Points outside the boundaries of the input are filled according to the given mode ('constant', 'nearest', 'mirror', 'reflect', 'wrap', 'grid-mirror', 'grid-wrap', 'grid-constant' or 'opencv').

**Returns** The result of prefiltering the input.

**Return type** `cupy.ndarray`

See also:

`scipy.spline_filter1d()`

### cupyx.scipy.ndimage.spline\_filter1d

```
cupyx.scipy.ndimage.spline_filter1d(input, order=3, axis=-1, output=<class 'numpy.float64'>,
                                   mode='mirror')
```

Calculate a 1-D spline filter along the given axis.

The lines of the array along the given axis are filtered by a spline filter. The order of the spline must be  $\geq 2$  and  $\leq 5$ .

#### Parameters

- **input** (`cupy.ndarray`) – The input array.
- **order** (`int`) – The order of the spline interpolation, default is 3. Must be in the range 0-5.
- **axis** (`int`) – The axis along which the spline filter is applied. Default is the last axis.
- **output** (`cupy.ndarray` or `dtype`, *optional*) – The array in which to place the output, or the dtype of the returned array. Default is `numpy.float64`.
- **mode** (`str`) – Points outside the boundaries of the input are filled according to the given mode ('constant', 'nearest', 'mirror', 'reflect', 'wrap', 'grid-mirror', 'grid-wrap', 'grid-constant' or 'opencv').

**Returns** The result of prefiltering the input.

**Return type** `cupy.ndarray`

See also:

`scipy.spline_filter1d()`

### cupyx.scipy.ndimage.zoom

```
cupyx.scipy.ndimage.zoom(input, zoom, output=None, order=3, mode='constant', cval=0.0, prefilter=True, *,
                          grid_mode=False)
```

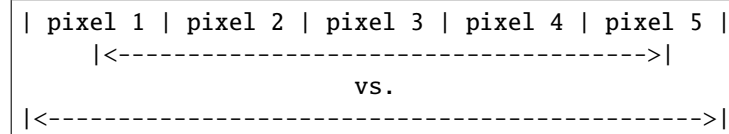
Zoom an array.

The array is zoomed using spline interpolation of the requested order.

#### Parameters

- **input** (`cupy.ndarray`) – The input array.
- **zoom** (`float` or `sequence`) – The zoom factor along the axes. If a float, zoom is the same for each axis. If a sequence, zoom should contain one value for each axis.
- **output** (`cupy.ndarray` or `dtype`) – The array in which to place the output, or the dtype of the returned array.
- **order** (`int`) – The order of the spline interpolation, default is 3. Must be in the range 0-5.
- **mode** (`str`) – Points outside the boundaries of the input are filled according to the given mode ('constant', 'nearest', 'mirror', 'reflect', 'wrap', 'grid-mirror', 'grid-wrap', 'grid-constant' or 'opencv').
- **cval** (`scalar`) – Value used for points outside the boundaries of the input if mode='constant' or mode='opencv'. Default is 0.0
- **prefilter** (`bool`) – It is not used yet. It just exists for compatibility with `scipy.ndimage`.

- **grid\_mode** (*bool*, *optional*) – If False, the distance from the pixel centers is zoomed. Otherwise, the distance including the full pixel extent is used. For example, a 1d signal of length 5 is considered to have length 4 when `grid_mode` is False, but length 5 when `grid_mode` is True. See the following visual illustration:



The starting point of the arrow in the diagram above corresponds to coordinate location 0 in each mode.

**Returns** The zoomed input.

**Return type** `cupy.ndarray` or `None`

**See also:**

`scipy.ndimage.zoom()`

## Measurements

<code>center_of_mass(input[, labels, index])</code>	Calculate the center of mass of the values of an array at labels.
<code>extrema(input[, labels, index])</code>	Calculate the minimums and maximums of the values of an array at labels, along with their positions.
<code>histogram(input, min, max, bins[, labels, index])</code>	Calculate the histogram of the values of an array, optionally at labels.
<code>label(input[, structure, output])</code>	Labels features in an array.
<code>labeled_comprehension(input, labels, index, ...)</code>	Array resulting from applying <code>func</code> to each labeled region.
<code>maximum(input[, labels, index])</code>	Calculate the maximum of the values of an array over labeled regions.
<code>maximum_position(input[, labels, index])</code>	Find the positions of the maximums of the values of an array at labels.
<code>mean(input[, labels, index])</code>	Calculates the mean of the values of an n-D image array, optionally
<code>median(input[, labels, index])</code>	Calculate the median of the values of an array over labeled regions.
<code>minimum(input[, labels, index])</code>	Calculate the minimum of the values of an array over labeled regions.
<code>minimum_position(input[, labels, index])</code>	Find the positions of the minimums of the values of an array at labels.
<code>standard_deviation(input[, labels, index])</code>	Calculates the standard deviation of the values of an n-D image array, optionally at specified sub-regions.
<code>sum_labels(input[, labels, index])</code>	Calculates the sum of the values of an n-D image array, optionally
<code>variance(input[, labels, index])</code>	Calculates the variance of the values of an n-D image array, optionally at specified sub-regions.



**cupyx.scipy.ndimage.center\_of\_mass**

`cupyx.scipy.ndimage.center_of_mass(input, labels=None, index=None)`

Calculate the center of mass of the values of an array at labels.

**Parameters**

- **input** (`cupy.ndarray`) – Data from which to calculate center-of-mass. The masses can either be positive or negative.
- **labels** (`cupy.ndarray`, *optional*) – Labels for objects in *input*, as generated by *ndimage.label*. Only used with *index*. Dimensions must be the same as *input*.
- **index** (*int or sequence of ints*, *optional*) – Labels for which to calculate centers-of-mass. If not specified, all labels greater than zero are used. Only used with *labels*.

**Returns** Coordinates of centers-of-mass.

**Return type** `tuple` or list of tuples

**See also:**

`scipy.ndimage.center_of_mass()`

**cupyx.scipy.ndimage.extrema**

`cupyx.scipy.ndimage.extrema(input, labels=None, index=None)`

Calculate the minimums and maximums of the values of an array at labels, along with their positions.

**Parameters**

- **input** (`cupy.ndarray`) – N-D image data to process.
- **labels** (`cupy.ndarray`, *optional*) – Labels of features in input. If not None, must be same shape as *input*.
- **index** (*int or sequence of ints*, *optional*) – Labels to include in output. If None (default), all values where non-zero *labels* are used.

**Returns**

A tuple that contains the following values.

**minimums** (`cupy.ndarray`): Values of minimums in each feature.

**maximums** (`cupy.ndarray`): Values of maximums in each feature.

**min\_positions** (`tuple or list of tuples`): Each tuple gives the N-D coordinates of the corresponding minimum.

**max\_positions** (`tuple or list of tuples`): Each tuple gives the N-D coordinates of the corresponding maximum.

**See also:**

`scipy.ndimage.extrema()`

### cupyx.scipy.ndimage.histogram

cupyx.scipy.ndimage.**histogram**(*input*, *min*, *max*, *bins*, *labels=None*, *index=None*)

Calculate the histogram of the values of an array, optionally at labels.

Histogram calculates the frequency of values in an array within bins determined by *min*, *max*, and *bins*. The *labels* and *index* keywords can limit the scope of the histogram to specified sub-regions within the array.

#### Parameters

- **input** (`cupy.ndarray`) – Data for which to calculate histogram.
- **min** (`int`) – Minimum values of range of histogram bins.
- **max** (`int`) – Maximum values of range of histogram bins.
- **bins** (`int`) – Number of bins.
- **labels** (`cupy.ndarray`, *optional*) – Labels for objects in *input*. If not `None`, must be same shape as *input*.
- **index** (`int` or *sequence of ints*, *optional*) – Label or labels for which to calculate histogram. If `None`, all values where label is greater than zero are used.

**Returns** Histogram counts.

**Return type** `cupy.ndarray`

See also:

`scipy.ndimage.histogram()`

### cupyx.scipy.ndimage.label

cupyx.scipy.ndimage.**label**(*input*, *structure=None*, *output=None*)

Labels features in an array.

#### Parameters

- **input** (`cupy.ndarray`) – The input array.
- **structure** (*array\_like* or `None`) – A structuring element that defines feature connections. `structure` must be centersymmetric. If None, structure is automatically generated with a squared connectivity equal to one.`
- **output** (`cupy.ndarray`, *dtype* or `None`) – The array in which to place the output.

#### Returns

An integer array where each unique feature in `input` has a unique label in the array.`

`num_features` (`int`): Number of features found.

**Return type** `label` (`cupy.ndarray`)

**Warning:** This function may synchronize the device.

See also:

`scipy.ndimage.label()`

### cupyx.scipy.ndimage.labeled\_comprehension

`cupyx.scipy.ndimage.labeled_comprehension(input, labels, index, func, out_dtype, default, pass_positions=False)`

Array resulting from applying `func` to each labeled region.

Roughly equivalent to `[func(input[labels == i]) for i in index]`.

Sequentially applies an arbitrary function (that works on `array_like` input) to subsets of an N-D image array specified by `labels` and `index`. The option exists to provide the function with positional parameters as the second argument.

#### Parameters

- **input** (`cupy.ndarray`) – Data from which to select `labels` to process.
- **labels** (`cupy.ndarray` or `None`) – Labels to objects in `input`. If not `None`, array must be same shape as `input`. If `None`, `func` is applied to raveled `input`.
- **index** (`int`, *sequence of ints* or `None`) – Subset of `labels` to which to apply `func`. If a scalar, a single value is returned. If `None`, `func` is applied to all non-zero values of `labels`.
- **func** (*callable*) – Python function to apply to `labels` from `input`.
- **out\_dtype** (*dtype*) – Dtype to use for `result`.
- **default** (`int`, *float* or `None`) – Default return value when a element of `index` does not exist in `labels`.
- **pass\_positions** (*bool*, *optional*) – If `True`, pass linear indices to `func` as a second argument.

**Returns** Result of applying `func` to each of `labels` to `input` in `index`.

**Return type** `cupy.ndarray`

See also:

`scipy.ndimage.labeled_comprehension()`

### cupyx.scipy.ndimage.maximum

`cupyx.scipy.ndimage.maximum(input, labels=None, index=None)`

Calculate the maximum of the values of an array over labeled regions.

#### Parameters

- **input** (`cupy.ndarray`) – Array of values. For each region specified by `labels`, the maximal values of `input` over the region is computed.
- **labels** (`cupy.ndarray`, *optional*) – An array of integers marking different regions over which the maximum value of `input` is to be computed. `labels` must have the same shape as `input`. If `labels` is not specified, the maximum over the whole array is returned.
- **index** (*array\_like*, *optional*) – A list of region labels that are taken into account for computing the maxima. If `index` is `None`, the maximum over all elements where `labels` is non-zero is returned.

**Returns** Array of maxima of `input` over the regions determined by `labels` and whose index is in `index`. If `index` or `labels` are not specified, a 0-dimensional `cupy.ndarray` is returned: the maximal value of `input` if `labels` is `None`, and the maximal value of elements where `labels` is greater than zero if `index` is `None`.

**Return type** `cupy.ndarray`

**See also:**

`scipy.ndimage.maximum()`

### `cupyx.scipy.ndimage.maximum_position`

`cupyx.scipy.ndimage.maximum_position(input, labels=None, index=None)`

Find the positions of the maximums of the values of an array at labels.

For each region specified by *labels*, the position of the maximum value of *input* within the region is returned.

#### Parameters

- **input** (`cupy.ndarray`) – Array of values. For each region specified by *labels*, the maximal values of *input* over the region is computed.
- **labels** (`cupy.ndarray`, *optional*) – An array of integers marking different regions over which the position of the maximum value of *input* is to be computed. *labels* must have the same shape as *input*. If *labels* is not specified, the location of the first maximum over the whole array is returned.

The *labels* argument only works when *index* is specified.

- **index** (*array\_like*, *optional*) – A list of region labels that are taken into account for finding the location of the maxima. If *index* is *None*, the first maximum over all elements where *labels* is non-zero is returned.

The *index* argument only works when *labels* is specified.

#### Returns

Tuple of ints or list of tuples of ints that specify the location of maxima of *input* over the regions determined by *labels* and whose index is in *index*.

If *index* or *labels* are not specified, a tuple of ints is returned specifying the location of the first maximal value of *input*.

---

**Note:** When *input* has multiple identical maxima within a labeled region, the coordinates returned are not guaranteed to match those returned by SciPy.

---

**See also:**

`scipy.ndimage.maximum_position()`

### `cupyx.scipy.ndimage.mean`

`cupyx.scipy.ndimage.mean(input, labels=None, index=None)`

Calculates the mean of the values of an n-D image array, *optionally* at specified sub-regions.

#### Parameters

- **input** (`cupy.ndarray`) – Nd-image data to process.
- **labels** (`cupy.ndarray` or *None*) – Labels defining sub-regions in *input*. If not *None*, must be same shape as *input*.

- **index** (`cupy.ndarray` or `None`) – *labels* to include in output. If `None` (default), all values where *labels* is non-zero are used.

**Returns** mean of values, for each sub-region if *labels* and *index* are specified.

**Return type** `mean` (`cupy.ndarray`)

See also:

`scipy.ndimage.mean()`

### `cupyx.scipy.ndimage.median`

`cupyx.scipy.ndimage.median(input, labels=None, index=None)`

Calculate the median of the values of an array over labeled regions.

#### Parameters

- **input** (`cupy.ndarray`) – Array of values. For each region specified by *labels*, the median values of *input* over the region is computed.
- **labels** (`cupy.ndarray`, *optional*) – An array of integers marking different regions over which the median value of *input* is to be computed. *labels* must have the same shape as *input*. If *labels* is not specified, the median over the whole array is returned.
- **index** (*array\_like*, *optional*) – A list of region labels that are taken into account for computing the medians. If *index* is `None`, the median over all elements where *labels* is non-zero is returned.

**Returns** Array of medians of *input* over the regions determined by *labels* and whose index is in *index*. If *index* or *labels* are not specified, a 0-dimensional `cupy.ndarray` is returned: the median value of *input* if *labels* is `None`, and the median value of elements where *labels* is greater than zero if *index* is `None`.

**Return type** `cupy.ndarray`

See also:

`scipy.ndimage.median()`

### `cupyx.scipy.ndimage.minimum`

`cupyx.scipy.ndimage.minimum(input, labels=None, index=None)`

Calculate the minimum of the values of an array over labeled regions.

#### Parameters

- **input** (`cupy.ndarray`) – Array of values. For each region specified by *labels*, the minimal values of *input* over the region is computed.
- **labels** (`cupy.ndarray`, *optional*) – An array of integers marking different regions over which the minimum value of *input* is to be computed. *labels* must have the same shape as *input*. If *labels* is not specified, the minimum over the whole array is returned.
- **index** (*array\_like*, *optional*) – A list of region labels that are taken into account for computing the minima. If *index* is `None`, the minimum over all elements where *labels* is non-zero is returned.

**Returns** Array of minima of *input* over the regions determined by *labels* and whose index is in *index*. If *index* or *labels* are not specified, a 0-dimensional `cupy.ndarray` is returned: the minimal value of *input* if *labels* is `None`, and the minimal value of elements where *labels* is greater than zero if *index* is `None`.

**Return type** `cupy.ndarray`

**See also:**

`scipy.ndimage.minimum()`

### `cupyx.scipy.ndimage.minimum_position`

`cupyx.scipy.ndimage.minimum_position(input, labels=None, index=None)`

Find the positions of the minimums of the values of an array at labels.

For each region specified by *labels*, the position of the minimum value of *input* within the region is returned.

#### Parameters

- **input** (`cupy.ndarray`) – Array of values. For each region specified by *labels*, the minimal values of *input* over the region is computed.
- **labels** (`cupy.ndarray`, *optional*) – An array of integers marking different regions over which the position of the minimum value of *input* is to be computed. *labels* must have the same shape as *input*. If *labels* is not specified, the location of the first minimum over the whole array is returned.

The *labels* argument only works when *index* is specified.

- **index** (*array\_like*, *optional*) – A list of region labels that are taken into account for finding the location of the minima. If *index* is `None`, the **first** minimum over all elements where *labels* is non-zero is returned.

The *index* argument only works when *labels* is specified.

#### Returns

Tuple of ints or list of tuples of ints that specify the location of minima of *input* over the regions determined by *labels* and whose index is in *index*.

If *index* or *labels* are not specified, a tuple of ints is returned specifying the location of the first minimal value of *input*.

---

**Note:** When *input* has multiple identical minima within a labeled region, the coordinates returned are not guaranteed to match those returned by SciPy.

---

**See also:**

`scipy.ndimage.minimum_position()`

### cupyx.scipy.ndimage.standard\_deviation

cupyx.scipy.ndimage.**standard\_deviation**(input, labels=None, index=None)

Calculates the standard deviation of the values of an n-D image array, optionally at specified sub-regions.

#### Parameters

- **input** (`cupy.ndarray`) – Nd-image data to process.
- **labels** (`cupy.ndarray` or `None`) – Labels defining sub-regions in *input*. If not `None`, must be same shape as *input*.
- **index** (`cupy.ndarray` or `None`) – *labels* to include in output. If `None` (default), all values where *labels* is non-zero are used.

**Returns** standard deviation of values, for each sub-region if *labels* and *index* are specified.

**Return type** standard\_deviation (`cupy.ndarray`)

See also:

`scipy.ndimage.standard_deviation()`

### cupyx.scipy.ndimage.sum\_labels

cupyx.scipy.ndimage.**sum\_labels**(input, labels=None, index=None)

Calculates the sum of the values of an n-D image array, optionally at specified sub-regions.

#### Parameters

- **input** (`cupy.ndarray`) – Nd-image data to process.
- **labels** (`cupy.ndarray` or `None`) – Labels defining sub-regions in *input*. If not `None`, must be same shape as *input*.
- **index** (`cupy.ndarray` or `None`) – *labels* to include in output. If `None` (default), all values where *labels* is non-zero are used.

**Returns** sum of values, for each sub-region if *labels* and *index* are specified.

**Return type** sum (`cupy.ndarray`)

See also:

`scipy.ndimage.sum_labels()`

### cupyx.scipy.ndimage.variance

cupyx.scipy.ndimage.**variance**(input, labels=None, index=None)

Calculates the variance of the values of an n-D image array, optionally at specified sub-regions.

#### Parameters

- **input** (`cupy.ndarray`) – Nd-image data to process.
- **labels** (`cupy.ndarray` or `None`) – Labels defining sub-regions in *input*. If not `None`, must be same shape as *input*.

- **index** (`cupy.ndarray` or `None`) – *labels* to include in output. If `None` (default), all values where *labels* is non-zero are used.

**Returns** Values of variance, for each sub-region if *labels* and *index* are specified.

**Return type** `cupy.ndarray`

See also:

`scipy.ndimage.variance()`

## Morphology

<code>binary_closing</code> (input[, structure, ...])	Multidimensional binary closing with the given structuring element.
<code>binary_dilation</code> (input[, structure, ...])	Multidimensional binary dilation with the given structuring element.
<code>binary_erosion</code> (input[, structure, ...])	Multidimensional binary erosion with a given structuring element.
<code>binary_fill_holes</code> (input[, structure, ...])	Fill the holes in binary objects.
<code>binary_hit_or_miss</code> (input[, structure1, ...])	Multidimensional binary hit-or-miss transform.
<code>binary_opening</code> (input[, structure, ...])	Multidimensional binary opening with the given structuring element.
<code>binary_propagation</code> (input[, structure, mask, ...])	Multidimensional binary propagation with the given structuring element.
<code>black_tophat</code> (input[, size, footprint, ...])	Multidimensional black tophat filter.
<code>generate_binary_structure</code> (rank, connectivity)	Generate a binary structure for binary morphological operations.
<code>grey_closing</code> (input[, size, footprint, ...])	Calculates a multi-dimensional greyscale closing.
<code>grey_dilation</code> (input[, size, footprint, ...])	Calculates a greyscale dilation.
<code>grey_erosion</code> (input[, size, footprint, ...])	Calculates a greyscale erosion.
<code>grey_opening</code> (input[, size, footprint, ...])	Calculates a multi-dimensional greyscale opening.
<code>iterate_structure</code> (structure, iterations[, ...])	Iterate a structure by dilating it with itself.
<code>morphological_gradient</code> (input[, size, ...])	Multidimensional morphological gradient.
<code>morphological_laplace</code> (input[, size, ...])	Multidimensional morphological laplace.
<code>white_tophat</code> (input[, size, footprint, ...])	Multidimensional white tophat filter.

## cupyx.scipy.ndimage.binary\_closing

`cupyx.scipy.ndimage.binary_closing`(input, structure=None, iterations=1, output=None, origin=0, mask=None, border\_value=0, brute\_force=False)

Multidimensional binary closing with the given structuring element.

The *closing* of an input image by a structuring element is the *erosion* of the *dilation* of the image by the structuring element.

### Parameters

- **input** (`cupy.ndarray`) – The input binary array to be closed. Non-zero (True) elements form the subset to be closed.
- **structure** (`cupy.ndarray`, optional) – The structuring element used for the closing. Non-zero elements are considered True. If no structuring element is provided an element is generated with a square connectivity equal to one. (Default value = None).



- **iterations** (*int*, *optional*) – The closing is repeated *iterations* times (one, by default). If *iterations* is less than 1, the closing is repeated until the result does not change anymore. Only an integer of iterations is accepted.
- **output** (*cupy.ndarray*, *optional*) – Array of the same shape as input, into which the output is placed. By default, a new array is created.
- **origin** (*int* or *tuple of ints*, *optional*) – Placement of the filter, by default 0.
- **mask** (*cupy.ndarray* or *None*, *optional*) – If a mask is given, only those elements with a True value at the corresponding mask element are modified at each iteration. (Default value = None)
- **border\_value** (*int* (cast to 0 or 1), *optional*) – Value at the border in the output array. (Default value = 0)
- **brute\_force** (*boolean*, *optional*) – Memory condition: if False, only the pixels whose value was changed in the last iteration are tracked as candidates to be updated (dilated) in the current iteration; if True all pixels are considered as candidates for closing, regardless of what happened in the previous iteration.

**Returns** The result of binary closing.

**Return type** *cupy.ndarray*

**Warning:** This function may synchronize the device.

See also:

*scipy.ndimage.binary\_closing()*

### cupyx.scipy.ndimage.binary\_dilation

*cupyx.scipy.ndimage.binary\_dilation(input, structure=None, iterations=1, mask=None, output=None, border\_value=0, origin=0, brute\_force=False)*

Multidimensional binary dilation with the given structuring element.

#### Parameters

- **input** (*cupy.ndarray*) – The input binary array\_like to be dilated. Non-zero (True) elements form the subset to be dilated.
- **structure** (*cupy.ndarray*, *optional*) – The structuring element used for the dilation. Non-zero elements are considered True. If no structuring element is provided an element is generated with a square connectivity equal to one. (Default value = None).
- **iterations** (*int*, *optional*) – The dilation is repeated *iterations* times (one, by default). If *iterations* is less than 1, the dilation is repeated until the result does not change anymore. Only an integer of iterations is accepted.
- **mask** (*cupy.ndarray* or *None*, *optional*) – If a mask is given, only those elements with a True value at the corresponding mask element are modified at each iteration. (Default value = None)
- **output** (*cupy.ndarray*, *optional*) – Array of the same shape as input, into which the output is placed. By default, a new array is created.
- **border\_value** (*int* (cast to 0 or 1), *optional*) – Value at the border in the output array. (Default value = 0)

- **origin** (*int or tuple of ints, optional*) – Placement of the filter, by default 0.
- **brute\_force** (*boolean, optional*) – Memory condition: if False, only the pixels whose value was changed in the last iteration are tracked as candidates to be updated (dilated) in the current iteration; if True all pixels are considered as candidates for dilation, regardless of what happened in the previous iteration.

**Returns** The result of binary dilation.

**Return type** *cupy.ndarray*

**Warning:** This function may synchronize the device.

See also:

*scipy.ndimage.binary\_dilation()*

### cupyx.scipy.ndimage.binary\_erosion

`cupyx.scipy.ndimage.binary_erosion(input, structure=None, iterations=1, mask=None, output=None, border_value=0, origin=0, brute_force=False)`

Multidimensional binary erosion with a given structuring element.

Binary erosion is a mathematical morphology operation used for image processing.

#### Parameters

- **input** (*cupy.ndarray*) – The input binary array\_like to be eroded. Non-zero (True) elements form the subset to be eroded.
- **structure** (*cupy.ndarray, optional*) – The structuring element used for the erosion. Non-zero elements are considered True. If no structuring element is provided an element is generated with a square connectivity equal to one. (Default value = None).
- **iterations** (*int, optional*) – The erosion is repeated `iterations` times (one, by default). If `iterations` is less than 1, the erosion is repeated until the result does not change anymore. Only an integer of iterations is accepted.
- **mask** (*cupy.ndarray or None, optional*) – If a mask is given, only those elements with a True value at the corresponding mask element are modified at each iteration. (Default value = None)
- **output** (*cupy.ndarray, optional*) – Array of the same shape as input, into which the output is placed. By default, a new array is created.
- **border\_value** (*int (cast to 0 or 1), optional*) – Value at the border in the output array. (Default value = 0)
- **origin** (*int or tuple of ints, optional*) – Placement of the filter, by default 0.
- **brute\_force** (*boolean, optional*) – Memory condition: if False, only the pixels whose value was changed in the last iteration are tracked as candidates to be updated (eroded) in the current iteration; if True all pixels are considered as candidates for erosion, regardless of what happened in the previous iteration.

**Returns** The result of binary erosion.

**Return type** *cupy.ndarray*

**Warning:** This function may synchronize the device.

See also:

`scipy.ndimage.binary_erosion()`

### `cupyx.scipy.ndimage.binary_fill_holes`

`cupyx.scipy.ndimage.binary_fill_holes(input, structure=None, output=None, origin=0)`

Fill the holes in binary objects.

#### Parameters

- **input** (`cupy.ndarray`) – N-D binary array with holes to be filled.
- **structure** (`cupy.ndarray`, *optional*) – Structuring element used in the computation; large-size elements make computations faster but may miss holes separated from the background by thin regions. The default element (with a square connectivity equal to one) yields the intuitive result where all holes in the input have been filled.
- **output** (`cupy.ndarray`, *dtype or None, optional*) – Array of the same shape as input, into which the output is placed. By default, a new array is created.
- **origin** (*int, tuple of ints, optional*) – Position of the structuring element.

**Returns** Transformation of the initial image input where holes have been filled.

**Return type** `cupy.ndarray`

**Warning:** This function may synchronize the device.

See also:

`scipy.ndimage.binary_fill_holes()`

### `cupyx.scipy.ndimage.binary_hit_or_miss`

`cupyx.scipy.ndimage.binary_hit_or_miss(input, structure1=None, structure2=None, output=None, origin1=0, origin2=None)`

Multidimensional binary hit-or-miss transform.

The hit-or-miss transform finds the locations of a given pattern inside the input image.

#### Parameters

- **input** (`cupy.ndarray`) – Binary image where a pattern is to be detected.
- **structure1** (`cupy.ndarray`, *optional*) – Part of the structuring element to be fitted to the foreground (non-zero elements) of input. If no value is provided, a structure of square connectivity 1 is chosen.
- **structure2** (`cupy.ndarray`, *optional*) – Second part of the structuring element that has to miss completely the foreground. If no value is provided, the complementary of structure1 is taken.
- **output** (`cupy.ndarray`, *dtype or None, optional*) – Array of the same shape as input, into which the output is placed. By default, a new array is created.

- **origin1** (*int or tuple of ints, optional*) – Placement of the first part of the structuring element `structure1`, by default 0 for a centered structure.
- **origin2** (*int or tuple of ints or `None`, optional*) – Placement of the second part of the structuring element `structure2`, by default 0 for a centered structure. If a value is provided for `origin1` and not for `origin2`, then `origin2` is set to `origin1`.

**Returns** Hit-or-miss transform of `input` with the given structuring element (`structure1`, `structure2`).

**Return type** `cupy.ndarray`

**Warning:** This function may synchronize the device.

See also:

`scipy.ndimage.binary_hit_or_miss()`

### `cupyx.scipy.ndimage.binary_opening`

`cupyx.scipy.ndimage.binary_opening(input, structure=None, iterations=1, output=None, origin=0, mask=None, border_value=0, brute_force=False)`

Multidimensional binary opening with the given structuring element.

The *opening* of an input image by a structuring element is the *dilation* of the *erosion* of the image by the structuring element.

#### Parameters

- **input** (`cupy.ndarray`) – The input binary array to be opened. Non-zero (True) elements form the subset to be opened.
- **structure** (`cupy.ndarray`, *optional*) – The structuring element used for the opening. Non-zero elements are considered True. If no structuring element is provided an element is generated with a square connectivity equal to one. (Default value = `None`).
- **iterations** (*int, optional*) – The opening is repeated `iterations` times (one, by default). If `iterations` is less than 1, the opening is repeated until the result does not change anymore. Only an integer of iterations is accepted.
- **output** (`cupy.ndarray`, *optional*) – Array of the same shape as input, into which the output is placed. By default, a new array is created.
- **origin** (*int or tuple of ints, optional*) – Placement of the filter, by default 0.
- **mask** (`cupy.ndarray` or `None`, *optional*) – If a mask is given, only those elements with a True value at the corresponding mask element are modified at each iteration. (Default value = `None`)
- **border\_value** (*int (cast to 0 or 1), optional*) – Value at the border in the output array. (Default value = 0)
- **brute\_force** (*boolean, optional*) – Memory condition: if False, only the pixels whose value was changed in the last iteration are tracked as candidates to be updated (dilated) in the current iteration; if True all pixels are considered as candidates for opening, regardless of what happened in the previous iteration.

**Returns** The result of binary opening.

**Return type** `cupy.ndarray`

**Warning:** This function may synchronize the device.

See also:

`scipy.ndimage.binary_opening()`

### cupyx.scipy.ndimage.binary\_propagation

`cupyx.scipy.ndimage.binary_propagation(input, structure=None, mask=None, output=None, border_value=0, origin=0)`

Multidimensional binary propagation with the given structuring element.

#### Parameters

- **input** (`cupy.ndarray`) – Binary image to be propagated inside mask.
- **structure** (`cupy.ndarray`, *optional*) – Structuring element used in the successive dilations. The output may depend on the structuring element, especially if `mask` has several connex components. If no structuring element is provided, an element is generated with a squared connectivity equal to one.
- **mask** (`cupy.ndarray`, *optional*) – Binary mask defining the region into which `input` is allowed to propagate.
- **output** (`cupy.ndarray`, *optional*) – Array of the same shape as `input`, into which the output is placed. By default, a new array is created.
- **border\_value** (`int`, *optional*) – Value at the border in the output array. The value is cast to 0 or 1.
- **origin** (`int` or *tuple of ints*, *optional*) – Placement of the filter.

**Returns** Binary propagation of `input` inside `mask`.

**Return type** `cupy.ndarray`

**Warning:** This function may synchronize the device.

See also:

`scipy.ndimage.binary_propagation()`

### cupyx.scipy.ndimage.black\_tophat

`cupyx.scipy.ndimage.black_tophat(input, size=None, footprint=None, structure=None, output=None, mode='reflect', cval=0.0, origin=0)`

Multidimensional black tophat filter.

#### Parameters

- **input** (`cupy.ndarray`) – The input array.
- **size** (*tuple of ints*) – Shape of a flat and full structuring element used for the black tophat. Optional if `footprint` or `structure` is provided.

- **footprint** (*array of ints*) – Positions of non-infinite elements of a flat structuring element used for the black tophat. Non-zero values give the set of neighbors of the center over which opening is chosen.
- **structure** (*array of ints*) – Structuring element used for the black tophat. `structure` may be a non-flat structuring element.
- **output** (`cupy.ndarray`, *dtype* or `None`) – The array in which to place the output.
- **mode** (*str*) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (*scalar*) – Value to fill past edges of input if mode is constant. Default is 0.0.
- **origin** (*scalar or tuple of scalar*) – The origin parameter controls the placement of the filter, relative to the center of the current element of the input. Default of 0 is equivalent to (0,)\*input.ndim.

**Returns** Result of the filter of input with structure.

**Return type** `cupy.ndarray`

See also:

`scipy.ndimage.black_tophat()`

## `cupyx.scipy.ndimage.generate_binary_structure`

`cupyx.scipy.ndimage.generate_binary_structure(rank, connectivity)`

Generate a binary structure for binary morphological operations.

### Parameters

- **rank** (*int*) – Number of dimensions of the array to which the structuring element will be applied, as returned by `np.ndim`.
- **connectivity** (*int*) – `connectivity` determines which elements of the output array belong to the structure, i.e., are considered as neighbors of the central element. Elements up to a squared distance of `connectivity` from the center are considered neighbors. `connectivity` may range from 1 (no diagonal elements are neighbors) to `rank` (all elements are neighbors).

**Returns** Structuring element which may be used for binary morphological operations, with `rank` dimensions and all dimensions equal to 3.

**Return type** `cupy.ndarray`

See also:

`scipy.ndimage.generate_binary_structure()`

### cupyx.scipy.ndimage.grey\_closing

`cupyx.scipy.ndimage.grey_closing(input, size=None, footprint=None, structure=None, output=None, mode='reflect', cval=0.0, origin=0)`

Calculates a multi-dimensional greyscale closing.

#### Parameters

- **input** (`cupy.ndarray`) – The input array.
- **size** (*tuple of ints*) – Shape of a flat and full structuring element used for the greyscale closing. Optional if `footprint` or `structure` is provided.
- **footprint** (*array of ints*) – Positions of non-infinite elements of a flat structuring element used for greyscale closing. Non-zero values give the set of neighbors of the center over which closing is chosen.
- **structure** (*array of ints*) – Structuring element used for the greyscale closing. structure may be a non-flat structuring element.
- **output** (`cupy.ndarray`, *dtype* or `None`) – The array in which to place the output.
- **mode** (*str*) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (*scalar*) – Value to fill past edges of input if mode is constant. Default is 0.0.
- **origin** (*scalar or tuple of scalar*) – The origin parameter controls the placement of the filter, relative to the center of the current element of the input. Default of 0 is equivalent to (0,)\*input.ndim.

**Returns** The result of greyscale closing.

**Return type** `cupy.ndarray`

See also:

`scipy.ndimage.grey_closing()`

### cupyx.scipy.ndimage.grey\_dilation

`cupyx.scipy.ndimage.grey_dilation(input, size=None, footprint=None, structure=None, output=None, mode='reflect', cval=0.0, origin=0)`

Calculates a greyscale dilation.

#### Parameters

- **input** (`cupy.ndarray`) – The input array.
- **size** (*tuple of ints*) – Shape of a flat and full structuring element used for the greyscale dilation. Optional if `footprint` or `structure` is provided.
- **footprint** (*array of ints*) – Positions of non-infinite elements of a flat structuring element used for greyscale dilation. Non-zero values give the set of neighbors of the center over which maximum is chosen.
- **structure** (*array of ints*) – Structuring element used for the greyscale dilation. structure may be a non-flat structuring element.
- **output** (`cupy.ndarray`, *dtype* or `None`) – The array in which to place the output.
- **mode** (*str*) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.

- **cval** (*scalar*) – Value to fill past edges of input if mode is `constant`. Default is `0.0`.
- **origin** (*scalar or tuple of scalar*) – The origin parameter controls the placement of the filter, relative to the center of the current element of the input. Default of 0 is equivalent to `(0,)*input.ndim`.

**Returns** The result of greyscale dilation.

**Return type** `cupy.ndarray`

**See also:**

`scipy.ndimage.grey_dilation()`

### `cupyx.scipy.ndimage.grey_erosion`

`cupyx.scipy.ndimage.grey_erosion(input, size=None, footprint=None, structure=None, output=None, mode='reflect', cval=0.0, origin=0)`

Calculates a greyscale erosion.

#### Parameters

- **input** (`cupy.ndarray`) – The input array.
- **size** (*tuple of ints*) – Shape of a flat and full structuring element used for the greyscale erosion. Optional if `footprint` or `structure` is provided.
- **footprint** (*array of ints*) – Positions of non-infinite elements of a flat structuring element used for greyscale erosion. Non-zero values give the set of neighbors of the center over which minimum is chosen.
- **structure** (*array of ints*) – Structuring element used for the greyscale erosion. `structure` may be a non-flat structuring element.
- **output** (`cupy.ndarray`, *dtype* or `None`) – The array in which to place the output.
- **mode** (`str`) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (*scalar*) – Value to fill past edges of input if mode is `constant`. Default is `0.0`.
- **origin** (*scalar or tuple of scalar*) – The origin parameter controls the placement of the filter, relative to the center of the current element of the input. Default of 0 is equivalent to `(0,)*input.ndim`.

**Returns** The result of greyscale erosion.

**Return type** `cupy.ndarray`

**See also:**

`scipy.ndimage.grey_erosion()`



### cupyx.scipy.ndimage.grey\_opening

`cupyx.scipy.ndimage.grey_opening(input, size=None, footprint=None, structure=None, output=None, mode='reflect', cval=0.0, origin=0)`

Calculates a multi-dimensional greyscale opening.

#### Parameters

- **input** (`cupy.ndarray`) – The input array.
- **size** (*tuple of ints*) – Shape of a flat and full structuring element used for the greyscale opening. Optional if `footprint` or `structure` is provided.
- **footprint** (*array of ints*) – Positions of non-infinite elements of a flat structuring element used for greyscale opening. Non-zero values give the set of neighbors of the center over which opening is chosen.
- **structure** (*array of ints*) – Structuring element used for the greyscale opening. structure may be a non-flat structuring element.
- **output** (`cupy.ndarray`, *dtype* or `None`) – The array in which to place the output.
- **mode** (`str`) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (*scalar*) – Value to fill past edges of input if mode is constant. Default is 0.0.
- **origin** (*scalar or tuple of scalar*) – The origin parameter controls the placement of the filter, relative to the center of the current element of the input. Default of 0 is equivalent to (0,)\*input.ndim.

**Returns** The result of greyscale opening.

**Return type** `cupy.ndarray`

See also:

`scipy.ndimage.grey_opening()`

### cupyx.scipy.ndimage.iterate\_structure

`cupyx.scipy.ndimage.iterate_structure(structure, iterations, origin=None)`

Iterate a structure by dilating it with itself.

#### Parameters

- **structure** (*array\_like*) – Structuring element (an array of bools, for example), to be dilated with itself.
- **iterations** (*int*) – The number of dilations performed on the structure with itself.
- **origin** (*int or tuple of int, optional*) – If origin is None, only the iterated structure is returned. If not, a tuple of the iterated structure and the modified origin is returned.

**Returns** A new structuring element obtained by dilating `structure` (`iterations - 1`) times with itself.

**Return type** `cupy.ndarray`

See also:

`scipy.ndimage.iterate_structure()`

### cupyx.scipy.ndimage.morphological\_gradient

`cupyx.scipy.ndimage.morphological_gradient`(*input*, *size=None*, *footprint=None*, *structure=None*, *output=None*, *mode='reflect'*, *cval=0.0*, *origin=0*)

Multidimensional morphological gradient.

The morphological gradient is calculated as the difference between a dilation and an erosion of the input with a given structuring element.

#### Parameters

- **input** (`cupy.ndarray`) – The input array.
- **size** (*tuple of ints*) – Shape of a flat and full structuring element used for the morphological gradient. Optional if **footprint** or **structure** is provided.
- **footprint** (*array of ints*) – Positions of non-infinite elements of a flat structuring element used for morphological gradient. Non-zero values give the set of neighbors of the center over which opening is chosen.
- **structure** (*array of ints*) – Structuring element used for the morphological gradient. structure may be a non-flat structuring element.
- **output** (`cupy.ndarray`, *dtype* or *None*) – The array in which to place the output.
- **mode** (*str*) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (*scalar*) – Value to fill past edges of input if mode is constant. Default is 0.0.
- **origin** (*scalar or tuple of scalar*) – The origin parameter controls the placement of the filter, relative to the center of the current element of the input. Default of 0 is equivalent to (0,)\*input.ndim.

**Returns** The morphological gradient of the input.

**Return type** `cupy.ndarray`

See also:

`scipy.ndimage.morphological_gradient()`

### cupyx.scipy.ndimage.morphological\_laplace

`cupyx.scipy.ndimage.morphological_laplace`(*input*, *size=None*, *footprint=None*, *structure=None*, *output=None*, *mode='reflect'*, *cval=0.0*, *origin=0*)

Multidimensional morphological laplace.

#### Parameters

- **input** (`cupy.ndarray`) – The input array.
- **size** (*tuple of ints*) – Shape of a flat and full structuring element used for the morphological laplace. Optional if **footprint** or **structure** is provided.
- **footprint** (*array of ints*) – Positions of non-infinite elements of a flat structuring element used for morphological laplace. Non-zero values give the set of neighbors of the center over which opening is chosen.
- **structure** (*array of ints*) – Structuring element used for the morphological laplace. structure may be a non-flat structuring element.
- **output** (`cupy.ndarray`, *dtype* or *None*) – The array in which to place the output.

- **mode** (*str*) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (*scalar*) – Value to fill past edges of input if mode is constant. Default is 0.0.
- **origin** (*scalar or tuple of scalar*) – The origin parameter controls the placement of the filter, relative to the center of the current element of the input. Default of 0 is equivalent to (0,)\*input.ndim.

**Returns** The morphological laplace of the input.

**Return type** *cupy.ndarray*

See also:

`scipy.ndimage.morphological_laplace()`

### cupyx.scipy.ndimage.white\_tophat

`cupyx.scipy.ndimage.white_tophat(input, size=None, footprint=None, structure=None, output=None, mode='reflect', cval=0.0, origin=0)`

Multidimensional white tophat filter.

#### Parameters

- **input** (*cupy.ndarray*) – The input array.
- **size** (*tuple of ints*) – Shape of a flat and full structuring element used for the white tophat. Optional if **footprint** or **structure** is provided.
- **footprint** (*array of ints*) – Positions of non-infinite elements of a flat structuring element used for the white tophat. Non-zero values give the set of neighbors of the center over which opening is chosen.
- **structure** (*array of ints*) – Structuring element used for the white tophat. **structure** may be a non-flat structuring element.
- **output** (*cupy.ndarray, dtype or None*) – The array in which to place the output.
- **mode** (*str*) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (*scalar*) – Value to fill past edges of input if mode is constant. Default is 0.0.
- **origin** (*scalar or tuple of scalar*) – The origin parameter controls the placement of the filter, relative to the center of the current element of the input. Default of 0 is equivalent to (0,)\*input.ndim.

**Returns** Result of the filter of input with structure.

**Return type** *cupy.ndarray*

See also:

`scipy.ndimage.white_tophat()`

## OpenCV mode

`cupyx.scipy.ndimage` supports additional mode, `opencv`. If it is given, the function performs like `cv2.warpAffine` or `cv2.resize`. Example:

```
import cupyx.scipy.ndimage
import cupy as cp
import cv2

im = cv2.imread('TODO') # pls fill in your image path

trans_mat = cp.eye(4)
trans_mat[0][0] = trans_mat[1][1] = 0.5

smaller_shape = (im.shape[0] // 2, im.shape[1] // 2, 3)
smaller = cp.zeros(smaller_shape) # preallocate memory for resized image

cupyx.scipy.ndimage.affine_transform(im, trans_mat, output_shape=smaller_shape,
                                     output=smaller, mode='opencv')

cv2.imwrite('smaller.jpg', cp.asnumpy(smaller)) # smaller image saved locally
```

### 5.4.5 Sparse matrices (`cupyx.scipy.sparse`)

---

**Hint:** SciPy API Reference: Sparse matrices (`scipy.sparse`)

---

CuPy supports sparse matrices using `cuSPARSE`. These matrices have the same interfaces of SciPy's sparse matrices.

#### Conversion to/from SciPy sparse matrices

`cupyx.scipy.sparse.*_matrix` and `scipy.sparse.*_matrix` are not implicitly convertible to each other. That means, SciPy functions cannot take `cupyx.scipy.sparse.*_matrix` objects as inputs, and vice versa.

- To convert SciPy sparse matrices to CuPy, pass it to the constructor of each CuPy sparse matrix class.
- To convert CuPy sparse matrices to SciPy, use `get` method of each CuPy sparse matrix class.

Note that converting between CuPy and SciPy incurs data transfer between the host (CPU) device and the GPU device, which is costly in terms of performance.

#### Conversion to/from CuPy ndarrays

- To convert CuPy ndarray to CuPy sparse matrices, pass it to the constructor of each CuPy sparse matrix class.
- To convert CuPy sparse matrices to CuPy ndarray, use `toarray` of each CuPy sparse matrix instance (e.g., `cupyx.scipy.sparse.csr_matrix.toarray()`).

Converting between CuPy ndarray and CuPy sparse matrices does not incur data transfer; it is copied inside the GPU device.

## Sparse matrix classes

<code>coo_matrix</code> (arg1[, shape, dtype, copy])	COOrdinate format sparse matrix.
<code>csc_matrix</code> (arg1[, shape, dtype, copy])	Compressed Sparse Column matrix.
<code>csr_matrix</code> (arg1[, shape, dtype, copy])	Compressed Sparse Row matrix.
<code>dia_matrix</code> (arg1[, shape, dtype, copy])	Sparse matrix with DIAGONAL storage.
<code>spmatrix</code> ([maxprint])	Base class of all sparse matrixes.

### cupyx.scipy.sparse.coo\_matrix

**class** cupyx.scipy.sparse.coo\_matrix(*arg1*, *shape=None*, *dtype=None*, *copy=False*)

COOrdinate format sparse matrix.

This can be instantiated in several ways.

**coo\_matrix(D)** D is a rank-2 [cupy.ndarray](#).

**coo\_matrix(S)** S is another sparse matrix. It is equivalent to `S.tocoo()`.

**coo\_matrix((M, N), [dtype])** It constructs an empty matrix whose shape is (M, N). Default dtype is float64.

**coo\_matrix((data, (row, col)))** All data, row and col are one-dimenaional [cupy.ndarray](#).

#### Parameters

- **arg1** – Arguments for the initializer.
- **shape** ([tuple](#)) – Shape of a matrix. Its length must be two.
- **dtype** – Data type. It must be an argument of [numpy.dtype](#).
- **copy** ([bool](#)) – If True, copies of given data are always used.

See also:

[scipy.sparse.coo\\_matrix](#)

#### Methods

**\_\_len\_\_()**

**\_\_iter\_\_()**

**arcsin()**

Elementwise arcsin.

**arcsinh()**

Elementwise arcsinh.

**arctan()**

Elementwise arctan.

**arctanh()**

Elementwise arctanh.

**asformat**(*format*)

Return this matrix in a given sparse format.

**Parameters** `format` (*str* or *None*) – Format you need.

#### **asfptype()**

Upcasts matrix to a floating point format.

When the matrix has floating point type, the method returns itself. Otherwise it makes a copy with floating point type and the same format.

**Returns** A matrix with float type.

**Return type** *cupyx.scipy.sparse.spmatrix*

#### **astype(*t*)**

Casts the array to given data type.

**Parameters** `dtype` – Type specifier.

**Returns** A copy of the array with a given type.

#### **ceil()**

Elementwise ceil.

#### **conj(*copy=True*)**

Element-wise complex conjugation.

If the matrix is of non-complex data type and `copy` is `False`, this method does nothing and the data is not copied.

**Parameters** `copy` (*bool*) – If `True`, the result is guaranteed to not share data with self.

**Returns** The element-wise complex conjugate.

**Return type** *cupyx.scipy.sparse.spmatrix*

#### **conjugate(*copy=True*)**

Element-wise complex conjugation.

If the matrix is of non-complex data type and `copy` is `False`, this method does nothing and the data is not copied.

**Parameters** `copy` (*bool*) – If `True`, the result is guaranteed to not share data with self.

**Returns** The element-wise complex conjugate.

**Return type** *cupyx.scipy.sparse.spmatrix*

#### **copy()**

Returns a copy of this matrix.

No data/indices will be shared between the returned value and current matrix.

#### **count\_nonzero()**

Returns number of non-zero entries.

---

**Note:** This method counts the actual number of non-zero entories, which does not include explicit zero entries. Instead `nnz` returns the number of entries including explicit zeros.

---

**Returns** Number of non-zero entries.

#### **deg2rad()**

Elementwise deg2rad.

**diagonal**(*k=0*)

Returns the k-th diagonal of the matrix.

**Parameters**

- **k** (*int*, *optional*) – Which diagonal to get, corresponding to elements
- **a**[*i* – 0 (the main diagonal).
- **Default** (*i+k*) – 0 (the main diagonal).

**Returns** The k-th diagonal.**Return type** *cupy.ndarray***dot**(*other*)

Ordinary dot product

**eliminate\_zeros**()

Removes zero entries in place.

**expm1**()

Elementwise expm1.

**floor**()

Elementwise floor.

**get**(*stream=None*)

Returns a copy of the array on host memory.

**Parameters** **stream** (*cupy.cuda.Stream*) – CUDA stream object. If it is given, the copy runs asynchronously. Otherwise, the copy is synchronous.

**Returns** Copy of the array on host memory.**Return type** *scipy.sparse.coo\_matrix***getH**()**get\_shape**()

Returns the shape of the matrix.

**Returns** Shape of the matrix.**Return type** *tuple***getformat**()**getmaxprint**()**getnnz**(*axis=None*)

Returns the number of stored values, including explicit zeros.

**log1p**()

Elementwise log1p.

**maximum**(*other*)**mean**(*axis=None, dtype=None, out=None*)

Compute the arithmetic mean along the specified axis.

**Parameters** **axis** (*int* or *None*) – Axis along which the sum is computed. If it is *None*, it computes the average of all the elements. Select from {*None*, 0, 1, -2, -1}.

**Returns** Summed array.**Return type** *cupy.ndarray*

See also:

`scipy.sparse.spmatrix.mean()`

**minimum**(*other*)

**multiply**(*other*)

Point-wise multiplication by another matrix

**power**(*n*, *dtype=None*)

Elementwise power function.

#### Parameters

- **n** – Exponent.
- **dtype** – Type specifier.

**rad2deg**()

Elementwise rad2deg.

**reshape**(*shape*, *order='C'*)

Gives a new shape to a sparse matrix without changing its data.

**rint**()

Elementwise rint.

**set\_shape**(*shape*)

**setdiag**(*values*, *k=0*)

Set diagonal or off-diagonal elements of the array.

#### Parameters

- **values** (`ndarray`) – New values of the diagonal elements. Values may have any length. If the diagonal is longer than values, then the remaining diagonal entries will not be set. If values are longer than the diagonal, then the remaining values are ignored. If a scalar value is given, all of the diagonal is set to it.
- **k** (`int`, *optional*) – Which off-diagonal to set, corresponding to elements `a[i,i+k]`. Default: 0 (the main diagonal).

**sign**()

Elementwise sign.

**sin**()

Elementwise sin.

**sinh**()

Elementwise sinh.

**sqrt**()

Elementwise sqrt.

**sum**(*axis=None*, *dtype=None*, *out=None*)

Sums the matrix elements over a given axis.

#### Parameters

- **axis** (int or None) – Axis along which the sum is computed. If it is None, it computes the sum of all the elements. Select from {None, 0, 1, -2, -1}.
- **dtype** – The type of returned matrix. If it is not specified, type of the array is used.
- **out** (`cupy.ndarray`) – Output matrix.



**Returns** Summed array.

**Return type** `cupy.ndarray`

**See also:**

`scipy.sparse.spmatrix.sum()`

**sum\_duplicates()**

Eliminate duplicate matrix entries by adding them together.

**Warning:** When sorting the indices, CuPy follows the convention of cuSPARSE, which is different from that of SciPy. Therefore, the order of the output indices may differ:

```
>>> #      1 0 0
>>> # A = 1 1 0
>>> #      1 1 1
>>> data = cupy.array([1, 1, 1, 1, 1, 1], 'f')
>>> row = cupy.array([0, 1, 1, 2, 2, 2], 'i')
>>> col = cupy.array([0, 0, 1, 0, 1, 2], 'i')
>>> A = cupyx.scipy.sparse.coo_matrix((data, (row, col)),
...                                  shape=(3, 3))
>>> a = A.get()
>>> A.sum_duplicates()
>>> a.sum_duplicates() # a is scipy.sparse.coo_matrix
>>> A.row
array([0, 1, 1, 2, 2, 2], dtype=int32)
>>> a.row
array([0, 1, 2, 1, 2, 2], dtype=int32)
>>> A.col
array([0, 0, 1, 0, 1, 2], dtype=int32)
>>> a.col
array([0, 0, 0, 1, 1, 2], dtype=int32)
```

**Warning:** Calling this function might synchronize the device.

**See also:**

`scipy.sparse.coo_matrix.sum_duplicates()`

**tan()**

Elementwise tan.

**tanh()**

Elementwise tanh.

**toarray**(*order=None, out=None*)

Returns a dense matrix representing the same value.

**Parameters**

- **order** (*str*) – Not supported.
- **out** – Not supported.

**Returns** Dense array representing the same value.

**Return type** *cupy.ndarray*

**See also:**

*scipy.sparse.coo\_matrix.toarray()*

**tobsr**(*blocksize=None, copy=False*)

Convert this matrix to Block Sparse Row format.

**tocoo**(*copy=False*)

Converts the matrix to COOrdinate format.

**Parameters** **copy** (*bool*) – If *False*, it shares data arrays as much as possible.

**Returns** Converted matrix.

**Return type** *cupyx.scipy.sparse.coo\_matrix*

**tocsc**(*copy=False*)

Converts the matrix to Compressed Sparse Column format.

**Parameters** **copy** (*bool*) – If *False*, it shares data arrays as much as possible. Actually this option is ignored because all arrays in a matrix cannot be shared in coo to csc conversion.

**Returns** Converted matrix.

**Return type** *cupyx.scipy.sparse.csc\_matrix*

**tocsr**(*copy=False*)

Converts the matrix to Compressed Sparse Row format.

**Parameters** **copy** (*bool*) – If *False*, it shares data arrays as much as possible. Actually this option is ignored because all arrays in a matrix cannot be shared in coo to csr conversion.

**Returns** Converted matrix.

**Return type** *cupyx.scipy.sparse.csr\_matrix*

**todense**(*order=None, out=None*)

Return a dense matrix representation of this matrix.

**todia**(*copy=False*)

Convert this matrix to sparse DIAgonal format.

**todok**(*copy=False*)

Convert this matrix to Dictionary Of Keys format.

**tolil**(*copy=False*)

Convert this matrix to LIInked List format.

**transpose**(*axes=None, copy=False*)

Returns a transpose matrix.

**Parameters**

- **axes** – This option is not supported.
- **copy** (*bool*) – If *True*, a returned matrix shares no data. Otherwise, it shared data arrays as much as possible.

**Returns** Transpose matrix.

**Return type** *cupyx.scipy.sparse.spmatrix*

**trunc**()

Elementwise trunc.

```

__eq__(other)
    Return self==value.

__ne__(other)
    Return self!=value.

__lt__(other)
    Return self<value.

__le__(other)
    Return self<=value.

__gt__(other)
    Return self>value.

__ge__(other)
    Return self>=value.

__nonzero__()

__bool__()

```

### Attributes

**A**  
Dense ndarray representation of this matrix.  
This property is equivalent to `toarray()` method.

**H**

**T**

**device**  
CUDA device on which this array resides.

**dtype**  
Data type of the matrix.

**format = 'coo'**

**ndim**

**nnz**

**shape**

**size**

### cupyx.scipy.sparse.csc\_matrix

**class** cupyx.scipy.sparse.**csc\_matrix**(*arg1*, *shape=None*, *dtype=None*, *copy=False*)  
Compressed Sparse Column matrix.

This can be instantiated in several ways.

**csc\_matrix(D)** D is a rank-2 [cupy.ndarray](#).

**csc\_matrix(S)** S is another sparse matrix. It is equivalent to `S.tocsc()`.

**csc\_matrix((M, N), [dtype])** It constructs an empty matrix whose shape is (M, N). Default dtype is float64.

**csc\_matrix((data, (row, col)))** All data, row and col are one-dimensional [cupy.ndarray](#).

**csc\_matrix((data, indices, indptr))** All data, indices and indptr are one-dimensional [cupy.ndarray](#).

#### Parameters

- **arg1** – Arguments for the initializer.
- **shape** ([tuple](#)) – Shape of a matrix. Its length must be two.
- **dtype** – Data type. It must be an argument of [numpy.dtype](#).
- **copy** ([bool](#)) – If True, copies of given arrays are always used.

See also:

[scipy.sparse.csc\\_matrix](#)

#### Methods

**\_\_getitem\_\_**(*key*)

**\_\_setitem\_\_**(*key, x*)

**\_\_len\_\_**()

**\_\_iter\_\_**()

**arcsin**()

Elementwise arcsin.

**arcsinh**()

Elementwise arcsinh.

**arctan**()

Elementwise arctan.

**arctanh**()

Elementwise arctanh.

**argmax**(*axis=None, out=None*)

Returns indices of maximum elements along an axis.

Implicit zero elements are taken into account. If there are several maximum values, the index of the first occurrence is returned. If NaN values occur in the matrix, the output defaults to a zero entry for the row/column in which the NaN occurs.

#### Parameters

- **axis** ([int](#)) – {-2, -1, 0, 1, None} (optional) Axis along which the argmax is computed. If None (default), index of the maximum element in the flatten data is returned.
- **out** ([None](#)) – (optional) This argument is in the signature *solely* for NumPy compatibility reasons. Do not pass in anything except for the default value, as this argument is not used.

**Returns** Indices of maximum elements. If array, its size along *axis* is 1.

**Return type** ([cupy.ndarray](#) or [int](#))

**argmin**(*axis=None, out=None*)

Returns indices of minimum elements along an axis.

Implicit zero elements are taken into account. If there are several minimum values, the index of the first occurrence is returned. If NaN values occur in the matrix, the output defaults to a zero entry for the row/column in which the NaN occurs.

**Parameters**

- **axis** (*int*) – {-2, -1, 0, 1, None} (optional) Axis along which the argmin is computed. If None (default), index of the minimum element in the flattened data is returned.
- **out** (*None*) – (optional) This argument is in the signature *solely* for NumPy compatibility reasons. Do not pass in anything except for the default value, as this argument is not used.

**Returns** Indices of minimum elements. If matrix, its size along axis is 1.

**Return type** (`cupy.ndarray` or `int`)

**asformat**(*format*)

Return this matrix in a given sparse format.

**Parameters** **format** (*str* or *None*) – Format you need.

**asfptype**()

Upcasts matrix to a floating point format.

When the matrix has floating point type, the method returns itself. Otherwise it makes a copy with floating point type and the same format.

**Returns** A matrix with float type.

**Return type** `cupyx.scipy.sparse.spmatrix`

**astype**(*t*)

Casts the array to given data type.

**Parameters** **dtype** – Type specifier.

**Returns** A copy of the array with a given type.

**ceil**()

Elementwise ceil.

**conj**(*copy=True*)

Element-wise complex conjugation.

If the matrix is of non-complex data type and *copy* is False, this method does nothing and the data is not copied.

**Parameters** **copy** (*bool*) – If True, the result is guaranteed to not share data with self.

**Returns** The element-wise complex conjugate.

**Return type** `cupyx.scipy.sparse.spmatrix`

**conjugate**(*copy=True*)

Element-wise complex conjugation.

If the matrix is of non-complex data type and *copy* is False, this method does nothing and the data is not copied.

**Parameters** **copy** (*bool*) – If True, the result is guaranteed to not share data with self.

**Returns** The element-wise complex conjugate.

**Return type** `cupyx.scipy.sparse.spmatrix`

**copy()**

Returns a copy of this matrix.

No data/indices will be shared between the returned value and current matrix.

**count\_nonzero()**

Returns number of non-zero entries.

---

**Note:** This method counts the actual number of non-zero entories, which does not include explicit zero entries. Instead `nnz` returns the number of entries including explicit zeros.

---

**Returns** Number of non-zero entries.

**deg2rad()**

Elementwise `deg2rad`.

**diagonal(*k=0*)**

Returns the *k*-th diagonal of the matrix.

**Parameters**

- **k** (*int*, *optional*) – Which diagonal to get, corresponding to elements
- **a**[*i* – 0 (the main diagonal).
- **Default** (*i+k*]) – 0 (the main diagonal).

**Returns** The *k*-th diagonal.

**Return type** *cupy.ndarray*

**dot(*other*)**

Ordinary dot product

**eliminate\_zeros()**

Removes zero entories in place.

**expm1()**

Elementwise `expm1`.

**floor()**

Elementwise floor.

**get(*stream=None*)**

Returns a copy of the array on host memory.

**Warning:** You need to install SciPy to use this method.

**Parameters** **stream** (*cupy.cuda.Stream*) – CUDA stream object. If it is given, the copy runs asynchronously. Otherwise, the copy is synchronous.

**Returns** Copy of the array on host memory.

**Return type** *scipy.sparse.csc\_matrix*

**getH()****get\_shape()**

Returns the shape of the matrix.

**Returns** Shape of the matrix.

**Return type** `tuple`

**getcol**(*i*)

Returns a copy of column *i* of the matrix, as a (*m* x 1) CSC matrix (column vector).

**Parameters** *i* (`integer`) – Column

**Returns** Sparse matrix with single column

**Return type** `cupyx.scipy.sparse.csc_matrix`

**getformat**()

**getmaxprint**()

**getnnz**(*axis=None*)

Returns the number of stored values, including explicit zeros.

**Parameters** *axis* – Not supported yet.

**Returns** The number of stored values.

**Return type** `int`

**getrow**(*i*)

Returns a copy of row *i* of the matrix, as a (1 x *n*) CSR matrix (row vector).

**Parameters** *i* (`integer`) – Row

**Returns** Sparse matrix with single row

**Return type** `cupyx.scipy.sparse.csc_matrix`

**log1p**()

Elementwise log1p.

**max**(*axis=None, out=None, \*, explicit=False*)

Returns the maximum of the matrix or maximum along an axis.

**Parameters**

- **axis** (`int`) – {-2, -1, 0, 1, `None`} (optional) Axis along which the sum is computed. The default is to compute the maximum over all the matrix elements, returning a scalar (i.e. `axis = None`).
- **out** (`None`) – (optional) This argument is in the signature *solely* for NumPy compatibility reasons. Do not pass in anything except for the default value, as this argument is not used.
- **explicit** (`bool`) – Return the maximum value explicitly specified and ignore all implicit zero entries. If the dimension has no explicit values, a zero is then returned to indicate that it is the only implicit value. This parameter is experimental and may change in the future.

**Returns** Maximum of *a*. If *axis* is `None`, the result is a scalar value. If *axis* is given, the result is an array of dimension *a.ndim* - 1. This differs from numpy for computational efficiency.

**Return type** (`cupy.ndarray` or `float`)

**See also:**

`min` : The minimum value of a sparse matrix along a given axis.

**See also:**

`numpy.matrix.max` : NumPy's implementation of `max` for matrices

**maximum**(*other*)

**mean**(*axis=None, dtype=None, out=None*)

Compute the arithmetic mean along the specified axis.

**Parameters** **axis** (int or None) – Axis along which the sum is computed. If it is None, it computes the average of all the elements. Select from {None, 0, 1, -2, -1}.

**Returns** Summed array.

**Return type** *cupy.ndarray*

**See also:**

*scipy.sparse.spmatrix.mean()*

**min**(*axis=None, out=None, \*, explicit=False*)

Returns the minimum of the matrix or maximum along an axis.

**Parameters**

- **axis** (*int*) – {-2, -1, 0, 1, None} (optional) Axis along which the sum is computed. The default is to compute the minimum over all the matrix elements, returning a scalar (i.e. *axis = None*).
- **out** (*None*) – (optional) This argument is in the signature *solely* for NumPy compatibility reasons. Do not pass in anything except for the default value, as this argument is not used.
- **explicit** (*bool*) – Return the minimum value explicitly specified and ignore all implicit zero entries. If the dimension has no explicit values, a zero is then returned to indicate that it is the only implicit value. This parameter is experimental and may change in the future.

**Returns** Minimum of a. If *axis* is None, the result is a scalar value. If *axis* is given, the result is an array of dimension *a.ndim - 1*. This differs from numpy for computational efficiency.

**Return type** (*cupy.ndarray* or *float*)

**See also:**

*max* : The maximum value of a sparse matrix along a given axis.

**See also:**

*numpy.matrix.min* : NumPy's implementation of 'min' for matrices

**minimum**(*other*)

**multiply**(*other*)

Point-wise multiplication by another matrix

**power**(*n, dtype=None*)

Elementwise power function.

**Parameters**

- **n** – Exponent.
- **dtype** – Type specifier.

**rad2deg**()

Elementwise rad2deg.

**reshape**(*shape, order='C'*)

Gives a new shape to a sparse matrix without changing its data.

**rint**()

Elementwise rint.

**set\_shape**(*shape*)



**setdiag**(*values*, *k=0*)

Set diagonal or off-diagonal elements of the array.

**Parameters**

- **values** (`cupy.ndarray`) – New values of the diagonal elements. Values may have any length. If the diagonal is longer than values, then the remaining diagonal entries will not be set. If values is longer than the diagonal, then the remaining values are ignored. If a scalar value is given, all of the diagonal is set to it.
- **k** (`int`, *optional*) – Which diagonal to set, corresponding to elements `a[i, i+k]`. Default: 0 (the main diagonal).

**sign**()

Elementwise sign.

**sin**()

Elementwise sin.

**sinh**()

Elementwise sinh.

**sort\_indices**()

Sorts the indices of this matrix *in place*.

**Warning:** Calling this function might synchronize the device.

**sorted\_indices**()

Return a copy of this matrix with sorted indices

**Warning:** Calling this function might synchronize the device.

**sqrt**()

Elementwise sqrt.

**sum**(*axis=None*, *dtype=None*, *out=None*)

Sums the matrix elements over a given axis.

**Parameters**

- **axis** (int or None) – Axis along which the sum is computed. If it is None, it computes the sum of all the elements. Select from {None, 0, 1, -2, -1}.
- **dtype** – The type of returned matrix. If it is not specified, type of the array is used.
- **out** (`cupy.ndarray`) – Output matrix.

**Returns** Summed array.

**Return type** `cupy.ndarray`

**See also:**

`scipy.sparse.spmatrix.sum()`

**sum\_duplicates**()

Eliminate duplicate matrix entries by adding them together.

---

**Note:** This is an *in place* operation.

---

**Warning:** Calling this function might synchronize the device.

**See also:**

`scipy.sparse.csr_matrix.sum_duplicates()`, `scipy.sparse.csc_matrix.sum_duplicates()`

**tan()**  
Elementwise tan.

**tanh()**  
Elementwise tanh.

**toarray**(*order=None, out=None*)  
Returns a dense matrix representing the same value.

**Parameters**

- **order** (`{'C', 'F', None}`) – Whether to store data in C (row-major) order or F (column-major) order. Default is C-order.
- **out** – Not supported.

**Returns** Dense array representing the same matrix.

**Return type** `cupy.ndarray`

**See also:**

`scipy.sparse.csc_matrix.toarray()`

**tobsr**(*blocksize=None, copy=False*)  
Convert this matrix to Block Sparse Row format.

**tocoo**(*copy=False*)  
Converts the matrix to COOrdinate format.

**Parameters** **copy** (`bool`) – If `False`, it shares data arrays as much as possible.

**Returns** Converted matrix.

**Return type** `cupyx.scipy.sparse.coo_matrix`

**tocsc**(*copy=None*)  
Converts the matrix to Compressed Sparse Column format.

**Parameters** **copy** (`bool`) – If `False`, the method returns itself. Otherwise it makes a copy of the matrix.

**Returns** Converted matrix.

**Return type** `cupyx.scipy.sparse.csc_matrix`

**tocsr**(*copy=False*)  
Converts the matrix to Compressed Sparse Row format.

**Parameters** **copy** (`bool`) – If `False`, it shares data arrays as much as possible. Actually this option is ignored because all arrays in a matrix cannot be shared in csr to csc conversion.

**Returns** Converted matrix.

**Return type** *cupyx.scipy.sparse.csr\_matrix*

**todense**(*order=None, out=None*)

Return a dense matrix representation of this matrix.

**todia**(*copy=False*)

Convert this matrix to sparse DIAgonal format.

**todok**(*copy=False*)

Convert this matrix to Dictionary Of Keys format.

**tolil**(*copy=False*)

Convert this matrix to LInked List format.

**transpose**(*axes=None, copy=False*)

Returns a transpose matrix.

#### Parameters

- **axes** – This option is not supported.
- **copy** (*bool*) – If *True*, a returned matrix shares no data. Otherwise, it shared data arrays as much as possible.

**Returns** Transpose matrix.

**Return type** *cupyx.scipy.sparse.spmatrix*

**trunc**()

Elementwise trunc.

**\_\_eq\_\_**(*other*)

Return self==value.

**\_\_ne\_\_**(*other*)

Return self!=value.

**\_\_lt\_\_**(*other*)

Return self<value.

**\_\_le\_\_**(*other*)

Return self<=value.

**\_\_gt\_\_**(*other*)

Return self>value.

**\_\_ge\_\_**(*other*)

Return self>=value.

**\_\_nonzero\_\_**()

**\_\_bool\_\_**()

## Attributes

**A**

Dense ndarray representation of this matrix.

This property is equivalent to `toarray()` method.

**H**

**T**

**device**

CUDA device on which this array resides.

**dtype**

Data type of the matrix.

**format = 'csc'**

**has\_canonical\_format**

Determine whether the matrix has sorted indices and no duplicates.

**Returns** bool: True if the above applies, otherwise False.

---

**Note:** `has_canonical_format` implies `has_sorted_indices`, so if the latter flag is False, so will the former be; if the former is found True, the latter flag is also set.

---

**Warning:** Getting this property might synchronize the device.

**has\_sorted\_indices**

Determine whether the matrix has sorted indices.

**Returns**

**bool:** True if the indices of the matrix are in sorted order, otherwise False.

**Warning:** Getting this property might synchronize the device.

**ndim**

**nnz**

**shape**

**size**

**cupyx.scipy.sparse.csr\_matrix**

**class** cupyx.scipy.sparse.csr\_matrix(*arg1*, *shape=None*, *dtype=None*, *copy=False*)

Compressed Sparse Row matrix.

This can be instantiated in several ways.

**csr\_matrix(D)** D is a rank-2 [cupy.ndarray](#).

**csr\_matrix(S)** S is another sparse matrix. It is equivalent to `S.tocsr()`.

**csr\_matrix((M, N), [dtype])** It constructs an empty matrix whose shape is (M, N). Default dtype is float64.

**csr\_matrix((data, (row, col)))** All data, row and col are one-dimensional [cupy.ndarray](#).

**csr\_matrix((data, indices, indptr))** All data, indices and indptr are one-dimensional [cupy.ndarray](#).

**Parameters**

- **arg1** – Arguments for the initializer.
- **shape** (*tuple*) – Shape of a matrix. Its length must be two.
- **dtype** – Data type. It must be an argument of [numpy.dtype](#).
- **copy** (*bool*) – If True, copies of given arrays are always used.

See also:

[scipy.sparse.csr\\_matrix](#)

**Methods**

**\_\_getitem\_\_**(*key*)

**\_\_setitem\_\_**(*key*, *x*)

**\_\_len\_\_**()

**\_\_iter\_\_**()

**arcsin**()

Elementwise arcsin.

**arcsinh**()

Elementwise arcsinh.

**arctan**()

Elementwise arctan.

**arctanh**()

Elementwise arctanh.

**argmax**(*axis=None*, *out=None*)

Returns indices of maximum elements along an axis.

Implicit zero elements are taken into account. If there are several maximum values, the index of the first occurrence is returned. If NaN values occur in the matrix, the output defaults to a zero entry for the row/column in which the NaN occurs.

**Parameters**

- **axis** (*int*) – {-2, -1, 0, 1, None} (optional) Axis along which the argmax is computed. If None (default), index of the maximum element in the flatten data is returned.
- **out** (*None*) – (optional) This argument is in the signature *solely* for NumPy compatibility reasons. Do not pass in anything except for the default value, as this argument is not used.

**Returns** Indices of maximum elements. If array, its size along **axis** is 1.

**Return type** (cupy.ndarray or *int*)

**argmin**(*axis=None, out=None*)

Returns indices of minimum elements along an axis.

Implicit zero elements are taken into account. If there are several minimum values, the index of the first occurrence is returned. If NaN values occur in the matrix, the output defaults to a zero entry for the row/column in which the NaN occurs.

**Parameters**

- **axis** (*int*) – {-2, -1, 0, 1, None} (optional) Axis along which the argmin is computed. If None (default), index of the minimum element in the flatten data is returned.
- **out** (*None*) – (optional) This argument is in the signature *solely* for NumPy compatibility reasons. Do not pass in anything except for the default value, as this argument is not used.

**Returns** Indices of minimum elements. If matrix, its size along **axis** is 1.

**Return type** (cupy.ndarray or *int*)

**asformat**(*format*)

Return this matrix in a given sparse format.

**Parameters** **format** (*str* or *None*) – Format you need.

**asfptype**()

Upcasts matrix to a floating point format.

When the matrix has floating point type, the method returns itself. Otherwise it makes a copy with floating point type and the same format.

**Returns** A matrix with float type.

**Return type** *cupyx.scipy.sparse.spmatrix*

**astype**(*t*)

Casts the array to given data type.

**Parameters** **dtype** – Type specifier.

**Returns** A copy of the array with a given type.

**ceil**()

Elementwise ceil.

**conj**(*copy=True*)

Element-wise complex conjugation.

If the matrix is of non-complex data type and *copy* is False, this method does nothing and the data is not copied.

**Parameters** **copy** (*bool*) – If True, the result is guaranteed to not share data with self.

**Returns** The element-wise complex conjugate.

**Return type** *cupyx.scipy.sparse.spmatrix*

**conjugate**(*copy=True*)

Element-wise complex conjugation.

If the matrix is of non-complex data type and *copy* is False, this method does nothing and the data is not copied.

**Parameters** *copy* (*bool*) – If True, the result is guaranteed to not share data with self.

**Returns** The element-wise complex conjugate.

**Return type** *cupyx.scipy.sparse.spmatrix*

**copy**()

Returns a copy of this matrix.

No data/indices will be shared between the returned value and current matrix.

**count\_nonzero**()

Returns number of non-zero entries.

---

**Note:** This method counts the actual number of non-zero entories, which does not include explicit zero entries. Instead *nnz* returns the number of entries including explicit zeros.

---

**Returns** Number of non-zero entries.

**deg2rad**()

Elementwise deg2rad.

**diagonal**(*k=0*)

Returns the k-th diagonal of the matrix.

**Parameters**

- *k* (*int*, *optional*) – Which diagonal to get, corresponding to elements
- *a*[*i* – 0 (the main diagonal).
- **Default** (*i+k*) – 0 (the main diagonal).

**Returns** The k-th diagonal.

**Return type** *cupy.ndarray*

**dot**(*other*)

Ordinary dot product

**eliminate\_zeros**()

Removes zero entories in place.

**expm1**()

Elementwise expm1.

**floor**()

Elementwise floor.

**get**(*stream=None*)

Returns a copy of the array on host memory.

**Parameters** *stream* (*cupy.cuda.Stream*) – CUDA stream object. If it is given, the copy runs asynchronously. Otherwise, the copy is synchronous.

**Returns** Copy of the array on host memory.

**Return type** *scipy.sparse.csr\_matrix*

**getH()**

**get\_shape()**

Returns the shape of the matrix.

**Returns** Shape of the matrix.

**Return type** *tuple*

**getcol(i)**

Returns a copy of column *i* of the matrix, as a (m x 1) CSR matrix (column vector).

**Parameters** *i* (*integer*) – Column

**Returns** Sparse matrix with single column

**Return type** *cupyx.scipy.sparse.csr\_matrix*

**getformat()**

**getmaxprint()**

**getnnz(axis=None)**

Returns the number of stored values, including explicit zeros.

**Parameters** *axis* – Not supported yet.

**Returns** The number of stored values.

**Return type** *int*

**getrow(i)**

Returns a copy of row *i* of the matrix, as a (1 x n) CSR matrix (row vector).

**Parameters** *i* (*integer*) – Row

**Returns** Sparse matrix with single row

**Return type** *cupyx.scipy.sparse.csr\_matrix*

**log1p()**

Elementwise log1p.

**max(axis=None, out=None, \*, explicit=False)**

Returns the maximum of the matrix or maximum along an axis.

**Parameters**

- **axis** (*int*) – {-2, -1, 0, 1, None} (optional) Axis along which the sum is computed. The default is to compute the maximum over all the matrix elements, returning a scalar (i.e. *axis = None*).
- **out** (*None*) – (optional) This argument is in the signature *solely* for NumPy compatibility reasons. Do not pass in anything except for the default value, as this argument is not used.
- **explicit** (*bool*) – Return the maximum value explicitly specified and ignore all implicit zero entries. If the dimension has no explicit values, a zero is then returned to indicate that it is the only implicit value. This parameter is experimental and may change in the future.

**Returns** Maximum of *a*. If *axis* is *None*, the result is a scalar value. If *axis* is given, the result is an array of dimension *a.ndim - 1*. This differs from numpy for computational efficiency.

**Return type** (*cupy.ndarray* or *float*)



**See also:**

`min` : The minimum value of a sparse matrix along a given axis.

**See also:**

`numpy.matrix.max` : NumPy's implementation of `max` for matrices

**maximum**(*other*)

**mean**(*axis=None, dtype=None, out=None*)

Compute the arithmetic mean along the specified axis.

**Parameters** **axis** (int or None) – Axis along which the sum is computed. If it is None, it computes the average of all the elements. Select from {None, 0, 1, -2, -1}.

**Returns** Summed array.

**Return type** *cupy.ndarray*

**See also:**

`scipy.sparse.spmatrix.mean()`

**min**(*axis=None, out=None, \*, explicit=False*)

Returns the minimum of the matrix or maximum along an axis.

**Parameters**

- **axis** (*int*) – {-2, -1, 0, 1, None} (optional) Axis along which the sum is computed. The default is to compute the minimum over all the matrix elements, returning a scalar (i.e. `axis = None`).
- **out** (*None*) – (optional) This argument is in the signature *solely* for NumPy compatibility reasons. Do not pass in anything except for the default value, as this argument is not used.
- **explicit** (*bool*) – Return the minimum value explicitly specified and ignore all implicit zero entries. If the dimension has no explicit values, a zero is then returned to indicate that it is the only implicit value. This parameter is experimental and may change in the future.

**Returns** Minimum of `a`. If `axis` is None, the result is a scalar value. If `axis` is given, the result is an array of dimension `a.ndim - 1`. This differs from `numpy` for computational efficiency.

**Return type** (*cupy.ndarray* or *float*)

**See also:**

`max` : The maximum value of a sparse matrix along a given axis.

**See also:**

`numpy.matrix.min` : NumPy's implementation of 'min' for matrices

**minimum**(*other*)

**multiply**(*other*)

Point-wise multiplication by another matrix, vector or scalar

**power**(*n, dtype=None*)

Elementwise power function.

**Parameters**

- **n** – Exponent.
- **dtype** – Type specifier.

**rad2deg()**

Elementwise rad2deg.

**reshape**(*shape*, *order*='C')

Gives a new shape to a sparse matrix without changing its data.

**rint()**

Elementwise rint.

**set\_shape**(*shape*)

**setdiag**(*values*, *k*=0)

Set diagonal or off-diagonal elements of the array.

**sign()**

Elementwise sign.

**sin()**

Elementwise sin.

**sinh()**

Elementwise sinh.

**sort\_indices()**

Sorts the indices of this matrix *in place*.

**Warning:** Calling this function might synchronize the device.

**sorted\_indices()**

Return a copy of this matrix with sorted indices

**Warning:** Calling this function might synchronize the device.

**sqrt()**

Elementwise sqrt.

**sum**(*axis*=None, *dtype*=None, *out*=None)

Sums the matrix elements over a given axis.

**Parameters**

- **axis** (int or None) – Axis along which the sum is computed. If it is None, it computes the sum of all the elements. Select from {None, 0, 1, -2, -1}.
- **dtype** – The type of returned matrix. If it is not specified, type of the array is used.
- **out** ([cupy.ndarray](#)) – Output matrix.

**Returns** Summed array.

**Return type** [cupy.ndarray](#)

**See also:**

[scipy.sparse.spmatrix.sum\(\)](#)

**sum\_duplicates()**

Eliminate duplicate matrix entries by adding them together.

---

**Note:** This is an *in place* operation.

---

**Warning:** Calling this function might synchronize the device.

**See also:**

`scipy.sparse.csr_matrix.sum_duplicates()`, `scipy.sparse.csc_matrix.sum_duplicates()`

**tan()**  
Elementwise tan.

**tanh()**  
Elementwise tanh.

**toarray**(*order=None, out=None*)  
Returns a dense matrix representing the same value.

**Parameters**

- **order** (`{'C', 'F', None}`) – Whether to store data in C (row-major) order or F (column-major) order. Default is C-order.
- **out** – Not supported.

**Returns** Dense array representing the same matrix.

**Return type** `cupy.ndarray`

**See also:**

`scipy.sparse.csr_matrix.toarray()`

**tobsr**(*blocksize=None, copy=False*)  
Convert this matrix to Block Sparse Row format.

**tocoo**(*copy=False*)  
Converts the matrix to COOrdinate format.

**Parameters** **copy** (`bool`) – If `False`, it shares data arrays as much as possible.

**Returns** Converted matrix.

**Return type** `cupyx.scipy.sparse.coo_matrix`

**tocsc**(*copy=False*)  
Converts the matrix to Compressed Sparse Column format.

**Parameters** **copy** (`bool`) – If `False`, it shares data arrays as much as possible. Actually this option is ignored because all arrays in a matrix cannot be shared in csr to csc conversion.

**Returns** Converted matrix.

**Return type** `cupyx.scipy.sparse.csc_matrix`

**tocsr**(*copy=False*)  
Converts the matrix to Compressed Sparse Row format.

**Parameters** **copy** (`bool`) – If `False`, the method returns itself. Otherwise it makes a copy of the matrix.

**Returns** Converted matrix.

**Return type** *cupyx.scipy.sparse.csr\_matrix*

**todense**(*order=None, out=None*)

Return a dense matrix representation of this matrix.

**todia**(*copy=False*)

Convert this matrix to sparse DIAgonal format.

**todok**(*copy=False*)

Convert this matrix to Dictionary Of Keys format.

**tolil**(*copy=False*)

Convert this matrix to LInked List format.

**transpose**(*axes=None, copy=False*)

Returns a transpose matrix.

#### Parameters

- **axes** – This option is not supported.
- **copy** (*bool*) – If *True*, a returned matrix shares no data. Otherwise, it shared data arrays as much as possible.

**Returns** Transpose matrix.

**Return type** *cupyx.scipy.sparse.spmatrix*

**trunc**()

Elementwise trunc.

**\_\_eq\_\_**(*other*)

Return self==value.

**\_\_ne\_\_**(*other*)

Return self!=value.

**\_\_lt\_\_**(*other*)

Return self<value.

**\_\_le\_\_**(*other*)

Return self<=value.

**\_\_gt\_\_**(*other*)

Return self>value.

**\_\_ge\_\_**(*other*)

Return self>=value.

**\_\_nonzero\_\_**()

**\_\_bool\_\_**()

## Attributes

**A**

Dense ndarray representation of this matrix.

This property is equivalent to `toarray()` method.

**H**

**T**

**device**

CUDA device on which this array resides.

**dtype**

Data type of the matrix.

**format = 'csr'**

**has\_canonical\_format**

Determine whether the matrix has sorted indices and no duplicates.

**Returns** bool: True if the above applies, otherwise False.

---

**Note:** `has_canonical_format` implies `has_sorted_indices`, so if the latter flag is False, so will the former be; if the former is found True, the latter flag is also set.

---

**Warning:** Getting this property might synchronize the device.

**has\_sorted\_indices**

Determine whether the matrix has sorted indices.

**Returns**

**bool:** True if the indices of the matrix are in sorted order, otherwise False.

**Warning:** Getting this property might synchronize the device.

**ndim**

**nnz**

**shape**

**size**

## cupyx.scipy.sparse.dia\_matrix

**class** cupyx.scipy.sparse.dia\_matrix(*arg1*, *shape=None*, *dtype=None*, *copy=False*)  
Sparse matrix with DIAgonal storage.

Now it has only one initializer format below:

dia\_matrix((data, offsets))

### Parameters

- **arg1** – Arguments for the initializer.
- **shape** (*tuple*) – Shape of a matrix. Its length must be two.
- **dtype** – Data type. It must be an argument of `numpy.dtype`.
- **copy** (*bool*) – If True, copies of given arrays are always used.

See also:

`scipy.sparse.dia_matrix`

### Methods

`__len__()`

`__iter__()`

`arcsin()`

Elementwise arcsin.

`arcsinh()`

Elementwise arcsinh.

`arctan()`

Elementwise arctan.

`arctanh()`

Elementwise arctanh.

`asformat(format)`

Return this matrix in a given sparse format.

**Parameters** **format** (*str* or *None*) – Format you need.

`asfptype()`

Upcasts matrix to a floating point format.

When the matrix has floating point type, the method returns itself. Otherwise it makes a copy with floating point type and the same format.

**Returns** A matrix with float type.

**Return type** `cupyx.scipy.sparse.spmatrix`

`astype(t)`

Casts the array to given data type.

**Parameters** **dtype** – Type specifier.

**Returns** A copy of the array with a given type.

`ceil()`

Elementwise ceil.

**conj**(*copy=True*)

Element-wise complex conjugation.

If the matrix is of non-complex data type and *copy* is False, this method does nothing and the data is not copied.

**Parameters** **copy** (*bool*) – If True, the result is guaranteed to not share data with self.

**Returns** The element-wise complex conjugate.

**Return type** *cupyx.scipy.sparse.spmatrix*

**conjugate**(*copy=True*)

Element-wise complex conjugation.

If the matrix is of non-complex data type and *copy* is False, this method does nothing and the data is not copied.

**Parameters** **copy** (*bool*) – If True, the result is guaranteed to not share data with self.

**Returns** The element-wise complex conjugate.

**Return type** *cupyx.scipy.sparse.spmatrix*

**copy**()

Returns a copy of this matrix.

No data/indices will be shared between the returned value and current matrix.

**count\_nonzero**()

Returns number of non-zero entries.

---

**Note:** This method counts the actual number of non-zero entories, which does not include explicit zero entries. Instead `nnz` returns the number of entries including explicit zeros.

---

**Returns** Number of non-zero entries.

**deg2rad**()

Elementwise deg2rad.

**diagonal**(*k=0*)

Returns the k-th diagonal of the matrix.

**Parameters**

- **k** (*int*, *optional*) – Which diagonal to get, corresponding to elements
- **a**[*i* – 0 (the main diagonal).
- **Default** (*i+k*]) – 0 (the main diagonal).

**Returns** The k-th diagonal.

**Return type** *cupy.ndarray*

**dot**(*other*)

Ordinary dot product

**expm1**()

Elementwise expm1.

**floor**()

Elementwise floor.

**get**(*stream=None*)

Returns a copy of the array on host memory.

**Parameters** **stream** ([cupy.cuda.Stream](#)) – CUDA stream object. If it is given, the copy runs asynchronously. Otherwise, the copy is synchronous.

**Returns** Copy of the array on host memory.

**Return type** [scipy.sparse.dia\\_matrix](#)

**getH**()

**get\_shape**()

Returns the shape of the matrix.

**Returns** Shape of the matrix.

**Return type** [tuple](#)

**getformat**()

**getmaxprint**()

**getnnz**(*axis=None*)

Returns the number of stored values, including explicit zeros.

**Parameters** **axis** – Not supported yet.

**Returns** The number of stored values.

**Return type** [int](#)

**log1p**()

Elementwise log1p.

**maximum**(*other*)

**mean**(*axis=None, dtype=None, out=None*)

Compute the arithmetic mean along the specified axis.

**Parameters** **axis** (int or None) – Axis along which the sum is computed. If it is None, it computes the average of all the elements. Select from {None, 0, 1, -2, -1}.

**Returns** Summed array.

**Return type** [cupy.ndarray](#)

**See also:**

[scipy.sparse.spmatrix.mean\(\)](#)

**minimum**(*other*)

**multiply**(*other*)

Point-wise multiplication by another matrix

**power**(*n, dtype=None*)

Elementwise power function.

**Parameters**

- **n** – Exponent.
- **dtype** – Type specifier.

**rad2deg**()

Elementwise rad2deg.



**reshape**(*shape*, *order*='C')

Gives a new shape to a sparse matrix without changing its data.

**rint**()

Elementwise rint.

**set\_shape**(*shape*)

**setdiag**(*values*, *k*=0)

Set diagonal or off-diagonal elements of the array.

#### Parameters

- **values** ([cupy.ndarray](#)) – New values of the diagonal elements. Values may have any length. If the diagonal is longer than values, then the remaining diagonal entries will not be set. If values is longer than the diagonal, then the remaining values are ignored. If a scalar value is given, all of the diagonal is set to it.
- **k** ([int](#), *optional*) – Which diagonal to set, corresponding to elements  $a[i, i+k]$ . Default: 0 (the main diagonal).

**sign**()

Elementwise sign.

**sin**()

Elementwise sin.

**sinh**()

Elementwise sinh.

**sqrt**()

Elementwise sqrt.

**sum**(*axis*=None, *dtype*=None, *out*=None)

Sums the matrix elements over a given axis.

#### Parameters

- **axis** (int or None) – Axis along which the sum is computed. If it is None, it computes the sum of all the elements. Select from {None, 0, 1, -2, -1}.
- **dtype** – The type of returned matrix. If it is not specified, type of the array is used.
- **out** ([cupy.ndarray](#)) – Output matrix.

**Returns** Summed array.

**Return type** [cupy.ndarray](#)

**See also:**

[scipy.sparse.spmatrix.sum\(\)](#)

**tan**()

Elementwise tan.

**tanh**()

Elementwise tanh.

**toarray**(*order*=None, *out*=None)

Returns a dense matrix representing the same value.

**tobsr**(*blocksize*=None, *copy*=False)

Convert this matrix to Block Sparse Row format.

**tocoo**(*copy=False*)

Convert this matrix to COOrdinate format.

**tocsc**(*copy=False*)

Converts the matrix to Compressed Sparse Column format.

**Parameters** **copy** (*bool*) – If *False*, it shares data arrays as much as possible. Actually this option is ignored because all arrays in a matrix cannot be shared in dia to csc conversion.

**Returns** Converted matrix.

**Return type** *cupyx.scipy.sparse.csc\_matrix*

**tocsr**(*copy=False*)

Converts the matrix to Compressed Sparse Row format.

**Parameters** **copy** (*bool*) – If *False*, it shares data arrays as much as possible. Actually this option is ignored because all arrays in a matrix cannot be shared in dia to csr conversion.

**Returns** Converted matrix.

**Return type** *cupyx.scipy.sparse.csc\_matrix*

**todense**(*order=None, out=None*)

Return a dense matrix representation of this matrix.

**todia**(*copy=False*)

Convert this matrix to sparse DIAgonal format.

**todok**(*copy=False*)

Convert this matrix to Dictionary Of Keys format.

**tolil**(*copy=False*)

Convert this matrix to LInked List format.

**transpose**(*axes=None, copy=False*)

Reverses the dimensions of the sparse matrix.

**trunc**()

Elementwise trunc.

**\_\_eq\_\_**(*other*)

Return self==value.

**\_\_ne\_\_**(*other*)

Return self!=value.

**\_\_lt\_\_**(*other*)

Return self<value.

**\_\_le\_\_**(*other*)

Return self<=value.

**\_\_gt\_\_**(*other*)

Return self>value.

**\_\_ge\_\_**(*other*)

Return self>=value.

**\_\_nonzero\_\_**()

**\_\_bool\_\_**()

## Attributes

**A**

Dense ndarray representation of this matrix.

This property is equivalent to `toarray()` method.

**H**

**T**

**device**

CUDA device on which this array resides.

**dtype**

Data type of the matrix.

**format = 'dia'**

**ndim**

**nnz**

**shape**

**size**

## cupyx.scipy.sparse.spmatrix

**class** cupyx.scipy.sparse.**spmatrix**(*maxprint=50*)

Base class of all sparse matrixes.

See `scipy.sparse.spmatrix`

## Methods

**\_\_len\_\_()**

**\_\_iter\_\_()**

**asformat**(*format*)

Return this matrix in a given sparse format.

**Parameters** *format* (*str* or *None*) – Format you need.

**asfptype()**

Upcasts matrix to a floating point format.

When the matrix has floating point type, the method returns itself. Otherwise it makes a copy with floating point type and the same format.

**Returns** A matrix with float type.

**Return type** `cupyx.scipy.sparse.spmatrix`

**astype**(*t*)

Casts the array to given data type.

**Parameters** *t* – Type specifier.

**Returns** A copy of the array with the given type and the same format.

**Return type** `cupyx.scipy.sparse.spmatrix`

**conj**(*copy=True*)

Element-wise complex conjugation.

If the matrix is of non-complex data type and *copy* is False, this method does nothing and the data is not copied.

**Parameters** **copy** (*bool*) – If True, the result is guaranteed to not share data with self.

**Returns** The element-wise complex conjugate.

**Return type** *cupyx.scipy.sparse.spmatrix*

**conjugate**(*copy=True*)

Element-wise complex conjugation.

If the matrix is of non-complex data type and *copy* is False, this method does nothing and the data is not copied.

**Parameters** **copy** (*bool*) – If True, the result is guaranteed to not share data with self.

**Returns** The element-wise complex conjugate.

**Return type** *cupyx.scipy.sparse.spmatrix*

**copy**()

Returns a copy of this matrix.

No data/indices will be shared between the returned value and current matrix.

**count\_nonzero**()

Number of non-zero entries, equivalent to

**diagonal**(*k=0*)

Returns the k-th diagonal of the matrix.

**Parameters**

- **k** (*int*, *optional*) – Which diagonal to get, corresponding to elements
- **a**[*i* – 0 (the main diagonal).
- **Default** (*i+k*) – 0 (the main diagonal).

**Returns** The k-th diagonal.

**Return type** *cupy.ndarray*

**dot**(*other*)

Ordinary dot product

**get**(*stream=None*)

Return a copy of the array on host memory.

**Parameters** **stream** (*cupy.cuda.Stream*) – CUDA stream object. If it is given, the copy runs asynchronously. Otherwise, the copy is synchronous.

**Returns** An array on host memory.

**Return type** *scipy.sparse.spmatrix*

**getH**()**get\_shape**()**getformat**()**getmaxprint**()

**getnnz**(*axis=None*)

Number of stored values, including explicit zeros.

**maximum**(*other*)

**mean**(*axis=None, dtype=None, out=None*)

Compute the arithmetic mean along the specified axis.

Returns the average of the matrix elements. The average is taken over all elements in the matrix by default, otherwise over the specified axis. *float64* intermediate and return values are used for integer inputs.

#### Parameters

- **-2** (*axis*) – optional Axis along which the mean is computed. The default is to compute the mean of all elements in the matrix (i.e., *axis = None*).
- **-1** – optional Axis along which the mean is computed. The default is to compute the mean of all elements in the matrix (i.e., *axis = None*).
- **0** – optional Axis along which the mean is computed. The default is to compute the mean of all elements in the matrix (i.e., *axis = None*).
- **1** – optional Axis along which the mean is computed. The default is to compute the mean of all elements in the matrix (i.e., *axis = None*).
- **None** – optional Axis along which the mean is computed. The default is to compute the mean of all elements in the matrix (i.e., *axis = None*).
- **dtype** (*dtype*) – optional Type to use in computing the mean. For integer inputs, the default is *float64*; for floating point inputs, it is the same as the input dtype.
- **out** (*cupy.ndarray*) – optional Alternative output matrix in which to place the result. It must have the same shape as the expected output, but the type of the output values will be cast if necessary.

**Returns** Output array of means

**Return type** *m* (*cupy.ndarray*)

**See also:**

`scipy.sparse.spmatrix.mean()`

**minimum**(*other*)

**multiply**(*other*)

Point-wise multiplication by another matrix

**power**(*n, dtype=None*)

**reshape**(*shape, order='C'*)

Gives a new shape to a sparse matrix without changing its data.

**set\_shape**(*shape*)

**setdiag**(*values, k=0*)

Set diagonal or off-diagonal elements of the array.

#### Parameters

- **values** (*cupy.ndarray*) – New values of the diagonal elements. Values may have any length. If the diagonal is longer than values, then the remaining diagonal entries will not be set. If values is longer than the diagonal, then the remaining values are ignored. If a scalar value is given, all of the diagonal is set to it.

- **k** (*int*, *optional*) – Which diagonal to set, corresponding to elements  $a[i, i+k]$ . Default: 0 (the main diagonal).

**sum**(*axis=None, dtype=None, out=None*)

Sums the matrix elements over a given axis.

#### Parameters

- **axis** (int or None) – Axis along which the sum is computed. If it is None, it computes the sum of all the elements. Select from {None, 0, 1, -2, -1}.
- **dtype** – The type of returned matrix. If it is not specified, type of the array is used.
- **out** (*cupy.ndarray*) – Output matrix.

**Returns** Summed array.

**Return type** *cupy.ndarray*

**See also:**

`scipy.sparse.spmatrix.sum()`

**toarray**(*order=None, out=None*)

Return a dense ndarray representation of this matrix.

**tobsr**(*blocksize=None, copy=False*)

Convert this matrix to Block Sparse Row format.

**tocoo**(*copy=False*)

Convert this matrix to COOrdinate format.

**tocsc**(*copy=False*)

Convert this matrix to Compressed Sparse Column format.

**tocsr**(*copy=False*)

Convert this matrix to Compressed Sparse Row format.

**todense**(*order=None, out=None*)

Return a dense matrix representation of this matrix.

**todia**(*copy=False*)

Convert this matrix to sparse DIAgonal format.

**todok**(*copy=False*)

Convert this matrix to Dictionary Of Keys format.

**tolil**(*copy=False*)

Convert this matrix to LInked List format.

**transpose**(*axes=None, copy=False*)

Reverses the dimensions of the sparse matrix.

**\_\_eq\_\_**(*other*)

Return self==value.

**\_\_ne\_\_**(*other*)

Return self!=value.

**\_\_lt\_\_**(*other*)

Return self<value.

**\_\_le\_\_**(*other*)

Return self<=value.

**\_\_gt\_\_**(*other*)  
Return self>value.

**\_\_ge\_\_**(*other*)  
Return self>=value.

**\_\_nonzero\_\_**()

**\_\_bool\_\_**()

## Attributes

**A**  
Dense ndarray representation of this matrix.  
This property is equivalent to `toarray()` method.

**H**

**T**

**device**  
CUDA device on which this array resides.

**ndim**

**nnz**

**shape**

**size**

## Functions

Building sparse matrices:

<code>eye(m[, n, k, dtype, format])</code>	Creates a sparse matrix with ones on diagonal.
<code>identity(n[, dtype, format])</code>	Creates an identity matrix in sparse format.
<code>kron(A, B[, format])</code>	Kronecker product of sparse matrices A and B.
<code>diags(diagonals[, offsets, shape, format, dtype])</code>	Construct a sparse matrix from diagonals.
<code>spdiags(data, diags, m, n[, format])</code>	Creates a sparse matrix from diagonals.
<code>tril(A[, k, format])</code>	Returns the lower triangular portion of a matrix in sparse format
<code>triu(A[, k, format])</code>	Returns the upper triangular portion of a matrix in sparse format
<code>bmat(blocks[, format, dtype])</code>	Builds a sparse matrix from sparse sub-blocks
<code>hstack(blocks[, format, dtype])</code>	Stacks sparse matrices horizontally (column wise)
<code>vstack(blocks[, format, dtype])</code>	Stacks sparse matrices vertically (row wise)
<code>rand(m, n[, density, format, dtype, ...])</code>	Generates a random sparse matrix.
<code>random(m, n[, density, format, dtype, ...])</code>	Generates a random sparse matrix.

### cupyx.scipy.sparse.eye

`cupyx.scipy.sparse.eye(m, n=None, k=0, dtype='d', format=None)`

Creates a sparse matrix with ones on diagonal.

#### Parameters

- **m** (*int*) – Number of rows.
- **n** (*int* or *None*) – Number of columns. If it is *None*, it makes a square matrix.
- **k** (*int*) – Diagonal to place ones on.
- **dtype** – Type of a matrix to create.
- **format** (*str* or *None*) – Format of the result, e.g. `format="csr"`.

**Returns** Created sparse matrix.

**Return type** *cupyx.scipy.sparse.spmatrix*

**See also:**

`scipy.sparse.eye()`

### cupyx.scipy.sparse.identity

`cupyx.scipy.sparse.identity(n, dtype='d', format=None)`

Creates an identity matrix in sparse format.

---

**Note:** Currently it only supports csr, csc and coo formats.

---

#### Parameters

- **n** (*int*) – Number of rows and columns.
- **dtype** – Type of a matrix to create.
- **format** (*str* or *None*) – Format of the result, e.g. `format="csr"`.

**Returns** Created identity matrix.

**Return type** *cupyx.scipy.sparse.spmatrix*

**See also:**

`scipy.sparse.identity()`

### cupyx.scipy.sparse.kron

`cupyx.scipy.sparse.kron(A, B, format=None)`

Kronecker product of sparse matrices A and B.

#### Parameters

- **A** (*cupyx.scipy.sparse.spmatrix*) – a sparse matrix.
- **B** (*cupyx.scipy.sparse.spmatrix*) – a sparse matrix.
- **format** (*str*) – the format of the returned sparse matrix.



**Returns** Generated sparse matrix with the specified format.

**Return type** `cupyx.scipy.sparse.spmatrix`

**See also:**

`scipy.sparse.kron()`

## `cupyx.scipy.sparse.diags`

`cupyx.scipy.sparse.diags(diagonals, offsets=0, shape=None, format=None, dtype=None)`

Construct a sparse matrix from diagonals.

### Parameters

- **diagonals** (*sequence of array\_like*) – Sequence of arrays containing the matrix diagonals, corresponding to *offsets*.
- **offsets** (*sequence of int or an int*) –

#### Diagonals to set:

- $k = 0$  the main diagonal (default)
- $k > 0$  the  $k$ -th upper diagonal
- $k < 0$  the  $k$ -th lower diagonal
- **shape** (*tuple of int*) – Shape of the result. If omitted, a square matrix large enough to contain the diagonals is returned.
- **format** (`{"dia", "csr", "csc", "lil", ...}`) – Matrix format of the result. By default (`format=None`) an appropriate sparse matrix format is returned. This choice is subject to change.
- **dtype** (*dtype*) – Data type of the matrix.

**Returns** Generated matrix.

**Return type** `cupyx.scipy.sparse.spmatrix`

### Notes

This function differs from `spdiags` in the way it handles off-diagonals.

The result from `diags` is the sparse equivalent of:

```
cupy.diag(diagonals[0], offsets[0])
+ ...
+ cupy.diag(diagonals[k], offsets[k])
```

Repeated diagonal offsets are disallowed.

### cupyx.scipy.sparse.spdiags

`cupyx.scipy.sparse.spdiags(data, diags, m, n, format=None)`

Creates a sparse matrix from diagonals.

#### Parameters

- **data** (`cupy.ndarray`) – Matrix diagonals stored row-wise.
- **diags** (`cupy.ndarray`) – Diagonals to set.
- **m** (`int`) – Number of rows.
- **n** (`int`) – Number of cols.
- **format** (`str` or `None`) – Sparse format, e.g. `format="csr"`.

**Returns** Created sparse matrix.

**Return type** `cupyx.scipy.sparse.spmatrix`

See also:

`scipy.sparse.spdiags()`

### cupyx.scipy.sparse.tril

`cupyx.scipy.sparse.tril(A, k=0, format=None)`

Returns the lower triangular portion of a matrix in sparse format

#### Parameters

- **A** (`cupy.ndarray` or `cupyx.scipy.sparse.spmatrix`) – Matrix whose lower triangular portion is desired.
- **k** (`integer`) – The top-most diagonal of the lower triangle.
- **format** (`string`) – Sparse format of the result, e.g. ‘csr’, ‘csc’, etc.

**Returns** Lower triangular portion of A in sparse format.

**Return type** `cupyx.scipy.sparse.spmatrix`

See also:

`scipy.sparse.tril()`

### cupyx.scipy.sparse.triu

`cupyx.scipy.sparse.triu(A, k=0, format=None)`

Returns the upper triangular portion of a matrix in sparse format

#### Parameters

- **A** (`cupy.ndarray` or `cupyx.scipy.sparse.spmatrix`) – Matrix whose upper triangular portion is desired.
- **k** (`integer`) – The bottom-most diagonal of the upper triangle.
- **format** (`string`) – Sparse format of the result, e.g. ‘csr’, ‘csc’, etc.

**Returns** Upper triangular portion of A in sparse format.

**Return type** `cupyx.scipy.sparse.spmatrix`

See also:

`scipy.sparse.triu()`

### `cupyx.scipy.sparse.bmat`

`cupyx.scipy.sparse.bmat(blocks, format=None, dtype=None)`

Builds a sparse matrix from sparse sub-blocks

#### Parameters

- **blocks** (*array\_like*) – Grid of sparse matrices with compatible shapes. An entry of `None` implies an all-zero matrix.
- **format** (*{'bsr', 'coo', 'csc', 'csr', 'dia', 'dok', 'lil'}, optional*) – The sparse format of the result (e.g. “csr”). By default an appropriate sparse matrix format is returned. This choice is subject to change.
- **dtype** (*dtype, optional*) – The data-type of the output matrix. If not given, the dtype is determined from that of *blocks*.

**Returns** `bmat` (sparse matrix)

See also:

`scipy.sparse.bmat()`

### Examples

```
>>> from cupy import array
>>> from cupyx.scipy.sparse import csr_matrix, bmat
>>> A = csr_matrix(array([[1., 2.], [3., 4.]])
>>> B = csr_matrix(array([[5.], [6.]])
>>> C = csr_matrix(array([[7.]])
>>> bmat([[A, B], [None, C]]).toarray()
array([[1., 2., 5.],
       [3., 4., 6.],
       [0., 0., 7.]])
>>> bmat([[A, None], [None, C]]).toarray()
array([[1., 2., 0.],
       [3., 4., 0.],
       [0., 0., 7.]])
```

### `cupyx.scipy.sparse.hstack`

`cupyx.scipy.sparse.hstack(blocks, format=None, dtype=None)`

Stacks sparse matrices horizontally (column wise)

#### Parameters

- **blocks** (*sequence of cupyx.scipy.sparse.spmatrix*) – sparse matrices to stack
- **format** (*str*) – sparse format of the result (e.g. “csr”) by default an appropriate sparse matrix format is returned. This choice is subject to change.
- **dtype** (*dtype, optional*) – The data-type of the output matrix. If not given, the dtype is determined from that of *blocks*.

**Returns** the stacked sparse matrix

**Return type** `cupyx.scipy.sparse.spmatrix`

See also:

`scipy.sparse.hstack()`

### Examples

```
>>> from cupy import array
>>> from cupyx.scipy.sparse import csr_matrix, hstack
>>> A = csr_matrix(array([[1., 2.], [3., 4.]])
>>> B = csr_matrix(array([[5.], [6.]])
>>> hstack([A, B]).toarray()
array([[1., 2., 5.],
       [3., 4., 6.]])
```

## `cupyx.scipy.sparse.vstack`

`cupyx.scipy.sparse.vstack(blocks, format=None, dtype=None)`

Stacks sparse matrices vertically (row wise)

### Parameters

- **blocks** (*sequence of cupyx.scipy.sparse.spmatrix*) – sparse matrices to stack
- **format** (*str, optional*) – sparse format of the result (e.g. “csr”) by default an appropriate sparse matrix format is returned. This choice is subject to change.
- **dtype** (*dtype, optional*) – The data-type of the output matrix. If not given, the dtype is determined from that of *blocks*.

**Returns** the stacked sparse matrix

**Return type** `cupyx.scipy.sparse.spmatrix`

See also:

`scipy.sparse.vstack()`

### Examples

```
>>> from cupy import array
>>> from cupyx.scipy.sparse import csr_matrix, vstack
>>> A = csr_matrix(array([[1., 2.], [3., 4.]])
>>> B = csr_matrix(array([[5.], [6.]])
>>> vstack([A, B]).toarray()
array([[1., 2.],
       [3., 4.],
       [5., 6.]])
```

## cupyx.scipy.sparse.rand

`cupyx.scipy.sparse.rand(m, n, density=0.01, format='coo', dtype=None, random_state=None)`

Generates a random sparse matrix.

See `cupyx.scipy.sparse.random()` for detail.

### Parameters

- **m** (*int*) – Number of rows.
- **n** (*int*) – Number of cols.
- **density** (*float*) – Ratio of non-zero entries.
- **format** (*str*) – Matrix format.
- **dtype** (*dtype*) – Type of the returned matrix values.
- **random\_state** (`cupy.random.RandomState` or *int*) – State of random number generator. If an integer is given, the method makes a new state for random number generator and uses it. If it is not given, the default state is used. This state is used to generate random indexes for nonzero entries.

**Returns** Generated matrix.

**Return type** `cupyx.scipy.sparse.spmatrix`

See also:

`scipy.sparse.rand()`

See also:

`cupyx.scipy.sparse.random()`

## cupyx.scipy.sparse.random

`cupyx.scipy.sparse.random(m, n, density=0.01, format='coo', dtype=None, random_state=None, data_rvs=None)`

Generates a random sparse matrix.

This function generates a random sparse matrix. First it selects non-zero elements with given density `density` from  $(m, n)$  elements. So the number of non-zero elements  $k$  is  $k = m * n * density$ . Value of each element is selected with `data_rvs` function.

### Parameters

- **m** (*int*) – Number of rows.
- **n** (*int*) – Number of cols.
- **density** (*float*) – Ratio of non-zero entries.
- **format** (*str*) – Matrix format.
- **dtype** (*dtype*) – Type of the returned matrix values.
- **random\_state** (`cupy.random.RandomState` or *int*) – State of random number generator. If an integer is given, the method makes a new state for random number generator and uses it. If it is not given, the default state is used. This state is used to generate random indexes for nonzero entries.

- **data\_rvs** (*callable*) – A function to generate data for a random matrix. If it is not given, *random\_state.rand* is used.

**Returns** Generated matrix.

**Return type** `cupyx.scipy.sparse.spmatrix`

**See also:**

`scipy.sparse.random()`

Sparse matrix tools:

---

<code>find(A)</code>	Returns the indices and values of the nonzero elements of a matrix
----------------------	--

---

### `cupyx.scipy.sparse.find`

`cupyx.scipy.sparse.find(A)`

Returns the indices and values of the nonzero elements of a matrix

**Parameters** **A** (`cupy.ndarray` or `cupyx.scipy.sparse.spmatrix`) – Matrix whose nonzero elements are desired.

**Returns** It returns (I, J, V). I, J, and V contain respectively the row indices, column indices, and values of the nonzero matrix entries.

**Return type** tuple of `cupy.ndarray`

**See also:**

`scipy.sparse.find()`

Identifying sparse matrices:

---

<code>issparse(x)</code>	Checks if a given matrix is a sparse matrix.
<code>isspmatrix(x)</code>	Checks if a given matrix is a sparse matrix.
<code>isspmatrix_csc(x)</code>	Checks if a given matrix is of CSC format.
<code>isspmatrix_csr(x)</code>	Checks if a given matrix is of CSR format.
<code>isspmatrix_coo(x)</code>	Checks if a given matrix is of COO format.
<code>isspmatrix_dia(x)</code>	Checks if a given matrix is of DIA format.

---

### `cupyx.scipy.sparse.issparse`

`cupyx.scipy.sparse.issparse(x)`

Checks if a given matrix is a sparse matrix.

**Returns** Returns if **x** is `cupyx.scipy.sparse.spmatrix` that is a base class of all sparse matrix classes.

**Return type** `bool`

**cupyx.scipy.sparse.isspmatrix****cupyx.scipy.sparse.isspmatrix**(x)

Checks if a given matrix is a sparse matrix.

**Returns** Returns if x is *cupyx.scipy.sparse.spmatrix* that is a base class of all sparse matrix classes.**Return type** bool**cupyx.scipy.sparse.isspmatrix\_csc****cupyx.scipy.sparse.isspmatrix\_csc**(x)

Checks if a given matrix is of CSC format.

**Returns** Returns if x is *cupyx.scipy.sparse.csc\_matrix*.**Return type** bool**cupyx.scipy.sparse.isspmatrix\_csr****cupyx.scipy.sparse.isspmatrix\_csr**(x)

Checks if a given matrix is of CSR format.

**Returns** Returns if x is *cupyx.scipy.sparse.csr\_matrix*.**Return type** bool**cupyx.scipy.sparse.isspmatrix\_coo****cupyx.scipy.sparse.isspmatrix\_coo**(x)

Checks if a given matrix is of COO format.

**Returns** Returns if x is *cupyx.scipy.sparse.coo\_matrix*.**Return type** bool**cupyx.scipy.sparse.isspmatrix\_dia****cupyx.scipy.sparse.isspmatrix\_dia**(x)

Checks if a given matrix is of DIA format.

**Returns** Returns if x is *cupyx.scipy.sparse.dia\_matrix*.**Return type** bool

## Linear Algebra (`cupyx.scipy.sparse.linalg`)

---

**Hint:** SciPy API Reference: Sparse linear algebra (`scipy.sparse.linalg`)

---

### Abstract linear operators

---

<code>LinearOperator</code> ( <i>shape</i> , <i>matvec</i> [, <i>rmatvec</i> , ...])	Common interface for performing matrix vector products
<code>aslinearoperator</code> ( <i>A</i> )	Return <i>A</i> as a <code>LinearOperator</code> .

---

### `cupyx.scipy.sparse.linalg.LinearOperator`

**class** `cupyx.scipy.sparse.linalg.LinearOperator`(*shape*, *matvec*, *rmatvec*=None, *matmat*=None, *dtype*=None, *rmatmat*=None)

Common interface for performing matrix vector products

To construct a concrete `LinearOperator`, either pass appropriate callables to the constructor of this class, or subclass it.

#### Parameters

- **shape** (*tuple*) – Matrix dimensions (M, N).
- **matvec** (*callable f(v)*) – Returns  $A * v$ .
- **rmatvec** (*callable f(v)*) – Returns  $A^H * v$ , where  $A^H$  is the conjugate transpose of *A*.
- **matmat** (*callable f(V)*) – Returns  $A * V$ , where *V* is a dense matrix with dimensions (N, K).
- **dtype** (*dtype*) – Data type of the matrix.
- **rmatmat** (*callable f(V)*) – Returns  $A^H * V$ , where *V* is a dense matrix with dimensions (M, K).

See also:

`scipy.sparse.linalg.LinearOperator`

#### Methods

**\_\_call\_\_**(*x*)

Call self as a function.

**adjoint**()

Hermitian adjoint.

**dot**(*x*)

Matrix-matrix or matrix-vector multiplication.

**matmat**(*X*)

Matrix-matrix multiplication.



**matvec**(*x*)  
Matrix-vector multiplication.

**rmatmat**(*X*)  
Adjoint matrix-matrix multiplication.

**rmatvec**(*x*)  
Adjoint matrix-vector multiplication.

**transpose**()  
Transpose this linear operator.

**\_\_eq\_\_**(*value*, /)  
Return self==value.

**\_\_ne\_\_**(*value*, /)  
Return self!=value.

**\_\_lt\_\_**(*value*, /)  
Return self<value.

**\_\_le\_\_**(*value*, /)  
Return self<=value.

**\_\_gt\_\_**(*value*, /)  
Return self>value.

**\_\_ge\_\_**(*value*, /)  
Return self>=value.

### Attributes

**H**  
Hermitian adjoint.

**T**  
Transpose this linear operator.

**ndim** = 2

## cupyx.scipy.sparse.linalg.aslinearoperator

**cupyx.scipy.sparse.linalg.aslinearoperator**(*A*)  
Return *A* as a `LinearOperator`.

**Parameters** *A* (*array-like*) – The input array to be converted to a `LinearOperator` object. It may be any of the following types:

- `cupy.ndarray`
- sparse matrix (e.g. `csr_matrix`, `coo_matrix`, etc.)
- `cupyx.scipy.sparse.linalg.LinearOperator`
- object with `.shape` and `.matvec` attributes

**Returns** `LinearOperator` object

**Return type** `cupyx.scipy.sparse.linalg.LinearOperator`

See also:

`scipy.sparse.aslinearoperator`()`

## Matrix norms

---

<code>cupyx.scipy.sparse.linalg.norm(x[, ord, axis])</code>	Norm of a <code>cupy.scipy.spmatrix</code>
---	--

---

### `cupyx.scipy.sparse.linalg.norm`

`cupyx.scipy.sparse.linalg.norm(x, ord=None, axis=None)`

Norm of a `cupy.scipy.spmatrix`

This function is able to return one of seven different sparse matrix norms, depending on the value of the `ord` parameter.

#### Parameters

- **x** (*sparse matrix*) – Input sparse matrix.
- **ord** (*non-zero int, inf, -inf, 'fro', optional*) – Order of the norm (see table under Notes). *inf* means numpy's *inf* object.
- **axis** – (int, 2-tuple of ints, None, optional): If *axis* is an integer, it specifies the axis of *x* along which to compute the vector norms. If *axis* is a 2-tuple, it specifies the axes that hold 2-D matrices, and the matrix norms of these matrices are computed. If *axis* is None then either a vector norm (when *x* is 1-D) or a matrix norm (when *x* is 2-D) is returned.

**Returns** 0-D or 1-D array or norm(s).

**Return type** *ndarray*

See also:

`scipy.sparse.linalg.norm()`

## Solving linear problems

Direct methods for linear equation systems:

---

<code>spsolve(A, b)</code>	Solves a sparse linear system $A \mathbf{x} = \mathbf{b}$
<code>spsolve_triangular(A, b[, lower, ...])</code>	Solves a sparse triangular system $A \mathbf{x} = \mathbf{b}$ .
<code>factorized(A)</code>	Return a function for solving a sparse linear system, with <i>A</i> pre-factorized.

---

### cupyx.scipy.sparse.linalg.spsolve

cupyx.scipy.sparse.linalg.spsolve(*A*, *b*)

Solves a sparse linear system  $A x = b$

**Parameters**

- **A** (cupyx.scipy.sparse.spmatrix) – Sparse matrix with dimension (M, M).
- **b** (cupy.ndarray) – Dense vector or matrix with dimension (M) or (M, 1).

**Returns** Solution to the system  $A x = b$ .

**Return type** cupy.ndarray

### cupyx.scipy.sparse.linalg.spsolve\_triangular

cupyx.scipy.sparse.linalg.spsolve\_triangular(*A*, *b*, *lower=True*, *overwrite\_A=False*,  
overwrite\_b=False, unit\_diagonal=False)

Solves a sparse triangular system  $A x = b$ .

**Parameters**

- **A** (cupyx.scipy.sparse.spmatrix) – Sparse matrix with dimension (M, M).
- **b** (cupy.ndarray) – Dense vector or matrix with dimension (M) or (M, K).
- **lower** (bool) – Whether A is a lower or upper triangular matrix. If True, it is lower triangular, otherwise, upper triangular.
- **overwrite\_A** (bool) – (not supported)
- **overwrite\_b** (bool) – Allows overwriting data in b.
- **unit\_diagonal** (bool) – If True, diagonal elements of A are assumed to be 1 and will not be referenced.

**Returns** Solution to the system  $A x = b$ . The shape is the same as b.

**Return type** cupy.ndarray

### cupyx.scipy.sparse.linalg.factorized

cupyx.scipy.sparse.linalg.factorized(*A*)

Return a function for solving a sparse linear system, with A pre-factorized.

**Parameters** **A** (cupyx.scipy.sparse.spmatrix) – Sparse matrix to factorize.

**Returns** a function to solve the linear system of equations given in A.

**Return type** callable

---

**Note:** This function computes LU decomposition of a sparse matrix on the CPU using *scipy.sparse.linalg.splu*. Therefore, LU decomposition is not accelerated on the GPU. On the other hand, the computation of solving linear equations using the method returned by this function is performed on the GPU.

---

**See also:**

`scipy.sparse.linalg.factorized()`

Iterative methods for linear equation systems:

<code>cg(A, b[, x0, tol, maxiter, M, callback, atol])</code>	Uses Conjugate Gradient iteration to solve $Ax = b$ .
<code>gmres(A, b[, x0, tol, restart, maxiter, M, ...])</code>	Uses Generalized Minimal RESidual iteration to solve $Ax = b$ .

### `cupyx.scipy.sparse.linalg.cg`

`cupyx.scipy.sparse.linalg.cg(A, b, x0=None, tol=1e-05, maxiter=None, M=None, callback=None, atol=None)`

Uses Conjugate Gradient iteration to solve  $Ax = b$ .

#### Parameters

- **A** (`ndarray`, `spmatrix` or `LinearOperator`) – The real or complex matrix of the linear system with shape  $(n, n)$ . A must be a hermitian, positive definitive matrix with type of `cupy.ndarray`, `cupyx.scipy.sparse.spmatrix` or `cupyx.scipy.sparse.linalg.LinearOperator`.
- **b** (`cupy.ndarray`) – Right hand side of the linear system with shape  $(n,)$  or  $(n, 1)$ .
- **x0** (`cupy.ndarray`) – Starting guess for the solution.
- **tol** (`float`) – Tolerance for convergence.
- **maxiter** (`int`) – Maximum number of iterations.
- **M** (`ndarray`, `spmatrix` or `LinearOperator`) – Preconditioner for A. The preconditioner should approximate the inverse of A. M must be `cupy.ndarray`, `cupyx.scipy.sparse.spmatrix` or `cupyx.scipy.sparse.linalg.LinearOperator`.
- **callback** (`function`) – User-specified function to call after each iteration. It is called as `callback(xk)`, where `xk` is the current solution vector.
- **atol** (`float`) – Tolerance for convergence.

**Returns** It returns `x` (`cupy.ndarray`) and `info` (`int`) where `x` is the converged solution and `info` provides convergence information.

**Return type** `tuple`

See also:

`scipy.sparse.linalg.cg()`

### `cupyx.scipy.sparse.linalg.gmres`

`cupyx.scipy.sparse.linalg.gmres(A, b, x0=None, tol=1e-05, restart=None, maxiter=None, M=None, callback=None, atol=None, callback_type=None)`

Uses Generalized Minimal RESidual iteration to solve  $Ax = b$ .

#### Parameters

- **A** (`ndarray`, `spmatrix` or `LinearOperator`) – The real or complex matrix of the linear system with shape  $(n, n)$ . A must be `cupy.ndarray`, `cupyx.scipy.sparse.spmatrix` or `cupyx.scipy.sparse.linalg.LinearOperator`.
- **b** (`cupy.ndarray`) – Right hand side of the linear system with shape  $(n,)$  or  $(n, 1)$ .
- **x0** (`cupy.ndarray`) – Starting guess for the solution.

- **tol** (*float*) – Tolerance for convergence.
- **restart** (*int*) – Number of iterations between restarts. Larger values increase iteration cost, but may be necessary for convergence.
- **maxiter** (*int*) – Maximum number of iterations.
- **M** (*ndarray, spmatrix or LinearOperator*) – Preconditioner for A. The preconditioner should approximate the inverse of A. M must be `cupy.ndarray`, `cupyx.scipy.sparse.spmatrix` or `cupyx.scipy.sparse.linalg.LinearOperator`.
- **callback** (*function*) – User-specified function to call on every restart. It is called as `callback(arg)`, where `arg` is selected by `callback_type`.
- **callback\_type** (*str*) – ‘x’ or ‘pr\_norm’. If ‘x’, the current solution vector is used as an argument of callback function. if ‘pr\_norm’, relative (preconditioned) residual norm is used as an argument.
- **atol** (*float*) – Tolerance for convergence.

**Returns** It returns `x` (`cupy.ndarray`) and `info` (`int`) where `x` is the converged solution and `info` provides convergence information.

**Return type** `tuple`

**Reference:** M. Wang, H. Klie, M. Parashar and H. Sudan, “Solving Sparse Linear Systems on NVIDIA Tesla GPUs”, ICCS 2009 (2009).

**See also:**

`scipy.sparse.linalg.gmres()`

Iterative methods for least-squares problems:

---

<code>lsqr(A, b)</code>	Solves linear system with QR decomposition.
-------------------------	---

---

## `cupyx.scipy.sparse.linalg.lsqr`

`cupyx.scipy.sparse.linalg.lsqr(A, b)`  
Solves linear system with QR decomposition.

Find the solution to a large, sparse, linear system of equations. The function solves  $Ax = b$ . Given two-dimensional matrix A is decomposed into  $Q * R$ .

### Parameters

- **A** (*cupy.ndarray or cupyx.scipy.sparse.csr\_matrix*) – The input matrix with dimension (N, N)
- **b** (*cupy.ndarray*) – Right-hand side vector.

**Returns** Its length must be ten. It has same type elements as SciPy. Only the first element, the solution vector `x`, is available and other elements are expressed as `None` because the implementation of cuSOLVER is different from the one of SciPy. You can easily calculate the fourth element by `norm(b - Ax)` and the ninth element by `norm(x)`.

**Return type** `tuple`

**See also:**

`scipy.sparse.linalg.lsqr()`

## Matrix factorizations

Eigenvalue problems:

---

<code>eigsh(a[, k, which, ncv, maxiter, tol, ...])</code>	Finds $k$ eigenvalues and eigenvectors of the real symmetric matrix.
<code>lobpcg(A, X[, B, M, Y, tol, maxiter, ...])</code>	Locally Optimal Block Preconditioned Conjugate Gradient Method (LOBPCG)

---

### `cupyx.scipy.sparse.linalg.eigsh`

`cupyx.scipy.sparse.linalg.eigsh(a, k=6, *, which='LM', ncv=None, maxiter=None, tol=0, return_eigenvectors=True)`

Finds  $k$  eigenvalues and eigenvectors of the real symmetric matrix.

Solves  $Ax = wx$ , the standard eigenvalue problem for  $w$  eigenvalues with corresponding eigenvectors  $x$ .

#### Parameters

- **a** (`ndarray`, `spmatrix` or `LinearOperator`) – A symmetric square matrix with dimension  $(n, n)$ . `a` must be `cupy.ndarray`, `cupyx.scipy.sparse.spmatrix` or `cupyx.scipy.sparse.linalg.LinearOperator`.
- **k** (`int`) – The number of eigenvalues and eigenvectors to compute. Must be  $1 \leq k < n$ .
- **which** (`str`) – 'LM' or 'LA'. 'LM': finds  $k$  largest (in magnitude) eigenvalues. 'LA': finds  $k$  largest (algebraic) eigenvalues.
- **ncv** (`int`) – The number of Lanczos vectors generated. Must be  $k + 1 < ncv < n$ . If `None`, default value is used.
- **maxiter** (`int`) – Maximum number of Lanczos update iterations. If `None`, default value is used.
- **tol** (`float`) – Tolerance for residuals  $\|Ax - wx\|$ . If `0`, machine precision is used.
- **return\_eigenvectors** (`bool`) – If `True`, returns eigenvectors in addition to eigenvalues.

**Returns** If `return_eigenvectors` is `True`, it returns  $w$  and  $x$  where  $w$  is eigenvalues and  $x$  is eigenvectors. Otherwise, it returns only  $w$ .

**Return type** `tuple`

**See also:**

`scipy.sparse.linalg.eigsh()`

---

**Note:** This function uses the thick-restart Lanczos methods (<https://sdm.lbl.gov/~kewu/ps/trlan.html>).

---

**cupyx.scipy.sparse.linalg.lobpcg**

```
cupyx.scipy.sparse.linalg.lobpcg(A, X, B=None, M=None, Y=None, tol=None, maxiter=None,
                                largest=True, verbosityLevel=0, retLambdaHistory=False,
                                retResidualNormsHistory=False)
```

Locally Optimal Block Preconditioned Conjugate Gradient Method (LOBPCG)

LOBPCG is a preconditioned eigensolver for large symmetric positive definite (SPD) generalized eigenproblems.

**Parameters**

- **A** (*array-like*) – The symmetric linear operator of the problem, usually a sparse matrix. Can be of the following types - `cupy.ndarray` - `cupyx.scipy.sparse.csr_matrix` - `cupy.scipy.sparse.linalg.LinearOperator`
- **X** (`cupy.ndarray`) – Initial approximation to the `k` eigenvectors (non-sparse). If `A` has `shape=(n,n)` then `X` should have `shape=(n,k)`.
- **B** (*array-like*) – The right hand side operator in a generalized eigenproblem. By default, `B = Identity`. Can be of following types: - `cupy.ndarray` - `cupyx.scipy.sparse.csr_matrix` - `cupy.scipy.sparse.linalg.LinearOperator`
- **M** (*array-like*) – Preconditioner to `A`; by default `M = Identity`. `M` should approximate the inverse of `A`. Can be of the following types: - `cupy.ndarray` - `cupyx.scipy.sparse.csr_matrix` - `cupy.scipy.sparse.linalg.LinearOperator`
- **Y** (`cupy.ndarray`) – `n-by-sizeY` matrix of constraints (non-sparse), `sizeY < n`. The iterations will be performed in the `B`-orthogonal complement of the column-space of `Y`. `Y` must be full rank.
- **tol** (*float*) – Solver tolerance (stopping criterion). The default is `tol=n*sqrt(eps)`.
- **maxiter** (*int*) – Maximum number of iterations. The default is `maxiter = 20`.
- **largest** (*bool*) – When `True`, solve for the largest eigenvalues, otherwise the smallest.
- **verbosityLevel** (*int*) – Controls solver output. The default is `verbosityLevel=0`.
- **retLambdaHistory** (*bool*) – Whether to return eigenvalue history. Default is `False`.
- **retResidualNormsHistory** (*bool*) – Whether to return history of residual norms. Default is `False`.

**Returns**

- `w` (`cupy.ndarray`): Array of `k` eigenvalues
- `v` (`cupy.ndarray`) An array of `k` eigenvectors. `v` has the same shape as `X`.
- `lambdas` (list of `cupy.ndarray`): The eigenvalue history, if `retLambdaHistory` is `True`.
- `rnorms` (list of `cupy.ndarray`): The history of residual norms, if `retResidualNormsHistory` is `True`.

**Return type** `tuple`

See also:

`scipy.sparse.linalg.lobpcg()`

---

**Note:** If both `retLambdaHistory` and `retResidualNormsHistory` are `True` the return tuple has the following format (`lambda`, `V`, `lambda history`, `residual norms history`).

---

Singular values problems:

---

<code>svds(a[, k, ncv, tol, which, maxiter, ...])</code>	Finds the largest $k$ singular values/vectors for a sparse matrix.
--	--

---

### `cupyx.scipy.sparse.linalg.svds`

`cupyx.scipy.sparse.linalg.svds(a, k=6, *, ncv=None, tol=0, which='LM', maxiter=None, return_singular_vectors=True)`

Finds the largest  $k$  singular values/vectors for a sparse matrix.

#### Parameters

- **a** (`ndarray`, `spmatrix` or `LinearOperator`) – A real or complex array with dimension  $(m, n)$ . **a** must `cupy.ndarray`, `cupyx.scipy.sparse.spmatrix` or `cupyx.scipy.sparse.linalg.LinearOperator`.
- **k** (`int`) – The number of singular values/vectors to compute. Must be  $1 \leq k < \min(m, n)$ .
- **ncv** (`int`) – The number of Lanczos vectors generated. Must be  $k + 1 < ncv < \min(m, n)$ . If `None`, default value is used.
- **tol** (`float`) – Tolerance for singular values. If `0`, machine precision is used.
- **which** (`str`) – Only 'LM' is supported. 'LM': finds  $k$  largest singular values.
- **maxiter** (`int`) – Maximum number of Lanczos update iterations. If `None`, default value is used.
- **return\_singular\_vectors** (`bool`) – If `True`, returns singular vectors in addition to singular values.

**Returns** If `return_singular_vectors` is `True`, it returns `u`, `s` and `vt` where `u` is left singular vectors, `s` is singular values and `vt` is right singular vectors. Otherwise, it returns only `s`.

**Return type** `tuple`

See also:

`scipy.sparse.linalg.svds()`

---

**Note:** This is a naive implementation using `cupyx.scipy.sparse.linalg.eigsh` as an eigensolver on  $a.H @ a$  or  $a @ a.H$ .

---

Complete or incomplete LU factorizations:

---

<code>splu(A[, permc_spec, diag_pivot_thresh, ...])</code>	Computes the LU decomposition of a sparse square matrix.
<code>spilu(A[, drop_tol, fill_factor, drop_rule, ...])</code>	Computes the incomplete LU decomposition of a sparse square matrix.

---



**cupyx.scipy.sparse.linalg.splu**

`cupyx.scipy.sparse.linalg.splu(A, permc_spec=None, diag_pivot_thresh=None, relax=None, panel_size=None, options={})`

Computes the LU decomposition of a sparse square matrix.

**Parameters**

- **A** (`cupyx.scipy.sparse.spmatrix`) – Sparse matrix to factorize.
- **permc\_spec** (`str`) – (For further augments, see `scipy.sparse.linalg.splu()`)
- **diag\_pivot\_thresh** (`float`) –
- **relax** (`int`) –
- **panel\_size** (`int`) –
- **options** (`dict`) –

**Returns** Object which has a solve method.

**Return type** `cupyx.scipy.sparse.linalg.SuperLU`

---

**Note:** This function LU-decomposes a sparse matrix on the CPU using `scipy.sparse.linalg.splu`. Therefore, LU decomposition is not accelerated on the GPU. On the other hand, the computation of solving linear equations using the solve method, which this function returns, is performed on the GPU.

---

See also:

`scipy.sparse.linalg.splu()`

**cupyx.scipy.sparse.linalg.spilu**

`cupyx.scipy.sparse.linalg.spilu(A, drop_tol=None, fill_factor=None, drop_rule=None, permc_spec=None, diag_pivot_thresh=None, relax=None, panel_size=None, options={})`

Computes the incomplete LU decomposition of a sparse square matrix.

**Parameters**

- **A** (`cupyx.scipy.sparse.spmatrix`) – Sparse matrix to factorize.
- **drop\_tol** (`float`) – (For further augments, see `scipy.sparse.linalg.spilu()`)
- **fill\_factor** (`float`) –
- **drop\_rule** (`str`) –
- **permc\_spec** (`str`) –
- **diag\_pivot\_thresh** (`float`) –
- **relax** (`int`) –
- **panel\_size** (`int`) –
- **options** (`dict`) –

**Returns** Object which has a solve method.

**Return type** `cupyx.scipy.sparse.linalg.SuperLU`

**Note:** This function computes incomplete LU decomposition of a sparse matrix on the CPU using `scipy.sparse.linalg.spilu` (unless you set `fill_factor` to 1). Therefore, incomplete LU decomposition is not accelerated on the GPU. On the other hand, the computation of solving linear equations using the `solve` method, which this function returns, is performed on the GPU.

If you set `fill_factor` to 1, this function computes incomplete LU decomposition on the GPU, but without fill-in or pivoting.

---

**See also:**

`scipy.sparse.linalg.spilu()`

## 5.4.6 Special functions (`cupyx.scipy.special`)

---

**Hint:** SciPy API Reference: Special functions (`scipy.special`)

---

### Bessel functions

<code>j0</code>	Bessel function of the first kind of order 0.
<code>j1</code>	Bessel function of the first kind of order 1.
<code>y0</code>	Bessel function of the second kind of order 0.
<code>y1</code>	Bessel function of the second kind of order 1.
<code>i0</code>	Modified Bessel function of order 0.
<code>i1</code>	Modified Bessel function of order 1.

#### `cupyx.scipy.special.j0`

`cupyx.scipy.special.j0 = <ufunc 'cupyx_scipy_special_j0'>`  
Bessel function of the first kind of order 0.

**See also:**

`scipy.special.j0()`

#### `cupyx.scipy.special.j1`

`cupyx.scipy.special.j1 = <ufunc 'cupyx_scipy_special_j1'>`  
Bessel function of the first kind of order 1.

**See also:**

`scipy.special.j1()`

**cupyx.scipy.special.y0**

`cupyx.scipy.special.y0` = <ufunc 'cupyx\_scipy\_special\_y0'>

Bessel function of the second kind of order 0.

**See also:**

`scipy.special.y0()`

**cupyx.scipy.special.y1**

`cupyx.scipy.special.y1` = <ufunc 'cupyx\_scipy\_special\_y1'>

Bessel function of the second kind of order 1.

**See also:**

`scipy.special.y1()`

**cupyx.scipy.special.i0**

`cupyx.scipy.special.i0` = <ufunc 'cupyx\_scipy\_special\_i0'>

Modified Bessel function of order 0.

**See also:**

`scipy.special.i0()`

**cupyx.scipy.special.i1**

`cupyx.scipy.special.i1` = <ufunc 'cupyx\_scipy\_special\_i1'>

Modified Bessel function of order 1.

**See also:**

`scipy.special.i1()`

**Raw statistical functions**

**See also:**

`cupyx.scipy.stats`

---

*ndtr*

Cumulative distribution function of normal distribution.

---

**cupyx.scipy.special.ndtr**

`cupyx.scipy.special.ndtr = <ufunc 'cupyx_scipy_special_ndtr'>`  
Cumulative distribution function of normal distribution.

**See also:**

`scipy.special.ndtr()`

**Information Theory functions**

<i>entr</i>	Elementwise function for computing entropy.
<i>rel_entr</i>	Elementwise function for computing relative entropy.
<i>kl_div</i>	Elementwise function for computing Kullback-Leibler divergence.
<i>huber</i>	Elementwise function for computing the Huber loss.
<i>pseudo_huber</i>	Elementwise function for computing the Pseudo-Huber loss.

**cupyx.scipy.special.entr**

`cupyx.scipy.special.entr = <ufunc 'cupyx_scipy_special_entr'>`  
Elementwise function for computing entropy.

**See also:**

`scipy.special.entr()`

**cupyx.scipy.special.rel\_entr**

`cupyx.scipy.special.rel_entr = <ufunc 'cupyx_scipy_special_rel_entr'>`  
Elementwise function for computing relative entropy.

**See also:**

`scipy.special.rel_entr()`

**cupyx.scipy.special.kl\_div**

`cupyx.scipy.special.kl_div = <ufunc 'cupyx_scipy_special_kl_div'>`  
Elementwise function for computing Kullback-Leibler divergence.

**See also:**

`scipy.special.kl_div()`

### cupyx.scipy.special.huber

`cupyx.scipy.special.huber` = <ufunc 'cupyx\_scipy\_special\_huber'>  
 Elementwise function for computing the Huber loss.

**See also:**

`scipy.special.huber()`

### cupyx.scipy.special.pseudo\_huber

`cupyx.scipy.special.pseudo_huber` = <ufunc 'cupyx\_scipy\_special\_pseudo\_huber'>  
 Elementwise function for computing the Pseudo-Huber loss.

**See also:**

`scipy.special.pseudo_huber()`

## Gamma and related functions

<i>gamma</i>	Gamma function.
<i>gammaln</i>	Logarithm of the absolute value of the Gamma function.
<i>polygamma</i> (n, x)	Polygamma function n.
<i>digamma</i>	The digamma function.

### cupyx.scipy.special.gamma

`cupyx.scipy.special.gamma` = <ufunc 'cupyx\_scipy\_gamma'>  
 Gamma function.

**Parameters** *z* (`cupy.ndarray`) – The input of gamma function.

**Returns** Computed value of gamma function.

**Return type** *cupy.ndarray*

**See also:**

`scipy.special.gamma`

### cupyx.scipy.special.gammaln

`cupyx.scipy.special.gammaln` = <ufunc 'cupyx\_scipy\_special\_gammaln'>  
 Logarithm of the absolute value of the Gamma function.

**Parameters**

- *x* (`cupy.ndarray`) – Values on the real line at which to compute
- *gammaln.* –

**Returns** Values of *gammaln* at *x*.

**Return type** *cupy.ndarray*

See also:

`scipy.special.gammaln`

### `cupyx.scipy.special.polygamma`

`cupyx.scipy.special.polygamma(n, x)`  
Polygamma function *n*.

**Parameters**

- ***n*** (`cupy.ndarray`) – The order of the derivative of *psi*.
- ***x*** (`cupy.ndarray`) – Where to evaluate the polygamma function.

**Returns** The result.

**Return type** `cupy.ndarray`

See also:

`scipy.special.polygamma`

### `cupyx.scipy.special.digamma`

`cupyx.scipy.special.digamma = <ufunc 'cupyx_scipy_special_digamma'>`  
The digamma function.

**Parameters** ***x*** (`cupy.ndarray`) – The input of digamma function.

**Returns** Computed value of digamma function.

**Return type** `cupy.ndarray`

See also:

`scipy.special.digamma`

## Error function and Fresnel integrals

<code>erf</code>	Error function.
<code>erfc</code>	Complementary error function.
<code>erfcx</code>	Scaled complementary error function.
<code>erfinv</code>	Inverse function of error function.
<code>erfcinv</code>	Inverse function of complementary error function.

### `cupyx.scipy.special.erf`

`cupyx.scipy.special.erf = <ufunc 'cupyx_scipy_special_erf'>`  
Error function.

See also:

`scipy.special.erf()`

### `cupyx.scipy.special.erfc`

`cupyx.scipy.special.erfc` = <ufunc 'cupyx\_scipy\_special\_erfc'>

Complementary error function.

**See also:**

`scipy.special.erfc()`

### `cupyx.scipy.special.erfcx`

`cupyx.scipy.special.erfcx` = <ufunc 'cupyx\_scipy\_special\_erfcx'>

Scaled complementary error function.

**See also:**

`scipy.special.erfcx()`

### `cupyx.scipy.special.erfinv`

`cupyx.scipy.special.erfinv` = <ufunc 'cupyx\_scipy\_special\_erfinv'>

Inverse function of error function.

**See also:**

`scipy.special.erfinv()`

---

**Note:** The behavior close to (and outside) the domain follows that of SciPy v1.4.0+.

---

### `cupyx.scipy.special.erfcinv`

`cupyx.scipy.special.erfcinv` = <ufunc 'cupyx\_scipy\_special\_erfcinv'>

Inverse function of complementary error function.

**See also:**

`scipy.special.erfcinv()`

---

**Note:** The behavior close to (and outside) the domain follows that of SciPy v1.4.0+.

---

## Other special functions

---

`zeta`

Hurwitz zeta function.

---

### cupyx.scipy.special.zeta

`cupyx.scipy.special.zeta = <ufunc 'cupyx_scipy_special_zeta'>`  
Hurwitz zeta function.

#### Parameters

- **x** (`cupy.ndarray`) – Input data, must be real.
- **q** (`cupy.ndarray`) – Input data, must be real.

**Returns** Values of  $\zeta(x, q)$ .

**Return type** `cupy.ndarray`

See also:

`scipy.special.zeta`

## 5.4.7 Signal processing (`cupyx.scipy.signal`)

---

**Hint:** SciPy API Reference: Signal processing (`scipy.signal`)

---

### Convolution

<code>convolve(in1, in2[, mode, method])</code>	Convolve two N-dimensional arrays.
<code>correlate(in1, in2[, mode, method])</code>	Cross-correlate two N-dimensional arrays.
<code>fftconvolve(in1, in2[, mode, axes])</code>	Convolve two N-dimensional arrays using FFT.
<code>oaconvolve(in1, in2[, mode, axes])</code>	Convolve two N-dimensional arrays using the overlap-add method.
<code>convolve2d(in1, in2[, mode, boundary, fillvalue])</code>	Convolve two 2-dimensional arrays.
<code>correlate2d(in1, in2[, mode, boundary, ...])</code>	Cross-correlate two 2-dimensional arrays.
<code>choose_conv_method(in1, in2[, mode])</code>	Find the fastest convolution/correlation method.

### cupyx.scipy.signal.convolve

`cupyx.scipy.signal.convolve(in1, in2, mode='full', method='auto')`  
Convolve two N-dimensional arrays.

Convolve `in1` and `in2`, with the output size determined by the `mode` argument.

#### Parameters

- **in1** (`cupy.ndarray`) – First input.
- **in2** (`cupy.ndarray`) – Second input. Should have the same number of dimensions as `in1`.
- **mode** (`str`) – Indicates the size of the output:
  - `'full'`: output is the full discrete linear convolution (default)
  - `'valid'`: output consists only of those elements that do not rely on the zero-padding. Either `in1` or `in2` must be at least as large as the other in every dimension.
  - `'same'`: - output is the same size as `in1`, centered with respect to the `'full'` output



- **method** (*str*) – Indicates which method to use for the computations:
  - 'direct': The convolution is determined directly from sums, the definition of convolution
  - 'fft': The Fourier Transform is used to perform the convolution by calling `fftconvolve`.
  - 'auto': Automatically choose direct or FFT based on an estimate of which is faster for the arguments (default).

**Returns** the result of convolution.

**Return type** *cupy.ndarray*

**See also:**

*cupyx.scipy.signal.choose\_conv\_method()*

**See also:**

*cupyx.scipy.signal.correlation()*

**See also:**

*cupyx.scipy.signal.fftconvolve()*

**See also:**

*cupyx.scipy.signal.oaconvolve()*

**See also:**

*cupyx.scipy.ndimage.convolve()*

**See also:**

*scipy.signal.convolve()*

---

**Note:** By default, `convolve` and `correlate` use `method='auto'`, which calls `choose_conv_method` to choose the fastest method using pre-computed values. CuPy may not choose the same method to compute the convolution as SciPy does given the same inputs.

---

## cupyx.scipy.signal.correlate

`cupyx.scipy.signal.correlate(in1, in2, mode='full', method='auto')`

Cross-correlate two N-dimensional arrays.

Cross-correlate `in1` and `in2`, with the output size determined by the `mode` argument.

### Parameters

- **in1** (*cupy.ndarray*) – First input.
- **in2** (*cupy.ndarray*) – Second input. Should have the same number of dimensions as `in1`.
- **mode** (*str*) – Indicates the size of the output:
  - 'full': output is the full discrete linear convolution (default)
  - 'valid': output consists only of those elements that do not rely on the zero-padding. Either `in1` or `in2` must be at least as large as the other in every dimension.
  - 'same': - output is the same size as `in1`, centered with respect to the 'full' output

- **method** (*str*) – Indicates which method to use for the computations:
  - 'direct': The convolution is determined directly from sums, the definition of convolution
  - 'fft': The Fourier Transform is used to perform the convolution by calling `fftconvolve`.
  - 'auto': Automatically choose direct or FFT based on an estimate of which is faster for the arguments (default).

**Returns** the result of correlation.

**Return type** *cupy.ndarray*

**See also:**

*cupyx.scipy.signal.choose\_conv\_method()*

**See also:**

*cupyx.scipy.signal.convolve()*

**See also:**

*cupyx.scipy.signal.fftconvolve()*

**See also:**

*cupyx.scipy.signal.oaconvolve()*

**See also:**

*cupyx.scipy.ndimage.correlation()*

**See also:**

*scipy.signal.correlation()*

---

**Note:** By default, `convolve` and `correlate` use `method='auto'`, which calls `choose_conv_method` to choose the fastest method using pre-computed values. CuPy may not choose the same method to compute the convolution as SciPy does given the same inputs.

---

## cupyx.scipy.signal.fftconvolve

`cupyx.scipy.signal.fftconvolve(in1, in2, mode='full', axes=None)`

Convolve two N-dimensional arrays using FFT.

Convolve `in1` and `in2` using the fast Fourier transform method, with the output size determined by the `mode` argument.

This is generally much faster than the 'direct' method of `convolve` for large arrays, but can be slower when only a few output values are needed, and can only output float arrays (int or object array inputs will be cast to float).

### Parameters

- **in1** (*cupy.ndarray*) – First input.
- **in2** (*cupy.ndarray*) – Second input. Should have the same number of dimensions as `in1`.
- **mode** (*str*) – Indicates the size of the output:

- 'full': output is the full discrete linear cross-correlation (default)
- 'valid': output consists only of those elements that do not rely on the zero-padding. Either `in1` or `in2` must be at least as large as the other in every dimension.
- 'same': output is the same size as `in1`, centered with respect to the 'full' output
- **axes** (*scalar or tuple of scalar or `None`*) – Axes over which to compute the convolution. The default is over all axes.

**Returns** the result of convolution

**Return type** `cupy.ndarray`

See also:

`cupyx.scipy.signal.choose_conv_method()`

See also:

`cupyx.scipy.signal.correlation()`

See also:

`cupyx.scipy.signal.convolve()`

See also:

`cupyx.scipy.signal.oaconvolve()`

See also:

`cupyx.scipy.ndimage.convolve()`

See also:

`scipy.signal.correlation()`

## cupyx.scipy.signal.oaconvolve

`cupyx.scipy.signal.oaconvolve(in1, in2, mode='full', axes=None)`

Convolve two N-dimensional arrays using the overlap-add method.

Convolve `in1` and `in2` using the overlap-add method, with the output size determined by the `mode` argument. This is generally faster than `convolve` for large arrays, and generally faster than `fftconvolve` when one array is much larger than the other, but can be slower when only a few output values are needed or when the arrays are very similar in shape, and can only output float arrays (int or object array inputs will be cast to float).

### Parameters

- **in1** (`cupy.ndarray`) – First input.
- **in2** (`cupy.ndarray`) – Second input. Should have the same number of dimensions as `in1`.
- **mode** (`str`) – Indicates the size of the output:
  - 'full': output is the full discrete linear cross-correlation (default)
  - 'valid': output consists only of those elements that do not rely on the zero-padding. Either `in1` or `in2` must be at least as large as the other in every dimension.
  - 'same': output is the same size as `in1`, centered with respect to the 'full' output
- **axes** (*scalar or tuple of scalar or `None`*) – Axes over which to compute the convolution. The default is over all axes.

**Returns** the result of convolution

**Return type** *cupy.ndarray*

See also:

*cupyx.scipy.signal.convolve()*

See also:

*cupyx.scipy.signal.fftconvolve()*

See also:

*cupyx.scipy.ndimage.convolve()*

See also:

*scipy.signal.oaconvolve()*

### **cupyx.scipy.signal.convolve2d**

`cupyx.scipy.signal.convolve2d(in1, in2, mode='full', boundary='fill', fillvalue=0)`

Convolve two 2-dimensional arrays.

Convolve `in1` and `in2` with output size determined by `mode`, and boundary conditions determined by `boundary` and `fillvalue`.

#### **Parameters**

- **in1** (*cupy.ndarray*) – First input.
- **in2** (*cupy.ndarray*) – Second input. Should have the same number of dimensions as `in1`.
- **mode** (*str*) – Indicates the size of the output:
  - 'full': output is the full discrete linear convolution (default)
  - 'valid': output consists only of those elements that do not rely on the zero-padding. Either `in1` or `in2` must be at least as large as the other in every dimension.
  - 'same': - output is the same size as `in1`, centered with respect to the 'full' output
- **boundary** (*str*) – Indicates how to handle boundaries:
  - fill: pad input arrays with `fillvalue` (default)
  - wrap: circular boundary conditions
  - symm: symmetrical boundary conditions
- **fillvalue** (*scalar*) – Value to fill pad input arrays with. Default is 0.

**Returns** A 2-dimensional array containing a subset of the discrete linear convolution of `in1` with `in2`.

**Return type** *cupy.ndarray*

See also:

*cupyx.scipy.signal.convolve()*

See also:

*cupyx.scipy.signal.fftconvolve()*

See also:

`cupyx.scipy.signal.oaconvolve()`

See also:

`cupyx.scipy.signal.correlate2d()`

See also:

`cupyx.scipy.ndimage.convolve()`

See also:

`scipy.signal.convolve2d()`

### `cupyx.scipy.signal.correlate2d`

`cupyx.scipy.signal.correlate2d(in1, in2, mode='full', boundary='fill', fillvalue=0)`

Cross-correlate two 2-dimensional arrays.

Cross correlate `in1` and `in2` with output size determined by `mode`, and boundary conditions determined by `boundary` and `fillvalue`.

#### Parameters

- **in1** (`cupy.ndarray`) – First input.
- **in2** (`cupy.ndarray`) – Second input. Should have the same number of dimensions as `in1`.
- **mode** (`str`) – Indicates the size of the output:
  - 'full': output is the full discrete linear convolution (default)
  - 'valid': output consists only of those elements that do not rely on the zero-padding. Either `in1` or `in2` must be at least as large as the other in every dimension.
  - 'same': - output is the same size as `in1`, centered with respect to the 'full' output
- **boundary** (`str`) – Indicates how to handle boundaries:
  - fill: pad input arrays with `fillvalue` (default)
  - wrap: circular boundary conditions
  - symm: symmetrical boundary conditions
- **fillvalue** (`scalar`) – Value to fill pad input arrays with. Default is 0.

**Returns** A 2-dimensional array containing a subset of the discrete linear cross-correlation of `in1` with `in2`.

**Return type** `cupy.ndarray`

---

**Note:** When using "same" mode with even-length inputs, the outputs of `correlate` and `correlate2d` differ: There is a 1-index offset between them.

---

See also:

`cupyx.scipy.signal.correlate()`

See also:

`cupyx.scipy.signal.convolve2d()`

See also:

`cupyx.scipy.ndimage.correlate()`

See also:

`scipy.signal.correlate2d()`

### `cupyx.scipy.signal.choose_conv_method`

`cupyx.scipy.signal.choose_conv_method(in1, in2, mode='full')`

Find the fastest convolution/correlation method.

#### Parameters

- **in1** (`cupy.ndarray`) – first input.
- **in2** (`cupy.ndarray`) – second input.
- **mode** (`str`, optional) – 'valid', 'same', 'full'.

**Returns** A string indicating which convolution method is fastest, either 'direct' or 'fft'.

**Return type** `str`

**Warning:** This function currently doesn't support measure option, nor multidimensional inputs. It does not guarantee the compatibility of the return value to SciPy's one.

See also:

`scipy.signal.choose_conv_method()`

## Filtering

<code>order_filter(a, domain, rank)</code>	Perform an order filter on an N-D array.
<code>medfilt(volume[, kernel_size])</code>	Perform a median filter on an N-dimensional array.
<code>medfilt2d(input[, kernel_size])</code>	Median filter a 2-dimensional array.
<code>wiener(im[, mysize, noise])</code>	Perform a Wiener filter on an N-dimensional array.

### `cupyx.scipy.signal.order_filter`

`cupyx.scipy.signal.order_filter(a, domain, rank)`

Perform an order filter on an N-D array.

Perform an order filter on the array *in*. The *domain* argument acts as a mask centered over each pixel. The non-zero elements of *domain* are used to select elements surrounding each input pixel which are placed in a list. The list is sorted, and the output for that pixel is the element corresponding to *rank* in the sorted list.

#### Parameters

- **a** (`cupy.ndarray`) – The N-dimensional input array.
- **domain** (`cupy.ndarray`) – A mask array with the same number of dimensions as *a*. Each dimension should have an odd number of elements.

- **rank** (*int*) – A non-negative integer which selects the element from the sorted list (0 corresponds to the smallest element).

**Returns** The results of the order filter in an array with the same shape as *a*.

**Return type** *cupy.ndarray*

See also:

*cupyx.scipy.ndimage.rank\_filter()*

See also:

*scipy.signal.order\_filter()*

## cupyx.scipy.signal.medfilt

`cupyx.scipy.signal.medfilt(volume, kernel_size=None)`

Perform a median filter on an N-dimensional array.

Apply a median filter to the input array using a local window-size given by *kernel\_size*. The array will automatically be zero-padded.

### Parameters

- **volume** (*cupy.ndarray*) – An N-dimensional input array.
- **kernel\_size** (*int or list of ints*) – Gives the size of the median filter window in each dimension. Elements of *kernel\_size* should be odd. If *kernel\_size* is a scalar, then this scalar is used as the size in each dimension. Default size is 3 for each dimension.

**Returns** An array the same size as input containing the median filtered result.

**Return type** *cupy.ndarray*

See also:

*cupyx.scipy.ndimage.median\_filter()*

See also:

*scipy.signal.medfilt()*

## cupyx.scipy.signal.medfilt2d

`cupyx.scipy.signal.medfilt2d(input, kernel_size=3)`

Median filter a 2-dimensional array.

Apply a median filter to the *input* array using a local window-size given by *kernel\_size* (must be odd). The array is zero-padded automatically.

### Parameters

- **input** (*cupy.ndarray*) – A 2-dimensional input array.
- **kernel\_size** (*int or list of ints of length 2*) – Gives the size of the median filter window in each dimension. Elements of *kernel\_size* should be odd. If *kernel\_size* is a scalar, then this scalar is used as the size in each dimension. Default is a kernel of size (3, 3).

**Returns** An array the same size as input containing the median filtered result.

**Return type** *cupy.ndarray*

See also:

, ,

### `cupyx.scipy.signal.wiener`

`cupyx.scipy.signal.wiener(im, mysize=None, noise=None)`

Perform a Wiener filter on an N-dimensional array.

Apply a Wiener filter to the N-dimensional array *im*.

#### Parameters

- **im** (`cupy.ndarray`) – An N-dimensional array.
- **mysize** (`int` or `cupy.ndarray`, *optional*) – A scalar or an N-length list giving the size of the Wiener filter window in each dimension. Elements of *mysize* should be odd. If *mysize* is a scalar, then this scalar is used as the size in each dimension.
- **noise** (`float`, *optional*) – The noise-power to use. If *None*, then noise is estimated as the average of the local variance of the input.

**Returns** Wiener filtered result with the same shape as *im*.

**Return type** `cupy.ndarray`

See also:

`scipy.signal.wiener()`

## 5.4.8 Statistical functions (`cupyx.scipy.stats`)

---

**Hint:** SciPy API Reference: Statistical functions (`scipy.stats`)

---

### Summary statistics

---

<code>entropy(pk[, qk, base, axis])</code>	Calculate the entropy of a distribution for given probability values.
--	---

---

### `cupyx.scipy.stats.entropy`

`cupyx.scipy.stats.entropy(pk, qk=None, base=None, axis=0)`

Calculate the entropy of a distribution for given probability values.

If only probabilities *pk* are given, the entropy is calculated as  $S = -\sum(pk * \log(pk), axis=axis)$ .

If *qk* is not *None*, then compute the Kullback-Leibler divergence  $S = \sum(pk * \log(pk / qk), axis=axis)$ .

This routine will normalize *pk* and *qk* if they don't sum to 1.

#### Parameters

- **pk** (`ndarray`) – Defines the (discrete) distribution. *pk[i]* is the (possibly unnormalized) probability of event *i*.



- **qk** (`ndarray`, *optional*) – Sequence against which the relative entropy is computed. Should be in the same format as `pk`.
- **base** (`float`, *optional*) – The logarithmic base to use, defaults to `e` (natural logarithm).
- **axis** (`int`, *optional*) – The axis along which the entropy is calculated. Default is 0.

**Returns** The calculated entropy.

**Return type** `S` (`cupy.ndarray`)

## 5.5 CuPy-specific functions

CuPy-specific functions are placed under `cupyx` namespace.

<code>cupyx.rsqrt</code>	Returns the reciprocal square root.
<code>cupyx.scatter_add(a, slices, value)</code>	Adds given values to specified elements of an array.
<code>cupyx.scatter_max(a, slices, value)</code>	Stores a maximum value of elements specified by indices to an array.
<code>cupyx.scatter_min(a, slices, value)</code>	Stores a minimum value of elements specified by indices to an array.
<code>cupyx.empty_pinned(shape[, dtype, order])</code>	Returns a new, uninitialized NumPy array with the given shape and dtype.
<code>cupyx.empty_like_pinned(a[, dtype, order, ...])</code>	Returns a new, uninitialized NumPy array with the same shape and dtype as those of the given array.
<code>cupyx.zeros_pinned(shape[, dtype, order])</code>	Returns a new, zero-initialized NumPy array with the given shape and dtype.
<code>cupyx.zeros_like_pinned(a[, dtype, order, ...])</code>	Returns a new, zero-initialized NumPy array with the same shape and dtype as those of the given array.

### 5.5.1 `cupyx.rsqrt`

`cupyx.rsqrt = <ufunc 'cupy_rsqrt'>`  
Returns the reciprocal square root.

### 5.5.2 `cupyx.scatter_add`

`cupyx.scatter_add(a, slices, value)`  
Adds given values to specified elements of an array.

It adds `value` to the specified elements of `a`. If all of the indices target different locations, the operation of `scatter_add()` is equivalent to `a[slices] = a[slices] + value`. If there are multiple elements targeting the same location, `scatter_add()` uses all of these values for addition. On the other hand, `a[slices] = a[slices] + value` only adds the contribution from one of the indices targeting the same location.

Note that just like an array indexing, negative indices are interpreted as counting from the end of an array.

Also note that `scatter_add()` behaves identically to `numpy.add.at()`.

### Example

```
>>> import numpy
>>> import cupy
>>> a = cupy.zeros((6,), dtype=numpy.float32)
>>> i = cupy.array([1, 0, 1])
>>> v = cupy.array([1., 1., 1.])
>>> cupyx.scatter_add(a, i, v);
>>> a
array([1., 2., 0., 0., 0., 0.], dtype=float32)
```

#### Parameters

- **a** (`ndarray`) – An array that gets added.
- **slices** – It is integer, slices, ellipsis, `numpy.newaxis`, integer array-like, boolean array-like or tuple of them. It works for slices used for `cupy.ndarray.__getitem__()` and `cupy.ndarray.__setitem__()`.
- **v** (*array-like*) – Values to increment a at referenced locations.

---

**Note:** It only supports types that are supported by CUDA's `atomicAdd` when an integer array is included in slices. The supported types are `numpy.float32`, `numpy.int32`, `numpy.uint32`, `numpy.uint64` and `numpy.ulonglong`.

---

---

**Note:** `scatter_add()` does not raise an error when indices exceed size of axes. Instead, it wraps indices.

---

See also:

`numpy.ufunc.at()`.

## 5.5.3 cupyx.scatter\_max

`cupyx.scatter_max(a, slices, value)`

Stores a maximum value of elements specified by indices to an array.

It stores the maximum value of elements in `value` array indexed by `slices` to `a`. If all of the indices target different locations, the operation of `scatter_max()` is equivalent to `a[slices] = cupy.maximum(a[slices], value)`. If there are multiple elements targeting the same location, `scatter_max()` stores the maximum of all of these values to the given index of `a`, the initial element of `a` is also taken in account.

Note that just like an array indexing, negative indices are interpreted as counting from the end of an array.

Also note that `scatter_max()` behaves identically to `numpy.maximum.at()`.

### Example

```
>>> import numpy
>>> import cupy
>>> a = cupy.zeros((6,), dtype=numpy.float32)
>>> i = cupy.array([1, 0, 1, 2])
>>> v = cupy.array([1., 2., 3., -1.])
>>> cupyx.scatter_max(a, i, v);
>>> a
array([2., 3., 0., 0., 0., 0.], dtype=float32)
```

#### Parameters

- **a** (`ndarray`) – An array to store the results.
- **slices** – It is integer, slices, ellipsis, `numpy.newaxis`, integer array-like, boolean array-like or tuple of them. It works for slices used for `cupy.ndarray.__getitem__()` and `cupy.ndarray.__setitem__()`.
- **v** (*array-like*) – An array used for reference.

## 5.5.4 cupyx.scatter\_min

`cupyx.scatter_min(a, slices, value)`

Stores a minimum value of elements specified by indices to an array.

It stores the minimum value of elements in `value` array indexed by `slices` to `a`. If all of the indices target different locations, the operation of `scatter_min()` is equivalent to `a[slices] = cupy.minimum(a[slices], value)`. If there are multiple elements targeting the same location, `scatter_min()` stores the minimum of all of these values to the given index of `a`, the initial element of `a` is also taken in account.

Note that just like an array indexing, negative indices are interpreted as counting from the end of an array.

Also note that `scatter_min()` behaves identically to `numpy.minimum.at()`.

### Example

```
>>> import numpy
>>> import cupy
>>> a = cupy.zeros((6,), dtype=numpy.float32)
>>> i = cupy.array([1, 0, 1, 2])
>>> v = cupy.array([1., 2., 3., -1.])
>>> cupyx.scatter_min(a, i, v);
>>> a
array([ 0.,  0., -1.,  0.,  0.,  0.], dtype=float32)
```

#### Parameters

- **a** (`ndarray`) – An array to store the results.
- **slices** – It is integer, slices, ellipsis, `numpy.newaxis`, integer array-like, boolean array-like or tuple of them. It works for slices used for `cupy.ndarray.__getitem__()` and `cupy.ndarray.__setitem__()`.
- **v** (*array-like*) – An array used for reference.

### 5.5.5 cupyx.empty\_pinned

`cupyx.empty_pinned(shape, dtype=<class 'float'>, order='C')`

Returns a new, uninitialized NumPy array with the given shape and dtype.

This is a convenience function which is just `numpy.empty()`, except that the underlying memory is pinned/pagelocked.

**Parameters**

- **shape** (*int or tuple of ints*) – Dimensionalities of the array.
- **dtype** – Data type specifier.
- **order** (`{'C', 'F'}`) – Row-major (C-style) or column-major (Fortran-style) order.

**Returns** A new array with elements not initialized.

**Return type** `numpy.ndarray`

See also:

`numpy.empty()`

### 5.5.6 cupyx.empty\_like\_pinned

`cupyx.empty_like_pinned(a, dtype=None, order='K', subok=None, shape=None)`

Returns a new, uninitialized NumPy array with the same shape and dtype as those of the given array.

This is a convenience function which is just `numpy.empty_like()`, except that the underlying memory is pinned/pagelocked.

This function currently does not support subok option.

**Parameters**

- **a** (`numpy.ndarray` or `cupy.ndarray`) – Base array.
- **dtype** – Data type specifier. The data type of `a` is used by default.
- **order** (`{'C', 'F', 'A', or 'K'}`) – Overrides the memory layout of the result. 'C' means C-order, 'F' means F-order, 'A' means 'F' if `a` is Fortran contiguous, 'C' otherwise. 'K' means match the layout of `a` as closely as possible.
- **subok** – Not supported yet, must be `None`.
- **shape** (*int or tuple of ints*) – Overrides the shape of the result. If `order='K'` and the number of dimensions is unchanged, will try to keep order, otherwise, `order='C'` is implied.

**Returns** A new array with same shape and dtype of `a` with elements not initialized.

**Return type** `numpy.ndarray`

See also:

`numpy.empty_like()`

### 5.5.7 cupyx.zeros\_pinned

`cupyx.zeros_pinned(shape, dtype=<class 'float'>, order='C')`

Returns a new, zero-initialized NumPy array with the given shape and dtype.

This is a convenience function which is just `numpy.zeros()`, except that the underlying memory is pinned/pagelocked.

**Parameters**

- **shape** (*int or tuple of ints*) – Dimensionalities of the array.
- **dtype** – Data type specifier.
- **order** (`{'C', 'F'}`) – Row-major (C-style) or column-major (Fortran-style) order.

**Returns** An array filled with zeros.

**Return type** `numpy.ndarray`

See also:

`numpy.zeros()`

### 5.5.8 cupyx.zeros\_like\_pinned

`cupyx.zeros_like_pinned(a, dtype=None, order='K', subok=None, shape=None)`

Returns a new, zero-initialized NumPy array with the same shape and dtype as those of the given array.

This is a convenience function which is just `numpy.zeros_like()`, except that the underlying memory is pinned/pagelocked.

This function currently does not support subok option.

**Parameters**

- **a** (`numpy.ndarray` or `cupy.ndarray`) – Base array.
- **dtype** – Data type specifier. The dtype of a is used by default.
- **order** (`{'C', 'F', 'A', or 'K'}`) – Overrides the memory layout of the result. 'C' means C-order, 'F' means F-order, 'A' means 'F' if a is Fortran contiguous, 'C' otherwise. 'K' means match the layout of a as closely as possible.
- **subok** – Not supported yet, must be None.
- **shape** (*int or tuple of ints*) – Overrides the shape of the result. If order='K' and the number of dimensions is unchanged, will try to keep order, otherwise, order='C' is implied.

**Returns** An array filled with zeros.

**Return type** `numpy.ndarray`

See also:

`numpy.zeros_like()`

### 5.5.9 DLPack utilities

This is a helper function for creating a `cupy.ndarray` from a DLPack tensor. For further detail see [DLPack](#).

---

<code>cupy.fromDlpack(dltensor)</code>	Zero-copy conversion from a DLPack tensor to a <code>ndarray</code> .
--	---

---

#### `cupy.fromDlpack`

`cupy.fromDlpack(dltensor) → ndarray`

Zero-copy conversion from a DLPack tensor to a `ndarray`.

DLPack is a open in memory tensor structure proposed in this repository: [dmlc/dlpack](#).

This function takes a `PyCapsule` object which contains a pointer to a DLPack tensor as input, and returns a `ndarray`. This function does not copy the data in the DLPack tensor but both DLPack tensor and `ndarray` have pointers which are pointing to the same memory region for the data.

**Parameters** `dltensor` (`PyCapsule`) – Input DLPack tensor which is encapsulated in a `PyCapsule` object.

**Returns** A CuPy `ndarray`.

**Return type** array (`ndarray`)

**See also:**

`cupy.ndarray.toDlpack()` is a method for zero-copy conversion from a `ndarray` to a DLPack tensor (which is encapsulated in a `PyCapsule` object).

**Warning:** As of the DLPack v0.3 specification, it is (implicitly) assumed that the user is responsible to ensure the Producer and the Consumer are operating on the same stream. This requirement might be relaxed/changed in a future DLPack version.

---

#### Example

```
>>> import cupy
>>> array1 = cupy.array([0, 1, 2], dtype=cupy.float32)
>>> dltensor = array1.toDlpack()
>>> array2 = cupy.fromDlpack(dltensor)
>>> cupy.testing.assert_array_equal(array1, array2)
```

### 5.5.10 Automatic Kernel Parameters Optimizations (`cupyx.optimizing`)

---

<code>cupyx.optimizing.optimize(*[, key, path, ...])</code>	Context manager that optimizes kernel launch parameters.
---	--

---

## cupyx.optimizing.optimize

`cupyx.optimizing.optimize(*, key=None, path=None, readonly=False, **config_dict)`

Context manager that optimizes kernel launch parameters.

In this context, CuPy's routines find the best kernel launch parameter values (e.g., the number of threads and blocks). The found values are cached and reused with keys as the shapes, strides and dtypes of the given inputs arrays.

### Parameters

- **key** (*string* or *None*) – The cache key of optimizations.
- **path** (*string* or *None*) – The path to save optimization cache records. When path is specified and exists, records will be loaded from the path. When `readonly` option is set to `False`, optimization cache records will be saved to the path after the optimization.
- **readonly** (*bool*) – See the description of `path` option.
- **max\_trials** (*int*) – The number of trials that defaults to 100.
- **timeout** (*float*) – Stops study after the given number of seconds. Default is 1.
- **max\_total\_time\_per\_trial** (*float*) – Repeats measuring the execution time of the routine for the given number of seconds. Default is 0.1.

### Examples

```
>>> import cupy
>>> from cupyx import optimizing
>>>
>>> x = cupy.arange(100)
>>> with optimizing.optimize():
...     cupy.sum(x)
...
array(4950)
```

---

**Note:** Optuna (<https://optuna.org>) installation is required. Currently it works for reduction operations only.

---

## 5.6 Low-level CUDA support

### 5.6.1 Device management

---

`cupy.cuda.Device([device])`

Object that represents a CUDA device.

## cupy.cuda.Device

**class** `cupy.cuda.Device(device=None)`

Object that represents a CUDA device.

This class provides some basic manipulations on CUDA devices.

It supports the context protocol. For example, the following code is an example of temporarily switching the current device:

```
with Device(0):
    do_something_on_device_0()
```

After the *with* statement gets done, the current device is reset to the original one.

**Parameters** `device` (*int* or `cupy.cuda.Device`) – Index of the device to manipulate. Be careful that the device ID (a.k.a. GPU ID) is zero origin. If it is a Device object, then its ID is used. The current device is selected by default.

**Variables** `id` (*int*) – ID of this device.

## Methods

`__enter__(self)`

`__exit__(self, *args)`

`from_pci_bus_id(type cls, pci_bus_id)`

Returns a new device instance based on a PCI Bus ID

**Parameters** `pci_bus_id` (*str*) – The string for a device in the following format [domain]:[bus]:[device].[function] where domain, bus, device, and function are all hexadecimal values.

**Returns** An instance of the Device class that has the PCI Bus ID as given by the argument `pci_bus_id`.

**Return type** device (*Device*)

`synchronize(self)`

Synchronizes the current thread to the device.

`use(self)`

Makes this device current.

If you want to switch a device temporarily, use the *with* statement.

`__eq__(value, /)`

Return self==value.

`__ne__(value, /)`

Return self!=value.

`__lt__(value, /)`

Return self<value.

`__le__(value, /)`

Return self<=value.

`__gt__(value, /)`

Return self>value.



**\_\_ge\_\_**(*value*, /)  
Return self>=value.

## Attributes

### **attributes**

A dictionary of device attributes.

**Returns** Dictionary of attribute values with the names as keys. The string *cudaDevAttr* has been trimmed from the names. For example, the attribute corresponding to the enumerated value *cudaDevAttrMaxThreadsPerBlock* will have key *MaxThreadsPerBlock*.

**Return type** attributes (*dict*)

### **compute\_capability**

Compute capability of this device.

The capability is represented by a string containing the major index and the minor index. For example, compute capability 3.5 is represented by the string '35'.

### **cublas\_handle**

The cuBLAS handle for this device.

The same handle is used for the same device even if the Device instance itself is different.

### **cusolver\_handle**

The cuSOLVER handle for this device.

The same handle is used for the same device even if the Device instance itself is different.

### **cusolver\_sp\_handle**

The cuSOLVER Sphandle for this device.

The same handle is used for the same device even if the Device instance itself is different.

### **cusparse\_handle**

The cuSPARSE handle for this device.

The same handle is used for the same device even if the Device instance itself is different.

### **id**

'int'

**Type** id

### **mem\_info**

The device memory info.

**Returns** The amount of free memory, in bytes. total: The total amount of memory, in bytes.

**Return type** free

### **pci\_bus\_id**

A string of the PCI Bus ID

**Returns** Returned identifier string for the device in the following format [domain]:[bus]:[device].[function] where domain, bus, device, and function are all hexadecimal values.

**Return type** pci\_bus\_id (*str*)

## 5.6.2 Memory management

<code>cupy.get_default_memory_pool()</code>	Returns CuPy default memory pool for GPU memory.
<code>cupy.get_default_pinned_memory_pool()</code>	Returns CuPy default memory pool for pinned memory.
<code>cupy.cuda.Memory(size_t size)</code>	Memory allocation on a CUDA device.
<code>cupy.cuda.MemoryAsync(size_t size, stream)</code>	Asynchronous memory allocation on a CUDA device.
<code>cupy.cuda.ManagedMemory(size_t size)</code>	Managed memory (Unified memory) allocation on a CUDA device.
<code>cupy.cuda.UnownedMemory(intptr_t ptr, ...)</code>	CUDA memory that is not owned by CuPy.
<code>cupy.cuda.PinnedMemory(size[, flags])</code>	Pinned memory allocation on host.
<code>cupy.cuda.MemoryPointer(BaseMemory mem, ...)</code>	Pointer to a point on a device memory.
<code>cupy.cuda.PinnedMemoryPointer(mem, ...)</code>	Pointer of a pinned memory.
<code>cupy.cuda.malloc_managed(size_t size)</code>	Allocate managed memory (unified memory).
<code>cupy.cuda.malloc_async(size_t size)</code>	(Experimental) Allocate memory from Stream Ordered Memory Allocator.
<code>cupy.cuda.alloc(size)</code>	Calls the current allocator.
<code>cupy.cuda.alloc_pinned_memory(size_t size)</code>	Calls the current allocator.
<code>cupy.cuda.get_allocator()</code>	Returns the current allocator for GPU memory.
<code>cupy.cuda.set_allocator([allocator])</code>	Sets the current allocator for GPU memory.
<code>cupy.cuda.using_allocator([allocator])</code>	Sets a thread-local allocator for GPU memory inside
<code>cupy.cuda.set_pinned_memory_allocator([...])</code>	Sets the current allocator for the pinned memory.
<code>cupy.cuda.MemoryPool([allocator])</code>	Memory pool for all GPU devices on the host.
<code>cupy.cuda.MemoryAsyncPool([pool_handles])</code>	(Experimental) CUDA memory pool for all GPU devices on the host.
<code>cupy.cuda.PinnedMemoryPool([allocator])</code>	Memory pool for pinned memory on the host.
<code>cupy.cuda.PythonFunctionAllocator(...)</code>	Allocator with python functions to perform memory allocation.
<code>cupy.cuda.CFunctionAllocator(intptr_t param, ...)</code>	Allocator with C function pointers to allocation routines.

### cupy.get\_default\_memory\_pool

`cupy.get_default_memory_pool()`

Returns CuPy default memory pool for GPU memory.

**Returns** The memory pool object.

**Return type** `cupy.cuda.MemoryPool`

---

**Note:** If you want to disable memory pool, please use the following code.

```
>>> cupy.cuda.set_allocator(None)
```

---

## cupy.get\_default\_pinned\_memory\_pool

`cupy.get_default_pinned_memory_pool()`

Returns CuPy default memory pool for pinned memory.

**Returns** The memory pool object.

**Return type** *cupy.cuda.PinnedMemoryPool*

---

**Note:** If you want to disable memory pool, please use the following code.

```
>>> cupy.cuda.set_pinned_memory_allocator(None)
```

---

## cupy.cuda.Memory

**class** `cupy.cuda.Memory`(*size\_t* size)

Memory allocation on a CUDA device.

This class provides an RAII interface of the CUDA memory allocation.

**Parameters** **size** (*int*) – Size of the memory allocation in bytes.

### Methods

`__eq__(value, /)`

Return self==value.

`__ne__(value, /)`

Return self!=value.

`__lt__(value, /)`

Return self<value.

`__le__(value, /)`

Return self<=value.

`__gt__(value, /)`

Return self>value.

`__ge__(value, /)`

Return self>=value.

### Attributes

**device**

**device\_id**

‘int’

**Type** device\_id

**ptr**

‘intptr\_t’

**Type** ptr

**size**  
    'size\_t'  
    **Type** size

## cupy.cuda.MemoryAsync

**class** cupy.cuda.**MemoryAsync**(*size\_t size, stream*)  
Asynchronous memory allocation on a CUDA device.

This class provides an RAII interface of the CUDA memory allocation.

### Parameters

- **size** (*int*) – Size of the memory allocation in bytes.
- **stream** (*Stream*) – The stream on which the memory is allocated and freed.

### Methods

**\_\_eq\_\_**(*value, /*)  
Return self==value.

**\_\_ne\_\_**(*value, /*)  
Return self!=value.

**\_\_lt\_\_**(*value, /*)  
Return self<value.

**\_\_le\_\_**(*value, /*)  
Return self<=value.

**\_\_gt\_\_**(*value, /*)  
Return self>value.

**\_\_ge\_\_**(*value, /*)  
Return self>=value.

### Attributes

**device**

**device\_id**  
    'int'  
    **Type** device\_id

**ptr**  
    'intptr\_t'  
    **Type** ptr

**size**  
    'size\_t'  
    **Type** size

**stream\_ref**

## cupy.cuda.ManagedMemory

**class** cupy.cuda.**ManagedMemory**(*size\_t* size)

Managed memory (Unified memory) allocation on a CUDA device.

This class provides an RAII interface of the CUDA managed memory allocation.

**Parameters** **size** (*int*) – Size of the memory allocation in bytes.

### Methods

**advise**(*self*, *int* advise, *Device* dev)

(experimental) Advise about the usage of this memory.

**Parameters**

- **advics** (*int*) – Advise to be applied for this memory.
- **dev** (cupy.cuda.*Device*) – Device to apply the advice for.

**prefetch**(*self*, *stream*)

(experimental) Prefetch memory.

**Parameters** **stream** (cupy.cuda.*Stream*) – CUDA stream.

**\_\_eq\_\_**(*value*, /)

Return self==value.

**\_\_ne\_\_**(*value*, /)

Return self!=value.

**\_\_lt\_\_**(*value*, /)

Return self<value.

**\_\_le\_\_**(*value*, /)

Return self<=value.

**\_\_gt\_\_**(*value*, /)

Return self>value.

**\_\_ge\_\_**(*value*, /)

Return self>=value.

### Attributes

**device**

**device\_id**

‘int’

**Type** device\_id

**ptr**

‘intptr\_t’

**Type** ptr

**size**

‘size\_t’

**Type** size

## cupy.cuda.UnownedMemory

**class** `cupy.cuda.UnownedMemory(intptr_t ptr, size_t size, owner, int device_id=-1)`  
CUDA memory that is not owned by CuPy.

### Parameters

- **ptr** (*int*) – Pointer to the buffer.
- **size** (*int*) – Size of the buffer.
- **owner** (*object*) – Reference to the owner object to keep the memory alive.
- **device\_id** (*int*) – CUDA device ID of the buffer. If omitted, the device associated to the pointer is retrieved.

### Methods

`__eq__(value, /)`  
Return self==value.

`__ne__(value, /)`  
Return self!=value.

`__lt__(value, /)`  
Return self<value.

`__le__(value, /)`  
Return self<=value.

`__gt__(value, /)`  
Return self>value.

`__ge__(value, /)`  
Return self>=value.

### Attributes

**device**

**device\_id**  
‘int’  
**Type** device\_id

**ptr**  
‘intptr\_t’  
**Type** ptr

**size**  
‘size\_t’  
**Type** size

### cupy.cuda.PinnedMemory

**class** `cupy.cuda.PinnedMemory`(*size*, *flags*=0)

Pinned memory allocation on host.

This class provides a RAII interface of the pinned memory allocation.

**Parameters** **size** (*int*) – Size of the memory allocation in bytes.

#### Methods

`__eq__(value, /)`

Return self==value.

`__ne__(value, /)`

Return self!=value.

`__lt__(value, /)`

Return self<value.

`__le__(value, /)`

Return self<=value.

`__gt__(value, /)`

Return self>value.

`__ge__(value, /)`

Return self>=value.

### cupy.cuda.MemoryPointer

**class** `cupy.cuda.MemoryPointer`(*BaseMemory mem*, *ptrdiff\_t offset*)

Pointer to a point on a device memory.

An instance of this class holds a reference to the original memory buffer and a pointer to a place within this buffer.

#### Parameters

- **mem** (*BaseMemory*) – The device memory buffer.
- **offset** (*int*) – An offset from the head of the buffer to the place this pointer refers.

#### Variables

- **device** (*Device*) – Device whose memory the pointer refers to.
- **mem** (*BaseMemory*) – The device memory buffer.
- **ptr** (*int*) – Pointer to the place within the buffer.

## Methods

**copy\_from**(*self*, *mem*, *size\_t* size)

Copies a memory sequence from a (possibly different) device or host.

This function is a useful interface that selects appropriate one from `copy_from_device()` and `copy_from_host()`.

### Parameters

- **mem** (*int* or `ctypes.c_void_p` or `cupy.cuda.MemoryPointer`) – Source memory pointer.
- **size** (*int*) – Size of the sequence in bytes.

**Warning:** This function always uses the legacy default stream and does not honor the current stream. Use `copy_from_async` instead if you are using streams in your code, or have PTDS enabled.

**copy\_from\_async**(*self*, *mem*, *size\_t* size, *stream=None*)

Copies a memory sequence from an arbitrary place asynchronously.

This function is a useful interface that selects appropriate one from `copy_from_device_async()` and `copy_from_host_async()`.

### Parameters

- **mem** (*int* or `ctypes.c_void_p` or `cupy.cuda.MemoryPointer`) – Source memory pointer.
- **size** (*int*) – Size of the sequence in bytes.
- **stream** (`cupy.cuda.Stream`) – CUDA stream. The default uses CUDA stream of the current context.

**copy\_from\_device**(*self*, *MemoryPointer* src, *size\_t* size)

Copies a memory sequence from a (possibly different) device.

### Parameters

- **src** (`cupy.cuda.MemoryPointer`) – Source memory pointer.
- **size** (*int*) – Size of the sequence in bytes.

**Warning:** This function always uses the legacy default stream and does not honor the current stream. Use `copy_from_device_async` instead if you are using streams in your code, or have PTDS enabled.

**copy\_from\_device\_async**(*self*, *MemoryPointer* src, *size\_t* size, *stream=None*)

Copies a memory from a (possibly different) device asynchronously.

### Parameters

- **src** (`cupy.cuda.MemoryPointer`) – Source memory pointer.
- **size** (*int*) – Size of the sequence in bytes.
- **stream** (`cupy.cuda.Stream`) – CUDA stream. The default uses CUDA stream of the current context.

**copy\_from\_host**(*self*, *mem*, *size\_t* size)

Copies a memory sequence from the host memory.



**Parameters**

- **mem** (*int* or *ctypes.c\_void\_p*) – Source memory pointer.
- **size** (*int*) – Size of the sequence in bytes.

**Warning:** This function always uses the legacy default stream and does not honor the current stream. Use `copy_from_host_async` instead if you are using streams in your code, or have PTDS enabled.

**copy\_from\_host\_async**(*self*, *mem*, *size\_t* *size*, *stream=None*)

Copies a memory sequence from the host memory asynchronously.

**Parameters**

- **mem** (*int* or *ctypes.c\_void\_p*) – Source memory pointer. It must point to pinned memory.
- **size** (*int*) – Size of the sequence in bytes.
- **stream** (*cupy.cuda.Stream*) – CUDA stream. The default uses CUDA stream of the current context.

**copy\_to\_host**(*self*, *mem*, *size\_t* *size*)

Copies a memory sequence to the host memory.

**Parameters**

- **mem** (*int* or *ctypes.c\_void\_p*) – Target memory pointer.
- **size** (*int*) – Size of the sequence in bytes.

**Warning:** This function always uses the legacy default stream and does not honor the current stream. Use `copy_to_host_async` instead if you are using streams in your code, or have PTDS enabled.

**copy\_to\_host\_async**(*self*, *mem*, *size\_t* *size*, *stream=None*)

Copies a memory sequence to the host memory asynchronously.

**Parameters**

- **mem** (*int* or *ctypes.c\_void\_p*) – Target memory pointer. It must point to pinned memory.
- **size** (*int*) – Size of the sequence in bytes.
- **stream** (*cupy.cuda.Stream*) – CUDA stream. The default uses CUDA stream of the current context.

**memset**(*self*, *int* *value*, *size\_t* *size*)

Fills a memory sequence by constant byte value.

**Parameters**

- **value** (*int*) – Value to fill.
- **size** (*int*) – Size of the sequence in bytes.

**Warning:** This function always uses the legacy default stream and does not honor the current stream. Use `memset_async` instead if you are using streams in your code, or have PTDS enabled.

**memset\_async**(*self*, *int* value, *size\_t* size, *stream=None*)

Fills a memory sequence by constant byte value asynchronously.

#### Parameters

- **value** (*int*) – Value to fill.
- **size** (*int*) – Size of the sequence in bytes.
- **stream** (`cupy.cuda.Stream`) – CUDA stream. The default uses CUDA stream of the current context.

**\_\_eq\_\_**(*value*, /)

Return self==value.

**\_\_ne\_\_**(*value*, /)

Return self!=value.

**\_\_lt\_\_**(*value*, /)

Return self<value.

**\_\_le\_\_**(*value*, /)

Return self<=value.

**\_\_gt\_\_**(*value*, /)

Return self>value.

**\_\_ge\_\_**(*value*, /)

Return self>=value.

#### Attributes

**device**

**device\_id**

**mem**

**ptr**

### `cupy.cuda.PinnedMemoryPointer`

**class** `cupy.cuda.PinnedMemoryPointer`(*mem*, *ptrdiff\_t* offset)

Pointer of a pinned memory.

An instance of this class holds a reference to the original memory buffer and a pointer to a place within this buffer.

#### Parameters

- **mem** (`PinnedMemory`) – The device memory buffer.
- **offset** (*int*) – An offset from the head of the buffer to the place this pointer refers.

#### Variables

- **mem** (`PinnedMemory`) – The device memory buffer.
- **ptr** (*int*) – Pointer to the place within the buffer.

## Methods

**size**(*self*) → *size\_t*

**\_\_eq\_\_**(*value*, /)  
Return self==value.

**\_\_ne\_\_**(*value*, /)  
Return self!=value.

**\_\_lt\_\_**(*value*, /)  
Return self<value.

**\_\_le\_\_**(*value*, /)  
Return self<=value.

**\_\_gt\_\_**(*value*, /)  
Return self>value.

**\_\_ge\_\_**(*value*, /)  
Return self>=value.

## Attributes

**mem**

**ptr**

## cupy.cuda.malloc\_managed

**cupy.cuda.malloc\_managed**(*size\_t size*) → *MemoryPointer*  
Allocate managed memory (unified memory).

This method can be used as a CuPy memory allocator. The simplest way to use a managed memory as the default allocator is the following code:

```
set_allocator(malloc_managed)
```

The advantage using managed memory in CuPy is that device memory oversubscription is possible for GPUs that have a non-zero value for the device attribute `cudaDevAttrConcurrentManagedAccess`. CUDA >= 8.0 with GPUs later than or equal to Pascal is preferable.

Read more at: [https://docs.nvidia.com/cuda/cuda-runtime-api/group\\_\\_CUDART\\_\\_MEMORY.html#axzz4qygc1Ry1](https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html#axzz4qygc1Ry1) # NOQA

**Parameters** **size** (*int*) – Size of the memory allocation in bytes.

**Returns** Pointer to the allocated buffer.

**Return type** *MemoryPointer*

### cupy.cuda.malloc\_async

cupy.cuda.malloc\_async(size\_t size) → *MemoryPointer*

(Experimental) Allocate memory from Stream Ordered Memory Allocator.

This method can be used as a CuPy memory allocator. The simplest way to use CUDA's Stream Ordered Memory Allocator as the default allocator is the following code:

```
set_allocator(malloc_async)
```

Using this feature requires CUDA >= 11.2 with a supported GPU and platform. If it is not supported, an error will be raised.

The current CuPy stream is used to allocate/free the memory.

**Parameters** **size** (*int*) – Size of the memory allocation in bytes.

**Returns** Pointer to the allocated buffer.

**Return type** *MemoryPointer*

**Warning:** This feature is currently experimental and subject to change.

See also:

[Stream Ordered Memory Allocator](#)

### cupy.cuda.alloc

cupy.cuda.alloc(size) → *MemoryPointer*

Calls the current allocator.

Use [set\\_allocator\(\)](#) to change the current allocator.

**Parameters** **size** (*int*) – Size of the memory allocation.

**Returns** Pointer to the allocated buffer.

**Return type** *MemoryPointer*

### cupy.cuda.alloc\_pinned\_memory

cupy.cuda.alloc\_pinned\_memory(size\_t size) → *PinnedMemoryPointer*

Calls the current allocator.

Use [set\\_pinned\\_memory\\_allocator\(\)](#) to change the current allocator.

**Parameters** **size** (*int*) – Size of the memory allocation.

**Returns** Pointer to the allocated buffer.

**Return type** *PinnedMemoryPointer*

### cupy.cuda.get\_allocator

`cupy.cuda.get_allocator()`

Returns the current allocator for GPU memory.

**Returns** CuPy memory allocator.

**Return type** function

### cupy.cuda.set\_allocator

`cupy.cuda.set_allocator(allocator=None)`

Sets the current allocator for GPU memory.

**Parameters** `allocator` (*function*) – CuPy memory allocator. It must have the same interface as the `cupy.cuda.alloc()` function, which takes the buffer size as an argument and returns the device buffer of that size. When `None` is specified, raw memory allocator will be used (i.e., memory pool is disabled).

### cupy.cuda.using\_allocator

`cupy.cuda.using_allocator(allocator=None)`

Sets a thread-local allocator for GPU memory inside context manager

**Parameters** `allocator` (*function*) – CuPy memory allocator. It must have the same interface as the `cupy.cuda.alloc()` function, which takes the buffer size as an argument and returns the device buffer of that size. When `None` is specified, raw memory allocator will be used (i.e., memory pool is disabled).

### cupy.cuda.set\_pinned\_memory\_allocator

`cupy.cuda.set_pinned_memory_allocator(allocator=None)`

Sets the current allocator for the pinned memory.

**Parameters** `allocator` (*function*) – CuPy pinned memory allocator. It must have the same interface as the `cupy.cuda.alloc_pinned_memory()` function, which takes the buffer size as an argument and returns the device buffer of that size. When `None` is specified, raw memory allocator is used (i.e., memory pool is disabled).

### cupy.cuda.MemoryPool

**class** `cupy.cuda.MemoryPool(allocator=None)`

Memory pool for all GPU devices on the host.

A memory pool preserves any allocations even if they are freed by the user. Freed memory buffers are held by the memory pool as *free blocks*, and they are reused for further memory allocations of the same sizes. The allocated blocks are managed for each device, so one instance of this class can be used for multiple devices.

---

**Note:** When the allocation is skipped by reusing the pre-allocated block, it does not call `cudaMalloc` and therefore CPU-GPU synchronization does not occur. It makes interleaves of memory allocations and kernel invocations very fast.

---

---

**Note:** The memory pool holds allocated blocks without freeing as much as possible. It makes the program hold most of the device memory, which may make other CUDA programs running in parallel out-of-memory situation.

---

**Parameters** **allocator** (*function*) – The base CuPy memory allocator. It is used for allocating new blocks when the blocks of the required size are all in use.

## Methods

**free\_all\_blocks**(*self*, *stream=None*)

Releases free blocks.

**Parameters** **stream** (`cupy.cuda.Stream`) – Release free blocks in the arena of the given stream. The default releases blocks in all arenas.

---

**Note:** A memory pool may split a free block for space efficiency. A split block is not released until all its parts are merged back into one even if `free_all_blocks()` is called.

---

**free\_all\_free**(*self*)

(Deprecated) Use `free_all_blocks()` instead.

**free\_bytes**(*self*) → `size_t`

Gets the total number of bytes acquired but not used in the pool.

**Returns** The total number of bytes acquired but not used in the pool.

**Return type** `int`

**get\_limit**(*self*) → `size_t`

Gets the upper limit of memory allocation of the current device.

**Returns** The number of bytes

**Return type** `int`

**malloc**(*self*, `size_t size`) → `MemoryPointer`

Allocates the memory, from the pool if possible.

This method can be used as a CuPy memory allocator. The simplest way to use a memory pool as the default allocator is the following code:

```
set_allocator(MemoryPool().malloc)
```

Also, the way to use a memory pool of Managed memory (Unified memory) as the default allocator is the following code:

```
set_allocator(MemoryPool(malloc_managed).malloc)
```

**Parameters** **size** (`int`) – Size of the memory buffer to allocate in bytes.

**Returns** Pointer to the allocated buffer.

**Return type** `MemoryPointer`

**n\_free\_blocks**(*self*) → `size_t`

Counts the total number of free blocks.

**Returns** The total number of free blocks.

**Return type** `int`

**set\_limit**(*self*, *size=None*, *fraction=None*)

Sets the upper limit of memory allocation of the current device.

When *fraction* is specified, its value will become a fraction of the amount of GPU memory that is available for allocation. For example, if you have a GPU with 2 GiB memory, you can either use `set_limit(fraction=0.5)` or `set_limit(size=1024**3)` to limit the memory size to 1 GiB.

*size* and *fraction* cannot be specified at one time. If both of them are **not** specified or `0` is specified, the limit will be disabled.

---

**Note:** You can also set the limit by using `CUPY_GPU_MEMORY_LIMIT` environment variable. See [Environment variables](#) for the details. The limit set by this method supersedes the value specified in the environment variable.

Also note that this method only changes the limit for the current device, whereas the environment variable sets the default limit for all devices.

---

#### Parameters

- **size** (`int`) – Limit size in bytes.
- **fraction** (`float`) – Fraction in the range of `[0, 1]`.

**total\_bytes**(*self*) → `size_t`

Gets the total number of bytes acquired in the pool.

**Returns** The total number of bytes acquired in the pool.

**Return type** `int`

**used\_bytes**(*self*) → `size_t`

Gets the total number of bytes used.

**Returns** The total number of bytes used.

**Return type** `int`

**\_\_eq\_\_**(*value*, /)

Return `self==value`.

**\_\_ne\_\_**(*value*, /)

Return `self!=value`.

**\_\_lt\_\_**(*value*, /)

Return `self<value`.

**\_\_le\_\_**(*value*, /)

Return `self<=value`.

**\_\_gt\_\_**(*value*, /)

Return `self>value`.

**\_\_ge\_\_**(*value*, /)

Return `self>=value`.

## cupy.cuda.MemoryAsyncPool

**class** `cupy.cuda.MemoryAsyncPool(pool_handles='current')`  
(Experimental) CUDA memory pool for all GPU devices on the host.

A memory pool preserves any allocations even if they are freed by the user. One instance of this class can be used for multiple devices. This class uses CUDA's Stream Ordered Memory Allocator (supported on CUDA 11.2+). The simplest way to use this pool as CuPy's default allocator is the following code:

```
set_allocator(MemoryAsyncPool().malloc)
```

Using this feature requires CUDA  $\geq 11.2$  with a supported GPU and platform. If it is not supported, an error will be raised.

The current CuPy stream is used to allocate/free the memory.

**Parameters** `pool_handles` (*str* or *int*) – A flag to indicate which mempool to use. *'default'* is for the device's default mempool, *'current'* is for the current mempool (which could be the default one), and an *int* that represents `cudaMemPool_t` created from elsewhere for an external mempool. A list consisting of these flags can also be accepted, in which case the list length must equal to the total number of visible devices so that the mempools for each device can be set independently.

**Warning:** This feature is currently experimental and subject to change.

---

**Note:** `MemoryAsyncPool` currently cannot work with memory hooks.

---

See also:

[Stream Ordered Memory Allocator](#)

## Methods

**free\_all\_blocks**(*self*, *stream=None*)

**free\_bytes**(*self*)  $\rightarrow$  `size_t`

**get\_limit**(*self*)  $\rightarrow$  `size_t`

**malloc**(*self*, *size\_t* *size*)  $\rightarrow$  `MemoryPointer`

Allocate memory from the current device's pool on the current stream.

This method can be used as a CuPy memory allocator. The simplest way to use a memory pool as the default allocator is the following code:

```
set_allocator(MemoryAsyncPool().malloc)
```

**Parameters** `size` (*int*) – Size of the memory buffer to allocate in bytes.

**Returns** Pointer to the allocated buffer.

**Return type** `MemoryPointer`

**n\_free\_blocks**(*self*)  $\rightarrow$  `size_t`

**set\_limit**(*self*, *size=None*, *fraction=None*)



**total\_bytes**(*self*) → size\_t

**used\_bytes**(*self*) → size\_t

**\_\_eq\_\_**(*value*, /)  
Return self==value.

**\_\_ne\_\_**(*value*, /)  
Return self!=value.

**\_\_lt\_\_**(*value*, /)  
Return self<value.

**\_\_le\_\_**(*value*, /)  
Return self<=value.

**\_\_gt\_\_**(*value*, /)  
Return self>value.

**\_\_ge\_\_**(*value*, /)  
Return self>=value.

### cupy.cuda.PinnedMemoryPool

**class** cupy.cuda.PinnedMemoryPool(*allocator*=\_malloc)

Memory pool for pinned memory on the host.

Note that it preserves all allocated memory buffers even if the user explicitly release the one. Those released memory buffers are held by the memory pool as *free blocks*, and reused for further memory allocations of the same size.

**Parameters** **allocator** (*function*) – The base CuPy pinned memory allocator. It is used for allocating new blocks when the blocks of the required size are all in use.

### Methods

**free**(*self*, *intptr\_t* ptr, *size\_t* size)

**free\_all\_blocks**(*self*)  
Release free all blocks.

**malloc**(*self*, *size\_t* size) → *PinnedMemoryPointer*

**n\_free\_blocks**(*self*)  
Count the total number of free blocks.

**Returns** The total number of free blocks.

**Return type** *int*

**\_\_eq\_\_**(*value*, /)  
Return self==value.

**\_\_ne\_\_**(*value*, /)  
Return self!=value.

**\_\_lt\_\_**(*value*, /)  
Return self<value.

**\_\_le\_\_**(*value*, /)  
Return self<=value.

`__gt__(value, /)`  
Return self>value.

`__ge__(value, /)`  
Return self>=value.

## cupy.cuda.PythonFunctionAllocator

**class** `cupy.cuda.PythonFunctionAllocator`(*malloc\_func*, *free\_func*)

Allocator with python functions to perform memory allocation.

This allocator keeps functions corresponding to *malloc* and *free*, delegating the actual allocation to external sources while only handling the timing of the resource allocation and deallocation.

*malloc* should follow the signature `malloc(int, int) -> int` returning the pointer to the allocated memory given the *param* object, the number of bytes to allocate and the device id on which the allocation should take place.

Similarly, *free* should follow the signature `free(int, int)` with no return, taking the pointer to the allocated memory and the device id on which the memory was allocated.

If the external memory management supports asynchronous operations, the current CuPy stream can be retrieved inside *malloc\_func* and *free\_func* by calling `cupy.cuda.get_current_stream()`. To use external streams, wrap them with `cupy.cuda.ExternalStream()`.

### Parameters

- **malloc\_func** (*function*) – *malloc* function to be called.
- **free\_func** (*function*) – *free* function to be called.

### Methods

**malloc**(*self*, *size\_t* *size*) → *MemoryPointer*

`__eq__(value, /)`  
Return self==value.

`__ne__(value, /)`  
Return self!=value.

`__lt__(value, /)`  
Return self<value.

`__le__(value, /)`  
Return self<=value.

`__gt__(value, /)`  
Return self>value.

`__ge__(value, /)`  
Return self>=value.

## cupy.cuda.CFunctionAllocator

**class** `cupy.cuda.CFunctionAllocator(intptr_t param, intptr_t malloc_func, intptr_t free_func, owner)`  
 Allocator with C function pointers to allocation routines.

This allocator keeps raw pointers to a *param* object along with functions pointers to *malloc* and *free*, delegating the actual allocation to external sources while only handling the timing of the resource allocation and deallocation.

*malloc* should follow the signature `void*(*malloc)(void*, size_t, int)` returning the pointer to the allocated memory given the pointer to *param*, the number of bytes to allocate and the device id on which the allocation should take place.

Similarly, *free* should follow the signature `void(*free)(void*, void*, int)` with no return, taking the pointer to *param*, the pointer to the allocated memory and the device id on which the memory was allocated.

### Parameters

- **param** (*int*) – Address of *param*.
- **malloc\_func** (*int*) – Address of *malloc*.
- **free\_func** (*int*) – Address of *free*.
- **owner** (*object*) – Reference to the owner object to keep the param and the functions alive.

### Methods

**malloc**(*self*, *size\_t* size) → *MemoryPointer*

**\_\_eq\_\_**(*value*, /)  
 Return self==value.

**\_\_ne\_\_**(*value*, /)  
 Return self!=value.

**\_\_lt\_\_**(*value*, /)  
 Return self<value.

**\_\_le\_\_**(*value*, /)  
 Return self<=value.

**\_\_gt\_\_**(*value*, /)  
 Return self>value.

**\_\_ge\_\_**(*value*, /)  
 Return self>=value.

## 5.6.3 Memory hook

<code>cupy.cuda.MemoryHook()</code>	Base class of hooks for Memory allocations.
<code>cupy.cuda.memory_hooks.DebugPrintHook([...])</code>	Memory hook that prints debug information.
<code>cupy.cuda.memory_hooks.LineProfileHook([...])</code>	Code line CuPy memory profiler.

## cupy.cuda.MemoryHook

### class cupy.cuda.MemoryHook

Base class of hooks for Memory allocations.

*MemoryHook* is an callback object. Registered memory hooks are invoked before and after memory is allocated from GPU device, and memory is retrieved from memory pool, and memory is released to memory pool.

Memory hooks that derive *MemoryHook* are required to implement six methods: *alloc\_preprocess()*, *alloc\_postprocess()*, *malloc\_preprocess()*, *malloc\_postprocess()*, *free\_preprocess()*, and *free\_postprocess()*. By default, these methods do nothing.

Specifically, *alloc\_preprocess()* (resp. *alloc\_postprocess()*) of all memory hooks registered are called before (resp. after) memory is allocated from GPU device.

Likewise, *malloc\_preprocess()* (resp. *malloc\_postprocess()*) of all memory hooks registered are called before (resp. after) memory is retrieved from memory pool.

Below is a pseudo code to describe how malloc and hooks work. Please note that *alloc\_preprocess()* and *alloc\_postprocess()* are not invoked if a cached free chunk is found:

```
def malloc(size):
    Call malloc_preprocess of all memory hooks
    Try to find a cached free chunk from memory pool
    if chunk is not found:
        Call alloc_preprocess for all memory hooks
        Invoke actual memory allocation to get a new chunk
        Call alloc_postprocess for all memory hooks
    Call malloc_postprocess for all memory hooks
```

Moreover, *free\_preprocess()* (resp. *free\_postprocess()*) of all memory hooks registered are called before (resp. after) memory is released to memory pool.

Below is a pseudo code to describe how free and hooks work:

```
def free(ptr):
    Call free_preprocess of all memory hooks
    Push a memory chunk of a given pointer back to memory pool
    Call free_postprocess for all memory hooks
```

To register a memory hook, use with statement. Memory hooks are registered to all method calls within with statement and are unregistered at the end of with statement.

---

**Note:** CuPy stores the dictionary of registered function hooks as a thread local object. So, memory hooks registered can be different depending on threads.

---

## Methods

**\_\_enter\_\_**(*self*)

**\_\_exit\_\_**(*self*, \*\_)

**alloc\_postprocess**(*self*, \*\**kwargs*)

Callback function invoked after allocating memory from GPU device.

### Keyword Arguments

- **device\_id** (*int*) – CUDA device ID
- **mem\_size** (*int*) – Rounded memory bytesize allocated
- **mem\_ptr** (*int*) – Obtained memory pointer. 0 if an error occurred in allocation.

**alloc\_preprocess**(*self*, \*\**kwargs*)

Callback function invoked before allocating memory from GPU device.

### Keyword Arguments

- **device\_id** (*int*) – CUDA device ID
- **mem\_size** (*int*) – Rounded memory bytesize to be allocated

**free\_postprocess**(*self*, \*\**kwargs*)

Callback function invoked after releasing memory to memory pool.

### Keyword Arguments

- **device\_id** (*int*) – CUDA device ID
- **mem\_size** (*int*) – Memory bytesize
- **mem\_ptr** (*int*) – Memory pointer to free
- **pmem\_id** (*int*) – Pooled memory object ID.

**free\_preprocess**(*self*, \*\**kwargs*)

Callback function invoked before releasing memory to memory pool.

### Keyword Arguments

- **device\_id** (*int*) – CUDA device ID
- **mem\_size** (*int*) – Memory bytesize
- **mem\_ptr** (*int*) – Memory pointer to free
- **pmem\_id** (*int*) – Pooled memory object ID.

**malloc\_postprocess**(*self*, \*\**kwargs*)

Callback function invoked after retrieving memory from memory pool.

### Keyword Arguments

- **device\_id** (*int*) – CUDA device ID
- **size** (*int*) – Requested memory bytesize to allocate
- **mem\_size** (*int*) – Rounded memory bytesize allocated
- **mem\_ptr** (*int*) – Obtained memory pointer. 0 if an error occurred in malloc.
- **pmem\_id** (*int*) – Pooled memory object ID. 0 if an error occurred in malloc.

**malloc\_preprocess**(*self*, \*\**kwargs*)

Callback function invoked before retrieving memory from memory pool.

### Keyword Arguments

- **device\_id** (*int*) – CUDA device ID
- **size** (*int*) – Requested memory bytesize to allocate
- **mem\_size** (*int*) – Rounded memory bytesize to be allocated

**\_\_eq\_\_**(*value*, /)  
Return self==value.

**\_\_ne\_\_**(*value*, /)  
Return self!=value.

**\_\_lt\_\_**(*value*, /)  
Return self<value.

**\_\_le\_\_**(*value*, /)  
Return self<=value.

**\_\_gt\_\_**(*value*, /)  
Return self>value.

**\_\_ge\_\_**(*value*, /)  
Return self>=value.

### Attributes

**name** = 'MemoryHook'

### cupy.cuda.memory\_hooks.DebugPrintHook

**class** cupy.cuda.memory\_hooks.DebugPrintHook(*file=<\_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>, flush=True*)

Memory hook that prints debug information.

This memory hook outputs the debug information of input arguments of `malloc` and `free` methods involved in the hooked functions at postprocessing time (that is, just after each method is called).

### Example

The basic usage is to use it with `with` statement.

Code example:

```
>>> import cupy
>>> from cupy.cuda import memory_hooks
>>>
>>> cupy.cuda.set_allocator(cupy.cuda.MemoryPool().malloc)
>>> with memory_hooks.DebugPrintHook():
...     x = cupy.array([1, 2, 3])
...     del x
```

Output example:

```
{
  "hook": "alloc", "device_id": 0, "mem_size": 512, "mem_ptr": 150496608256
}
{
  "hook": "malloc", "device_id": 0, "size": 24, "mem_size": 512, "mem_ptr": 150496608256,
  "pmem_id": "0x7f39200c5278"
}
{
  "hook": "free", "device_id": 0, "mem_size": 512, "mem_ptr": 150496608256, "pmem_id":
  "0x7f39200c5278"
}
```

where the output format is JSONL (JSON Lines) and `hook` is the name of hook point, and `device_id` is the CUDA Device ID, and `size` is the requested memory size to allocate, and `mem_size` is the rounded memory size to be allocated, and `mem_ptr` is the memory pointer, and `pmem_id` is the pooled memory object ID.

#### Variables

- **file** – Output file\_like object that redirect to.
- **flush** – If True, this hook forcibly flushes the text stream at the end of print. The default is True.

#### Methods

`__enter__(self)`

`__exit__(self, *_)`

`alloc_postprocess(self, **kwargs)`

Callback function invoked after allocating memory from GPU device.

##### Keyword Arguments

- **device\_id** (*int*) – CUDA device ID
- **mem\_size** (*int*) – Rounded memory bytesize allocated
- **mem\_ptr** (*int*) – Obtained memory pointer. 0 if an error occurred in allocation.

`alloc_preprocess(self, **kwargs)`

Callback function invoked before allocating memory from GPU device.

##### Keyword Arguments

- **device\_id** (*int*) – CUDA device ID
- **mem\_size** (*int*) – Rounded memory bytesize to be allocated

`free_postprocess(self, **kwargs)`

Callback function invoked after releasing memory to memory pool.

##### Keyword Arguments

- **device\_id** (*int*) – CUDA device ID
- **mem\_size** (*int*) – Memory bytesize
- **mem\_ptr** (*int*) – Memory pointer to free
- **pmem\_id** (*int*) – Pooled memory object ID.

`free_preprocess(self, **kwargs)`

Callback function invoked before releasing memory to memory pool.

##### Keyword Arguments

- **device\_id** (*int*) – CUDA device ID
- **mem\_size** (*int*) – Memory bytesize

- `mem_ptr(int)` – Memory pointer to free
- `pmem_id(int)` – Pooled memory object ID.

`malloc_postprocess(self, **kwargs)`

Callback function invoked after retrieving memory from memory pool.

#### Keyword Arguments

- `device_id(int)` – CUDA device ID
- `size(int)` – Requested memory bytesize to allocate
- `mem_size(int)` – Rounded memory bytesize allocated
- `mem_ptr(int)` – Obtained memory pointer. 0 if an error occurred in `malloc`.
- `pmem_id(int)` – Pooled memory object ID. 0 if an error occurred in `malloc`.

`malloc_preprocess(self, **kwargs)`

Callback function invoked before retrieving memory from memory pool.

#### Keyword Arguments

- `device_id(int)` – CUDA device ID
- `size(int)` – Requested memory bytesize to allocate
- `mem_size(int)` – Rounded memory bytesize to be allocated

`__eq__(value, /)`

Return `self==value`.

`__ne__(value, /)`

Return `self!=value`.

`__lt__(value, /)`

Return `self<value`.

`__le__(value, /)`

Return `self<=value`.

`__gt__(value, /)`

Return `self>value`.

`__ge__(value, /)`

Return `self>=value`.

#### Attributes

`name = 'DebugPrintHook'`

### `cupy.cuda.memory_hooks.LineProfileHook`

`class cupy.cuda.memory_hooks.LineProfileHook(max_depth=0)`

Code line CuPy memory profiler.

This profiler shows line-by-line GPU memory consumption using `traceback` module. But, note that it can trace only CPython level, no Cython level. ref. <https://github.com/cython/cython/issues/1755>



## Example

Code example:

```

from cupy.cuda import memory_hooks
hook = memory_hooks.LineProfileHook()
with hook:
    # some CuPy codes
hook.print_report()

```

Output example:

```

_root (4.00KB, 4.00KB)
  lib/python3.6/unittest/__main__.py:18:<module> (4.00KB, 4.00KB)
    lib/python3.6/unittest/main.py:255:runTests (4.00KB, 4.00KB)
      tests/cupy_tests/test.py:37:test (1.00KB, 1.00KB)
      tests/cupy_tests/test.py:38:test (1.00KB, 1.00KB)
      tests/cupy_tests/test.py:39:test (2.00KB, 2.00KB)

```

Each line shows:

```
{filename}:{lineno}:{func_name} ({used_bytes}, {acquired_bytes})
```

where *used\_bytes* is the memory bytes used from CuPy memory pool, and *acquired\_bytes* is the actual memory bytes the CuPy memory pool acquired from GPU device. *\_root* is a root node of the stack trace to show total memory usage.

**Parameters** *max\_depth* (*int*) – maximum depth to follow stack traces. Default is 0 (no limit).

## Methods

**\_\_enter\_\_**(*self*)

**\_\_exit\_\_**(*self*, \*\_)

**alloc\_postprocess**(*self*, \*\*kwargs)

Callback function invoked after allocating memory from GPU device.

### Keyword Arguments

- **device\_id** (*int*) – CUDA device ID
- **mem\_size** (*int*) – Rounded memory bytesize allocated
- **mem\_ptr** (*int*) – Obtained memory pointer. 0 if an error occurred in allocation.

**alloc\_preprocess**(*self*, \*\*kwargs)

Callback function invoked before allocating memory from GPU device.

### Keyword Arguments

- **device\_id** (*int*) – CUDA device ID
- **mem\_size** (*int*) – Rounded memory bytesize to be allocated

**free\_postprocess**(*self*, \*\*kwargs)

Callback function invoked after releasing memory to memory pool.

### Keyword Arguments

- **device\_id** (*int*) – CUDA device ID

- **mem\_size** (*int*) – Memory bytesize
- **mem\_ptr** (*int*) – Memory pointer to free
- **pmem\_id** (*int*) – Pooled memory object ID.

**free\_preprocess**(*self*, *\*\*kwargs*)

Callback function invoked before releasing memory to memory pool.

**Keyword Arguments**

- **device\_id** (*int*) – CUDA device ID
- **mem\_size** (*int*) – Memory bytesize
- **mem\_ptr** (*int*) – Memory pointer to free
- **pmem\_id** (*int*) – Pooled memory object ID.

**malloc\_postprocess**(*self*, *\*\*kwargs*)

Callback function invoked after retrieving memory from memory pool.

**Keyword Arguments**

- **device\_id** (*int*) – CUDA device ID
- **size** (*int*) – Requested memory bytesize to allocate
- **mem\_size** (*int*) – Rounded memory bytesize allocated
- **mem\_ptr** (*int*) – Obtained memory pointer. 0 if an error occurred in malloc.
- **pmem\_id** (*int*) – Pooled memory object ID. 0 if an error occurred in malloc.

**malloc\_preprocess**(*self*, *\*\*kwargs*)

Callback function invoked before retrieving memory from memory pool.

**Keyword Arguments**

- **device\_id** (*int*) – CUDA device ID
- **size** (*int*) – Requested memory bytesize to allocate
- **mem\_size** (*int*) – Rounded memory bytesize to be allocated

**print\_report**(*file*=<\_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>)

Prints a report of line memory profiling.

**\_\_eq\_\_**(*value*, /)

Return self==value.

**\_\_ne\_\_**(*value*, /)

Return self!=value.

**\_\_lt\_\_**(*value*, /)

Return self<value.

**\_\_le\_\_**(*value*, /)

Return self<=value.

**\_\_gt\_\_**(*value*, /)

Return self>value.

**\_\_ge\_\_**(*value*, /)

Return self>=value.

## Attributes

`name = 'LineProfileHook'`

## 5.6.4 Streams and events

<code>cupy.cuda.Stream([null, non_blocking, ptds])</code>	CUDA stream.
<code>cupy.cuda.ExternalStream(ptr)</code>	CUDA stream.
<code>cupy.cuda.get_current_stream()</code>	Gets current CUDA stream.
<code>cupy.cuda.Event([block, disable_timing, ...])</code>	CUDA event, a synchronization point of CUDA streams.
<code>cupy.cuda.get_elapsed_time(start_event, ...)</code>	Gets the elapsed time between two events.

### cupy.cuda.Stream

**class** `cupy.cuda.Stream`(*null=False, non\_blocking=False, ptds=False*)  
 CUDA stream.

This class handles the CUDA stream handle in RAII way, i.e., when an Stream instance is destroyed by the GC, its handle is also destroyed.

Note that if both `null` and `ptds` are `False`, a plain new stream is created.

#### Parameters

- **null** (*bool*) – If `True`, the stream is a null stream (i.e. the default stream that synchronizes with all streams). Note that you can also use the `Stream.null` singleton object instead of creating a new null stream object.
- **ptds** (*bool*) – If `True` and `null` is `False`, the per-thread default stream is used. Note that you can also use the `Stream.ptds` singleton object instead of creating a new per-thread default stream object.
- **non\_blocking** (*bool*) – If `True` and both `null` and `ptds` are `False`, the stream does not synchronize with the `NULL` stream.

**Variables** `ptr` (*intptr\_t*) – Raw stream handle.

#### Methods

`__enter__(self)`

`__exit__(self, *args)`

`add_callback(self, callback, arg)`

Adds a callback that is called when all queued work is done.

#### Parameters

- **callback** (*function*) – Callback function. It must take three arguments (Stream object, int error status, and user data object), and returns nothing.
- **arg** (*object*) – Argument to the callback.

---

**Note:** Whenever possible, use the `launch_host_func()` method instead of this one, as it may be deprecated and removed from CUDA at some point.

---

**launch\_host\_func**(*self*, *callback*, *arg*)

Launch a callback on host when all queued work is done.

**Parameters**

- **callback** (*function*) – Callback function. It must take only one argument (user data object), and returns nothing.
- **arg** (*object*) – Argument to the callback.

---

**Note:** Whenever possible, this method is recommended over [add\\_callback\(\)](#), which may be deprecated and removed from CUDA at some point.

---

**See also:**

[cudaLaunchHostFunc\(\)](#)

**record**(*self*, *event=None*)

Records an event on the stream.

**Parameters** **event** (*None* or [cupy.cuda.Event](#)) – CUDA event. If *None*, then a new plain event is created and used.

**Returns** The recorded event.

**Return type** [cupy.cuda.Event](#)

**See also:**

[cupy.cuda.Event.record\(\)](#)

**synchronize**(*self*)

Waits for the stream completing all queued work.

**use**(*self*)

Makes this stream current.

If you want to switch a stream temporarily, use the *with* statement.

**wait\_event**(*self*, *event*)

Makes the stream wait for an event.

The future work on this stream will be done after the event.

**Parameters** **event** ([cupy.cuda.Event](#)) – CUDA event.

**\_\_eq\_\_**(*self*, *other*)

**\_\_ne\_\_**(*value*, /)

Return self!=value.

**\_\_lt\_\_**(*value*, /)

Return self<value.

**\_\_le\_\_**(*value*, /)

Return self<=value.

**\_\_gt\_\_**(*value*, /)

Return self>value.

**\_\_ge\_\_**(*value*, /)

Return self>=value.

## Attributes

### **done**

True if all work on this stream has been done.

**null** = <Stream 0>

**ptds** = <Stream 2>

## cupy.cuda.ExternalStream

**class** cupy.cuda.ExternalStream(*ptr*)

CUDA stream.

This class allows to use external streams in CuPy by providing the stream pointer obtained from the CUDA runtime call. The user is in charge of managing the life-cycle of the stream.

**Parameters** *ptr* (*intptr\_t*) – Address of the *cudaStream\_t* object.

**Variables** *ptr* (*intptr\_t*) – Raw stream handle.

## Methods

**\_\_enter\_\_**(*self*)

**\_\_exit\_\_**(*self*, \**args*)

**add\_callback**(*self*, *callback*, *arg*)

Adds a callback that is called when all queued work is done.

### **Parameters**

- **callback** (*function*) – Callback function. It must take three arguments (Stream object, int error status, and user data object), and returns nothing.
- **arg** (*object*) – Argument to the callback.

---

**Note:** Whenever possible, use the [launch\\_host\\_func\(\)](#) method instead of this one, as it may be deprecated and removed from CUDA at some point.

---

**launch\_host\_func**(*self*, *callback*, *arg*)

Launch a callback on host when all queued work is done.

### **Parameters**

- **callback** (*function*) – Callback function. It must take only one argument (user data object), and returns nothing.
- **arg** (*object*) – Argument to the callback.

---

**Note:** Whenever possible, this method is recommended over [add\\_callback\(\)](#), which may be deprecated and removed from CUDA at some point.

---

### **See also:**

[cudaLaunchHostFunc\(\)](#)

**record**(*self*, *event=None*)

Records an event on the stream.

**Parameters** **event** (*None* or `cupy.cuda.Event`) – CUDA event. If *None*, then a new plain event is created and used.

**Returns** The recorded event.

**Return type** `cupy.cuda.Event`

**See also:**

`cupy.cuda.Event.record()`

**synchronize**(*self*)

Waits for the stream completing all queued work.

**use**(*self*)

Makes this stream current.

If you want to switch a stream temporarily, use the *with* statement.

**wait\_event**(*self*, *event*)

Makes the stream wait for an event.

The future work on this stream will be done after the event.

**Parameters** **event** (`cupy.cuda.Event`) – CUDA event.

**\_\_eq\_\_**(*self*, *other*)

**\_\_ne\_\_**(*value*, /)

Return self!=value.

**\_\_lt\_\_**(*value*, /)

Return self<value.

**\_\_le\_\_**(*value*, /)

Return self<=value.

**\_\_gt\_\_**(*value*, /)

Return self>value.

**\_\_ge\_\_**(*value*, /)

Return self>=value.

## Attributes

**done**

True if all work on this stream has been done.

**null** = **None**

## cupy.cuda.get\_current\_stream

`cupy.cuda.get_current_stream()`

Gets current CUDA stream.

**Returns** The current CUDA stream.

**Return type** `cupy.cuda.Stream`

## cupy.cuda.Event

**class** `cupy.cuda.Event`(*block=False, disable\_timing=False, interprocess=False*)

CUDA event, a synchronization point of CUDA streams.

This class handles the CUDA event handle in RAII way, i.e., when an Event instance is destroyed by the GC, its handle is also destroyed.

### Parameters

- **block** (*bool*) – If True, the event blocks on the `synchronize()` method.
- **disable\_timing** (*bool*) – If True, the event does not prepare the timing data.
- **interprocess** (*bool*) – If True, the event can be passed to other processes.

**Variables** `ptr` (*intptr\_t*) – Raw event handle.

### Methods

**record**(*self, stream=None*)

Records the event to a stream.

**Parameters** `stream` (`cupy.cuda.Stream`) – CUDA stream to record event. The null stream is used by default.

**See also:**

`cupy.cuda.Stream.record()`

**synchronize**(*self*)

Synchronizes all device work to the event.

If the event is created as a blocking event, it also blocks the CPU thread until the event is done.

**\_\_eq\_\_**(*value, /*)

Return `self==value`.

**\_\_ne\_\_**(*value, /*)

Return `self!=value`.

**\_\_lt\_\_**(*value, /*)

Return `self<value`.

**\_\_le\_\_**(*value, /*)

Return `self<=value`.

**\_\_gt\_\_**(*value, /*)

Return `self>value`.

**\_\_ge\_\_**(*value, /*)

Return `self>=value`.

## Attributes

### done

True if the event is done.

## cupy.cuda.get\_elapsed\_time

`cupy.cuda.get_elapsed_time(start_event, end_event)`

Gets the elapsed time between two events.

### Parameters

- **start\_event** ([Event](#)) – Earlier event.
- **end\_event** ([Event](#)) – Later event.

**Returns** Elapsed time in milliseconds.

**Return type** `float`

## 5.6.5 Texture and surface memory

<code>cupy.cuda.texture.ChannelFormatDescriptor(...)</code>	A class that holds the channel format description.
<code>cupy.cuda.texture.CUDAarray(...)</code>	Allocate a CUDA array ( <i>cudaArray_t</i> ) that can be used as texture memory.
<code>cupy.cuda.texture.ResourceDescriptor(...)</code>	A class that holds the resource description.
<code>cupy.cuda.texture.TextureDescriptor([...])</code>	A class that holds the texture description.
<code>cupy.cuda.texture.TextureObject(...)</code>	A class that holds a texture object.
<code>cupy.cuda.texture.SurfaceObject(...)</code>	A class that holds a surface object.
<code>cupy.cuda.texture.TextureReference(...)</code>	A class that holds a texture reference.

## cupy.cuda.texture.ChannelFormatDescriptor

**class** `cupy.cuda.texture.ChannelFormatDescriptor(int x, int y, int z, int w, int f)`

A class that holds the channel format description. Equivalent to `cudaChannelFormatDesc`.

### Parameters

- **x** (*int*) – the number of bits for the x channel.
- **y** (*int*) – the number of bits for the y channel.
- **z** (*int*) – the number of bits for the z channel.
- **w** (*int*) – the number of bits for the w channel.
- **f** (*int*) – the channel format. Use one of the values in `cudaChannelFormat*`, such as `cupy.cuda.runtime.cudaChannelFormatKindFloat`.

See also:

`cudaCreateChannelDesc()`



## Methods

**get\_channel\_format**(*self*)

Returns a dict containing the input.

**\_\_eq\_\_**(*value*, /)

Return self==value.

**\_\_ne\_\_**(*value*, /)

Return self!=value.

**\_\_lt\_\_**(*value*, /)

Return self<value.

**\_\_le\_\_**(*value*, /)

Return self<=value.

**\_\_gt\_\_**(*value*, /)

Return self>value.

**\_\_ge\_\_**(*value*, /)

Return self>=value.

## Attributes

**ptr**

## cupy.cuda.texture.CUDAArray

**class** `cupy.cuda.texture.CUDAArray`(*ChannelFormatDescriptor desc*, *size\_t width*, *size\_t height=0*, *size\_t depth=0*, *unsigned int flags=0*)

Allocate a CUDA array (*cudaArray\_t*) that can be used as texture memory. Depending on the input, either 1D, 2D, or 3D CUDA array is returned.

### Parameters

- **desc** (*ChannelFormatDescriptor*) – an instance of *ChannelFormatDescriptor*.
- **width** (*int*) – the width (in elements) of the array.
- **height** (*int*, *optional*) – the height (in elements) of the array.
- **depth** (*int*, *optional*) – the depth (in elements) of the array.
- **flags** (*int*, *optional*) – the flag for extensions. Use one of the values in `cudaArray*`, such as `cupy.cuda.runtime.cudaArrayDefault`.

**Warning:** The memory allocation of *CUDAArray* is done outside of CuPy’s memory management (enabled by default) due to CUDA’s limitation. Users of *CUDAArray* should be cautious about any out-of-memory possibilities.

See also:

`cudaMalloc3DArray()`

## Methods

**copy\_from**(*self*, *in\_arr*, *stream=None*)

Copy data from device or host array to CUDA array.

### Parameters

- **in\_arr** (`cupy.ndarray` or `numpy.ndarray`) –
- **stream** (`cupy.cuda.Stream`) – if not `None`, an asynchronous copy is performed.

---

**Note:** For CUDA arrays with different dimensions, the requirements for the shape of the input array are given as follows:

- 1D: (nch \* width,)
- 2D: (height, nch \* width)
- 3D: (depth, height, nch \* width)

where nch is the number of channels specified in `desc`.

---

**copy\_to**(*self*, *out\_arr*, *stream=None*)

Copy data from CUDA array to device or host array.

### Parameters

- **out\_arr** (`cupy.ndarray` or `numpy.ndarray`) – must be C-contiguous
- **stream** (`cupy.cuda.Stream`) – if not `None`, an asynchronous copy is performed.

---

**Note:** For CUDA arrays with different dimensions, the requirements for the shape of the output array are given as follows:

- 1D: (nch \* width,)
- 2D: (height, nch \* width)
- 3D: (depth, height, nch \* width)

where nch is the number of channels specified in `desc`.

---

**\_\_eq\_\_**(*value*, /)

Return self==value.

**\_\_ne\_\_**(*value*, /)

Return self!=value.

**\_\_lt\_\_**(*value*, /)

Return self<value.

**\_\_le\_\_**(*value*, /)

Return self<=value.

**\_\_gt\_\_**(*value*, /)

Return self>value.

**\_\_ge\_\_**(*value*, /)

Return self>=value.

### Attributes

**depth**  
**desc**  
**flags**  
**height**  
**ndim**  
**ptr**  
**width**

### cupy.cuda.texture.ResourceDescriptor

```
class cupy.cuda.texture.ResourceDescriptor(int restype, CUDAarray cuArr=None, ndarray arr=None,
                                         ChannelFormatDescriptor chDesc=None, size_t
                                         sizeInBytes=0, size_t width=0, size_t height=0, size_t
                                         pitchInBytes=0)
```

A class that holds the resource description. Equivalent to `cudaResourceDesc`.

#### Parameters

- **restype** (*int*) – the resource type. Use one of the values in `cudaResourceType*`, such as `cupy.cuda.runtime.cudaResourceTypeArray`.
- **cuArr** (*CUDAarray*, *optional*) – An instance of *CUDAarray*, required if `restype` is set to `cupy.cuda.runtime.cudaResourceTypeArray`.
- **arr** (*cupy.ndarray*, *optional*) – An instance of *ndarray*, required if `restype` is set to `cupy.cuda.runtime.cudaResourceTypeLinear` or `cupy.cuda.runtime.cudaResourceTypePitch2D`.
- **chDesc** (*ChannelFormatDescriptor*, *optional*) – an instance of *ChannelFormatDescriptor*, required if `restype` is set to `cupy.cuda.runtime.cudaResourceTypeLinear` or `cupy.cuda.runtime.cudaResourceTypePitch2D`.
- **sizeInBytes** (*int*, *optional*) – total bytes in the linear memory, required if `restype` is set to `cupy.cuda.runtime.cudaResourceTypeLinear`.
- **width** (*int*, *optional*) – the width (in elements) of the 2D array, required if `restype` is set to `cupy.cuda.runtime.cudaResourceTypePitch2D`.
- **height** (*int*, *optional*) – the height (in elements) of the 2D array, required if `restype` is set to `cupy.cuda.runtime.cudaResourceTypePitch2D`.
- **pitchInBytes** (*int*, *optional*) – the number of bytes per pitch-aligned row, required if `restype` is set to `cupy.cuda.runtime.cudaResourceTypePitch2D`.

---

**Note:** A texture backed by *mipmap* arrays is currently not supported in CuPy.

---

See also:

`cudaCreateTextureObject()`

## Methods

**get\_resource\_desc**(*self*)

Returns a dict containing the input.

**\_\_eq\_\_**(*value*, /)

Return self==value.

**\_\_ne\_\_**(*value*, /)

Return self!=value.

**\_\_lt\_\_**(*value*, /)

Return self<value.

**\_\_le\_\_**(*value*, /)

Return self<=value.

**\_\_gt\_\_**(*value*, /)

Return self>value.

**\_\_ge\_\_**(*value*, /)

Return self>=value.

## Attributes

**arr**

**chDesc**

**cuArr**

**ptr**

## cupy.cuda.texture.TextureDescriptor

**class** `cupy.cuda.texture.TextureDescriptor`(*addressModes=None*, *int filterMode=0*, *int readMode=0*, *sRGB=None*, *borderColors=None*, *normalizedCoords=None*, *maxAnisotropy=None*)

A class that holds the texture description. Equivalent to `cudaTextureDesc`.

### Parameters

- **addressModes** (*tuple* or *list*) – an iterable with length up to 3, each element is one of the values in `cudaAddressMode*`, such as `cupy.cuda.runtime.cudaAddressModeWrap`.
- **filterMode** (*int*) – the filter mode. Use one of the values in `cudaFilterMode*`, such as `cupy.cuda.runtime.cudaFilterModePoint`.
- **readMode** (*int*) – the read mode. Use one of the values in `cudaReadMode*`, such as `cupy.cuda.runtime.cudaReadModeElementType`.
- **normalizedCoords** (*int*) – whether coordinates are normalized or not.
- **sRGB** (*int*, *optional*) –
- **borderColors** (*tuple* or *list*, *optional*) – an iterable with length up to 4.
- **maxAnisotropy** (*int*, *optional*) –

---

**Note:** A texture backed by *mipmap* arrays is currently not supported in CuPy.

---

**See also:**

`cudaCreateTextureObject()`

## Methods

**get\_texture\_desc**(*self*)

Returns a dict containing the input.

**\_\_eq\_\_**(*value*, /)

Return self==value.

**\_\_ne\_\_**(*value*, /)

Return self!=value.

**\_\_lt\_\_**(*value*, /)

Return self<value.

**\_\_le\_\_**(*value*, /)

Return self<=value.

**\_\_gt\_\_**(*value*, /)

Return self>value.

**\_\_ge\_\_**(*value*, /)

Return self>=value.

## Attributes

**ptr**

## cupy.cuda.texture.TextureObject

**class** `cupy.cuda.texture.TextureObject`(*ResourceDescriptor ResDesc*, *TextureDescriptor TexDesc*)

A class that holds a texture object. Equivalent to `cudaTextureObject_t`. The returned *TextureObject* instance can be passed as a argument when launching *RawKernel* or *ElementwiseKernel*.

### Parameters

- **ResDesc** (*ResourceDescriptor*) – an instance of the resource descriptor.
- **TexDesc** (*TextureDescriptor*) – an instance of the texture descriptor.

**See also:**

`cudaCreateTextureObject()`

## Methods

**\_\_eq\_\_**(*value*, /)  
Return self==value.

**\_\_ne\_\_**(*value*, /)  
Return self!=value.

**\_\_lt\_\_**(*value*, /)  
Return self<value.

**\_\_le\_\_**(*value*, /)  
Return self<=value.

**\_\_gt\_\_**(*value*, /)  
Return self>value.

**\_\_ge\_\_**(*value*, /)  
Return self>=value.

## Attributes

**ResDesc**

**TexDesc**

**ptr**

## cupy.cuda.texture.SurfaceObject

**class** `cupy.cuda.texture.SurfaceObject`(*ResourceDescriptor ResDesc*)

A class that holds a surface object. Equivalent to `cudaSurfaceObject_t`. The returned *SurfaceObject* instance can be passed as a argument when launching *RawKernel*.

**Parameters** **ResDesc** (*ResourceDescriptor*) – an intance of the resource descriptor.

**See also:**

`cudaCreateSurfaceObject()`

## Methods

**\_\_eq\_\_**(*value*, /)  
Return self==value.

**\_\_ne\_\_**(*value*, /)  
Return self!=value.

**\_\_lt\_\_**(*value*, /)  
Return self<value.

**\_\_le\_\_**(*value*, /)  
Return self<=value.

**\_\_gt\_\_**(*value*, /)  
Return self>value.

**\_\_ge\_\_**(*value*, /)  
Return self>=value.

## Attributes

**ResDesc**

**ptr**

## cupy.cuda.texture.TextureReference

**class** `cupy.cuda.texture.TextureReference(intptr_t texref, ResourceDescriptor ResDesc, TextureDescriptor TexDesc)`

A class that holds a texture reference. Equivalent to `CUtexref` (the driver API is used under the hood).

### Parameters

- **texref** (*intptr\_t*) – a handle to the texture reference declared in the CUDA source code. This can be obtained by calling `get_texref()`.
- **ResDesc** (*ResourceDescriptor*) – an instance of the resource descriptor.
- **TexDesc** (*TextureDescriptor*) – an instance of the texture descriptor.

**Warning:** As of CUDA Toolkit v10.1, the Texture Reference API (in both driver and runtime) is marked as deprecated. To help transition to the new Texture Object API, this class mimics the usage of `TextureObject`. Users who have legacy CUDA codes that use texture references should consider migration to the new API.

This CuPy interface is subject to removal once the official NVIDIA support is dropped in the future.

### See also:

`TextureObject`, `cudaCreateTextureObject()`

## Methods

`__eq__(value, /)`  
Return `self==value`.

`__ne__(value, /)`  
Return `self!=value`.

`__lt__(value, /)`  
Return `self<value`.

`__le__(value, /)`  
Return `self<=value`.

`__gt__(value, /)`  
Return `self>value`.

`__ge__(value, /)`  
Return `self>=value`.

**Attributes****ResDesc****TexDesc****texref**

## 5.6.6 Profiler

<code>cupy.cuda.profile()</code>	Enable CUDA profiling during with statement.
<code>cupy.cuda.profiler.initialize(...)</code>	Initialize the CUDA profiler.
<code>cupy.cuda.profiler.start()</code>	Enable profiling.
<code>cupy.cuda.profiler.stop()</code>	Disable profiling.
<code>cupy.cuda.nvtx.Mark(message, int id_color=-1)</code>	Marks an instantaneous event (marker) in the application.
<code>cupy.cuda.nvtx.MarkC(message, uint32_t color=0)</code>	Marks an instantaneous event (marker) in the application.
<code>cupy.cuda.nvtx.RangePush(message, ...)</code>	Starts a nested range.
<code>cupy.cuda.nvtx.RangePushC(message, ...)</code>	Starts a nested range.
<code>cupy.cuda.nvtx.RangePop()</code>	Ends a nested range.

### `cupy.cuda.profile`

**`cupy.cuda.profile()`**

Enable CUDA profiling during with statement.

This function enables profiling on entering a with statement, and disables profiling on leaving the statement.

```
>>> with cupy.cuda.profile():  
...     # do something you want to measure  
...     pass
```

---

**Note:** When starting nvprof from the command line, manually setting `--profile-from-start off` may be required for the desired behavior.

---

### `cupy.cuda.profiler.initialize`

**`cupy.cuda.profiler.initialize(unicode config_file, unicode output_file, int output_mode)`**

Initialize the CUDA profiler.

This function initialize the CUDA profiler. See the CUDA document for detail.

**Parameters**

- **config\_file** (*str*) – Name of the configuration file.
- **output\_file** (*str*) – Name of the output file.
- **output\_mode** (*int*) – `cupy.cuda.profiler.cudaKeyValuePair` or `cupy.cuda.profiler.cudaCSV`.



### cupy.cuda.profiler.start

`cupy.cuda.profiler.start()`

Enable profiling.

A user can enable CUDA profiling. When an error occurs, it raises an exception.

See the CUDA document for detail.

### cupy.cuda.profiler.stop

`cupy.cuda.profiler.stop()`

Disable profiling.

A user can disable CUDA profiling. When an error occurs, it raises an exception.

See the CUDA document for detail.

### cupy.cuda.nvtx.Mark

`cupy.cuda.nvtx.Mark(message, int id_color=-1)`

Marks an instantaneous event (marker) in the application.

Markers are used to describe events at a specific time during execution of the application.

#### Parameters

- **message** (*str*) – Name of a marker.
- **id\_color** (*int*) – ID of color for a marker.

### cupy.cuda.nvtx.MarkC

`cupy.cuda.nvtx.MarkC(message, uint32_t color=0)`

Marks an instantaneous event (marker) in the application.

Markers are used to describe events at a specific time during execution of the application.

#### Parameters

- **message** (*str*) – Name of a marker.
- **color** (*uint32*) – Color code for a marker.

### cupy.cuda.nvtx.RangePush

`cupy.cuda.nvtx.RangePush(message, int id_color=-1)`

Starts a nested range.

Ranges are used to describe events over a time span during execution of the application. The duration of a range is defined by the corresponding pair of `RangePush()` to `RangePop()` calls.

#### Parameters

- **message** (*str*) – Name of a range.
- **id\_color** (*int*) – ID of color for a range.

### cupy.cuda.nvtx.RangePushC

`cupy.cuda.nvtx.RangePushC(message, uint32_t color=0)`

Starts a nested range.

Ranges are used to describe events over a time span during execution of the application. The duration of a range is defined by the corresponding pair of `RangePush*()` to `RangePop()` calls.

#### Parameters

- **message** (*str*) – Name of a range.
- **color** (*uint32*) – ARGB color for a range.

### cupy.cuda.nvtx.RangePop

`cupy.cuda.nvtx.RangePop()`

Ends a nested range.

Ranges are used to describe events over a time span during execution of the application. The duration of a range is defined by the corresponding pair of `RangePush*()` to `RangePop()` calls.

## 5.6.7 NCCL

---

<code>cupy.cuda.nccl.NcclCommunicator(int ndev, ...)</code>	Initialize an NCCL communicator for one device controlled by one process.
<code>cupy.cuda.nccl.get_build_version()</code>	
<code>cupy.cuda.nccl.get_version()</code>	Returns the runtime version of NCCL.
<code>cupy.cuda.nccl.get_unique_id()</code>	
<code>cupy.cuda.nccl.groupStart()</code>	Start a group of NCCL calls.
<code>cupy.cuda.nccl.groupEnd()</code>	End a group of NCCL calls.

---

### cupy.cuda.nccl.NcclCommunicator

**class** `cupy.cuda.nccl.NcclCommunicator(int ndev, tuple commId, int rank)`

Initialize an NCCL communicator for one device controlled by one process.

#### Parameters

- **ndev** (*int*) – Total number of GPUs to be used.
- **commId** (*tuple*) – The unique ID returned by `get_unique_id()`.
- **rank** (*int*) – The rank of the GPU managed by the current process.

**Returns** An `NcclCommunicator` instance.

**Return type** `NcclCommunicator`

---

**Note:** This method is for creating an NCCL communicator in a multi-process environment, typically managed by MPI or multiprocessing. For controlling multiple devices by one process, use `initAll()` instead.

---

See also:

[ncclCommInitRank](#)

## Methods

**abort**(*self*)

**allGather**(*self*, *intptr\_t* sendbuf, *intptr\_t* recvbuf, *size\_t* count, *int* datatype, *intptr\_t* stream)

**allReduce**(*self*, *intptr\_t* sendbuf, *intptr\_t* recvbuf, *size\_t* count, *int* datatype, *int* op, *intptr\_t* stream)

**bcast**(*self*, *intptr\_t* buff, *int* count, *int* datatype, *int* root, *intptr\_t* stream)

**broadcast**(*self*, *intptr\_t* sendbuff, *intptr\_t* recvbuff, *int* count, *int* datatype, *int* root, *intptr\_t* stream)

**check\_async\_error**(*self*)

**destroy**(*self*)

**device\_id**(*self*)

**static initAll**(*devices*)

Initialize NCCL communicators for multiple devices in a single process.

**Parameters** *devices* (*int* or *list of int*) – The number of GPUs or a list of GPUs to be used. For the former case, the first *devices* GPUs will be used.

**Returns** A list of `NcclCommunicator` instances.

**Return type** `list`

**Note:** This method is for creating a group of NCCL communicators, each controlling one device, in a single process like this:

```
from cupy.cuda import nccl
# Use 3 GPUs: #0, #2, and #3
comms = nccl.NcclCommunicator.initAll([0, 2, 3])
assert len(comms) == 3
```

In a multi-process setup, use the default initializer instead.

See also:

[ncclCommInitAll](#)

**rank\_id**(*self*)

**recv**(*self*, *intptr\_t* recvbuf, *size\_t* count, *int* datatype, *int* peer, *intptr\_t* stream)

**reduce**(*self*, *intptr\_t* sendbuf, *intptr\_t* recvbuf, *size\_t* count, *int* datatype, *int* op, *int* root, *intptr\_t* stream)

**reduceScatter**(*self*, *intptr\_t* sendbuf, *intptr\_t* recvbuf, *size\_t* recvcnt, *int* datatype, *int* op, *intptr\_t* stream)

**send**(*self*, *intptr\_t* sendbuf, *size\_t* count, *int* datatype, *int* peer, *intptr\_t* stream)

**size**(*self*)

**\_\_eq\_\_**(*value*, /)

Return self==value.

**\_\_ne\_\_**(*value*, /)

Return self!=value.

```
__lt__(value, /)
    Return self<value.

__le__(value, /)
    Return self<=value.

__gt__(value, /)
    Return self>value.

__ge__(value, /)
    Return self>=value.
```

### `cupy.cuda.nccl.get_build_version`

`cupy.cuda.nccl.get_build_version()`

### `cupy.cuda.nccl.get_version`

`cupy.cuda.nccl.get_version()`  
Returns the runtime version of NCCL.

This function will return 0 when built with NCCL version earlier than 2.3.4, which does not support `ncclGetVersion` API.

### `cupy.cuda.nccl.get_unique_id`

`cupy.cuda.nccl.get_unique_id()`

### `cupy.cuda.nccl.groupStart`

`cupy.cuda.nccl.groupStart()`  
Start a group of NCCL calls. Must be paired with [`groupEnd\(\)`](#).

---

**Note:** This method is useful when the `NcclCommunicator` instances are created via [`initAll\(\)`](#). A typical usage pattern is like this:

```
comms = cupy.cuda.nccl.NcclCommunicator.initAll(n, dev_list)
# ... do some preparation work
cupy.cuda.nccl.groupStart()
for rank, comm in enumerate(comms):
    # ... make some collective calls ...
cupy.cuda.nccl.groupEnd()
```

Other use cases include fusing several NCCL calls into one, and point-to-point communications using [`send\(\)`](#) and [`recv\(\)`](#) (with NCCL 2.7+).

---

**See also:**

[ncclGroupStart](#), [Group Calls](#)

## cupy.cuda.nccl.groupEnd

`cupy.cuda.nccl.groupEnd()`

End a group of NCCL calls. Must be paired with [groupStart\(\)](#).

**Note:** This method is useful when the `NcclCommunicator` instances are created via [initAll\(\)](#). A typical usage pattern is like this:

```

comms = cupy.cuda.nccl.NcclCommunicator.initAll(n, dev_list)
# ... do some preparation work
cupy.cuda.nccl.groupStart()
for rank, comm in enumerate(comms):
    # ... make some collective calls ...
cupy.cuda.nccl.groupEnd()

```

Other use cases include fusing several NCCL calls into one, and point-to-point communications using [send\(\)](#) and [recv\(\)](#) (with NCCL 2.7+).

**See also:**

[ncclGroupEnd](#), [Group Calls](#)

## 5.6.8 Runtime API

CuPy wraps CUDA Runtime APIs to provide the native CUDA operations. Please check the [CUDA Runtime API documentation](#) to use these functions.

---

`cupy.cuda.runtime.driverGetVersion()`

---

`cupy.cuda.runtime.runtimeGetVersion()`

---

`cupy.cuda.runtime.getDevice()`

---

`cupy.cuda.runtime.getDeviceProperties(int device)`

---

`cupy.cuda.runtime.deviceGetAttribute(...)`

---

`cupy.cuda.runtime.deviceGetByPCIBusId(...)`

---

`cupy.cuda.runtime.deviceGetPCIBusId(int device)`

---

<code>cupy.cuda.runtime.deviceGetDefaultMemPool(...)</code>	Get the default mempool on the current device.
---	--

---

<code>cupy.cuda.runtime.deviceGetMemPool(int device)</code>	Get the current mempool on the current device.
---	--

---

<code>cupy.cuda.runtime.deviceSetMemPool(...)</code>	Set the current mempool on the current device to pool.
--	--

---

`cupy.cuda.runtime.memPoolTrimTo(...)`

---

`cupy.cuda.runtime.getDeviceCount()`

---

continues on next page

Table 135 – continued from previous page

---

<code>cupy.cuda.runtime.setDevice(int device)</code>
<code>cupy.cuda.runtime.deviceSynchronize()</code>
<code>cupy.cuda.runtime.deviceCanAccessPeer(...)</code>
<code>cupy.cuda.runtime. deviceEnablePeerAccess(...)</code>
<code>cupy.cuda.runtime.deviceGetLimit(int limit)</code>
<code>cupy.cuda.runtime.deviceSetLimit(int limit, ...)</code>
<code>cupy.cuda.runtime.malloc(size_t size)</code>
<code>cupy.cuda.runtime.mallocManaged(size_t size, ...)</code>
<code>cupy.cuda.runtime.malloc3DArray(...)</code>
<code>cupy.cuda.runtime.mallocArray(...)</code>
<code>cupy.cuda.runtime.mallocAsync(size_t size, ...)</code>
<code>cupy.cuda.runtime.hostAlloc(size_t size, ...)</code>
<code>cupy.cuda.runtime.hostRegister(intptr_t ptr, ...)</code>
<code>cupy.cuda.runtime.hostUnregister(intptr_t ptr)</code>
<code>cupy.cuda.runtime.free(intptr_t ptr)</code>
<code>cupy.cuda.runtime.freeHost(intptr_t ptr)</code>
<code>cupy.cuda.runtime.freeArray(intptr_t ptr)</code>
<code>cupy.cuda.runtime.freeAsync(intptr_t ptr, ...)</code>
<code>cupy.cuda.runtime.memGetInfo()</code>
<code>cupy.cuda.runtime.memcpy(intptr_t dst, ...)</code>
<code>cupy.cuda.runtime.memcpyAsync(intptr_t dst, ...)</code>
<code>cupy.cuda.runtime.memcpyPeer(intptr_t dst, ...)</code>
<code>cupy.cuda.runtime.memcpyPeerAsync(...)</code>
<code>cupy.cuda.runtime.memcpy2D(intptr_t dst, ...)</code>
<code>cupy.cuda.runtime.memcpy2DAsync(...)</code>

---

continues on next page

Table 135 – continued from previous page

<code>cupy.cuda.runtime.memcpy2DFromArray(...)</code>
<code>cupy.cuda.runtime. memcpy2DFromArrayAsync(...)</code>
<code>cupy.cuda.runtime.memcpy2DToArray(...)</code>
<code>cupy.cuda.runtime.memcpy2DToArrayAsync(...)</code>
<code>cupy.cuda.runtime.memcpy3D(...)</code>
<code>cupy.cuda.runtime.memcpy3DAsync(...)</code>
<code>cupy.cuda.runtime.memset(intptr_t ptr, ...)</code>
<code>cupy.cuda.runtime.memsetAsync(intptr_t ptr, ...)</code>
<code>cupy.cuda.runtime.memPrefetchAsync(...)</code>
<code>cupy.cuda.runtime.memAdvise(intptr_t devPtr, ...)</code>
<code>cupy.cuda.runtime.pointerGetAttributes(...)</code>
<code>cupy.cuda.runtime.streamCreate()</code>
<code>cupy.cuda.runtime. streamCreateWithFlags(...)</code>
<code>cupy.cuda.runtime.streamDestroy(intptr_t stream)</code>
<code>cupy.cuda.runtime.streamSynchronize(...)</code>
<code>cupy.cuda.runtime.streamAddCallback(...)</code>
<code>cupy.cuda.runtime.streamQuery(intptr_t stream)</code>
<code>cupy.cuda.runtime.streamWaitEvent(...)</code>
<code>cupy.cuda.runtime.launchHostFunc(...)</code>
<code>cupy.cuda.runtime.eventCreate()</code>
<code>cupy.cuda.runtime.eventCreateWithFlags(...)</code>
<code>cupy.cuda.runtime.eventDestroy(intptr_t event)</code>
<code>cupy.cuda.runtime.eventElapsedTime(...)</code>
<code>cupy.cuda.runtime.eventQuery(intptr_t event)</code>
<code>cupy.cuda.runtime.eventRecord(...)</code>

continues on next page

Table 135 – continued from previous page

---

`cupy.cuda.runtime.eventSynchronize(...)`

---

`cupy.cuda.runtime.ipcGetMemHandle(...)`

---

`cupy.cuda.runtime.ipcOpenMemHandle(...)`

---

`cupy.cuda.runtime.ipcCloseMemHandle(...)`

---

`cupy.cuda.runtime.ipcGetEventHandle(...)`

---

`cupy.cuda.runtime.ipcOpenEventHandle(...)`

---

### **cupy.cuda.runtime.driverGetVersion**

`cupy.cuda.runtime.driverGetVersion() → int`

### **cupy.cuda.runtime.runtimeGetVersion**

`cupy.cuda.runtime.runtimeGetVersion() → int`

### **cupy.cuda.runtime.getDevice**

`cupy.cuda.runtime.getDevice() → int`

### **cupy.cuda.runtime.getDeviceProperties**

`cupy.cuda.runtime.getDeviceProperties(int device)`

### **cupy.cuda.runtime.deviceGetAttribute**

`cupy.cuda.runtime.deviceGetAttribute(int attrib, int device) → int`

### **cupy.cuda.runtime.deviceGetByPCIBusId**

`cupy.cuda.runtime.deviceGetByPCIBusId(unicode pci_bus_id) → int`

### **cupy.cuda.runtime.deviceGetPCIBusId**

`cupy.cuda.runtime.deviceGetPCIBusId(int device) → unicode`



**cupy.cuda.runtime.deviceGetDefaultMemPool**

`cupy.cuda.runtime.deviceGetDefaultMemPool(int device) → intptr_t`  
Get the default mempool on the current device.

**cupy.cuda.runtime.deviceGetMemPool**

`cupy.cuda.runtime.deviceGetMemPool(int device) → intptr_t`  
Get the current mempool on the current device.

**cupy.cuda.runtime.deviceSetMemPool**

`cupy.cuda.runtime.deviceSetMemPool(int device, intptr_t pool)`  
Set the current mempool on the current device to pool.

**cupy.cuda.runtime.memPoolTrimTo**

`cupy.cuda.runtime.memPoolTrimTo(intptr_t pool, size_t size)`

**cupy.cuda.runtime.getDeviceCount**

`cupy.cuda.runtime.getDeviceCount() → int`

**cupy.cuda.runtime.setDevice**

`cupy.cuda.runtime.setDevice(int device)`

**cupy.cuda.runtime.deviceSynchronize**

`cupy.cuda.runtime.deviceSynchronize()`

**cupy.cuda.runtime.deviceCanAccessPeer**

`cupy.cuda.runtime.deviceCanAccessPeer(int device, int peerDevice) → int`

**cupy.cuda.runtime.deviceEnablePeerAccess**

`cupy.cuda.runtime.deviceEnablePeerAccess(int peerDevice)`

**cupy.cuda.runtime.deviceGetLimit**

`cupy.cuda.runtime.deviceGetLimit(int limit) → size_t`

**cupy.cuda.runtime.deviceSetLimit**

`cupy.cuda.runtime.deviceSetLimit(int limit, size_t value)`

**cupy.cuda.runtime.malloc**

`cupy.cuda.runtime.malloc(size_t size) → intptr_t`

**cupy.cuda.runtime.mallocManaged**

`cupy.cuda.runtime.mallocManaged(size_t size, unsigned int flags=cudaMemAttachGlobal) → intptr_t`

**cupy.cuda.runtime.malloc3DArray**

`cupy.cuda.runtime.malloc3DArray(intptr_t descPtr, size_t width, size_t height, size_t depth, unsigned int flags=0) → intptr_t`

**cupy.cuda.runtime.mallocArray**

`cupy.cuda.runtime.mallocArray(intptr_t descPtr, size_t width, size_t height, unsigned int flags=0) → intptr_t`

**cupy.cuda.runtime.mallocAsync**

`cupy.cuda.runtime.mallocAsync(size_t size, intptr_t stream) → intptr_t`

**cupy.cuda.runtime.hostAlloc**

`cupy.cuda.runtime.hostAlloc(size_t size, unsigned int flags) → intptr_t`

**cupy.cuda.runtime.hostRegister**

`cupy.cuda.runtime.hostRegister(intptr_t ptr, size_t size, unsigned int flags)`

**cupy.cuda.runtime.hostUnregister**

`cupy.cuda.runtime.hostUnregister(intptr_t ptr)`

**cupy.cuda.runtime.free**

`cupy.cuda.runtime.free(intptr_t ptr)`

**cupy.cuda.runtime.freeHost**

`cupy.cuda.runtime.freeHost(intptr_t ptr)`

**cupy.cuda.runtime.freeArray**

`cupy.cuda.runtime.freeArray(intptr_t ptr)`

**cupy.cuda.runtime.freeAsync**

`cupy.cuda.runtime.freeAsync(intptr_t ptr, intptr_t stream)`

**cupy.cuda.runtime.memGetInfo**

`cupy.cuda.runtime.memGetInfo()`

**cupy.cuda.runtime.memcpy**

`cupy.cuda.runtime.memcpy(intptr_t dst, intptr_t src, size_t size, int kind)`

**cupy.cuda.runtime.memcpyAsync**

`cupy.cuda.runtime.memcpyAsync(intptr_t dst, intptr_t src, size_t size, int kind, intptr_t stream)`

**cupy.cuda.runtime.memcpyPeer**

`cupy.cuda.runtime.memcpyPeer(intptr_t dst, int dstDevice, intptr_t src, int srcDevice, size_t size)`

**cupy.cuda.runtime.memcpyPeerAsync**

`cupy.cuda.runtime.memcpyPeerAsync(intptr_t dst, int dstDevice, intptr_t src, int srcDevice, size_t size, intptr_t stream)`

**cupy.cuda.runtime.memcpy2D**

`cupy.cuda.runtime.memcpy2D(intptr_t dst, size_t dpitch, intptr_t src, size_t spitch, size_t width, size_t height, MemoryKind kind)`

**cupy.cuda.runtime.memcpy2DAsync**

`cupy.cuda.runtime.memcpy2DAsync(intptr_t dst, size_t dpitch, intptr_t src, size_t spitch, size_t width, size_t height, MemoryKind kind, intptr_t stream)`

**cupy.cuda.runtime.memcpy2DFromArray**

`cupy.cuda.runtime.memcpy2DFromArray(intptr_t dst, size_t dpitch, intptr_t src, size_t wOffset, size_t hOffset, size_t width, size_t height, int kind)`

**cupy.cuda.runtime.memcpy2DFromArrayAsync**

`cupy.cuda.runtime.memcpy2DFromArrayAsync(intptr_t dst, size_t dpitch, intptr_t src, size_t wOffset, size_t hOffset, size_t width, size_t height, int kind, intptr_t stream)`

**cupy.cuda.runtime.memcpy2DToArray**

`cupy.cuda.runtime.memcpy2DToArray(intptr_t dst, size_t wOffset, size_t hOffset, intptr_t src, size_t spitch, size_t width, size_t height, int kind)`

**cupy.cuda.runtime.memcpy2DToArrayAsync**

`cupy.cuda.runtime.memcpy2DToArrayAsync(intptr_t dst, size_t wOffset, size_t hOffset, intptr_t src, size_t spitch, size_t width, size_t height, int kind, intptr_t stream)`

**cupy.cuda.runtime.memcpy3D**

`cupy.cuda.runtime.memcpy3D(intptr_t Memcpy3DParmsPtr)`

**cupy.cuda.runtime.memcpy3DAsync**

`cupy.cuda.runtime.memcpy3DAsync(intptr_t Memcpy3DParmsPtr, intptr_t stream)`

**cupy.cuda.runtime.memset**

`cupy.cuda.runtime.memset(intptr_t ptr, int value, size_t size)`

**cupy.cuda.runtime.memsetAsync**

`cupy.cuda.runtime.memsetAsync(intptr_t ptr, int value, size_t size, intptr_t stream)`

**cupy.cuda.runtime.memPrefetchAsync**

`cupy.cuda.runtime.memPrefetchAsync(intptr_t devPtr, size_t count, int dstDevice, intptr_t stream)`

**cupy.cuda.runtime.memAdvise**

`cupy.cuda.runtime.memAdvise(intptr_t devPtr, size_t count, int advice, int device)`

**cupy.cuda.runtime.pointerGetAttributes**

`cupy.cuda.runtime.pointerGetAttributes(intptr_t ptr) → PointerAttributes`

**cupy.cuda.runtime.streamCreate**

`cupy.cuda.runtime.streamCreate() → intptr_t`

**cupy.cuda.runtime.streamCreateWithFlags**

`cupy.cuda.runtime.streamCreateWithFlags(unsigned int flags) → intptr_t`

**cupy.cuda.runtime.streamDestroy**

`cupy.cuda.runtime.streamDestroy(intptr_t stream)`

**cupy.cuda.runtime.streamSynchronize**

`cupy.cuda.runtime.streamSynchronize(intptr_t stream)`

**cupy.cuda.runtime.streamAddCallback**

`cupy.cuda.runtime.streamAddCallback(intptr_t stream, callback, intptr_t arg, unsigned int flags=0)`

**cupy.cuda.runtime.streamQuery**

`cupy.cuda.runtime.streamQuery(intptr_t stream)`

**cupy.cuda.runtime.streamWaitEvent**

`cupy.cuda.runtime.streamWaitEvent(intptr_t stream, intptr_t event, unsigned int flags=0)`

**cupy.cuda.runtime.launchHostFunc**

`cupy.cuda.runtime.launchHostFunc(intptr_t stream, callback, intptr_t arg)`

**cupy.cuda.runtime.eventCreate**

`cupy.cuda.runtime.eventCreate()` → `intptr_t`

**cupy.cuda.runtime.eventCreateWithFlags**

`cupy.cuda.runtime.eventCreateWithFlags(unsigned int flags)` → `intptr_t`

**cupy.cuda.runtime.eventDestroy**

`cupy.cuda.runtime.eventDestroy(intptr_t event)`

**cupy.cuda.runtime.eventElapsedTime**

`cupy.cuda.runtime.eventElapsedTime(intptr_t start, intptr_t end)` → `float`

**cupy.cuda.runtime.eventQuery**

`cupy.cuda.runtime.eventQuery(intptr_t event)`

**cupy.cuda.runtime.eventRecord**

`cupy.cuda.runtime.eventRecord(intptr_t event, intptr_t stream)`

**cupy.cuda.runtime.eventSynchronize**

`cupy.cuda.runtime.eventSynchronize(intptr_t event)`

**cupy.cuda.runtime.ipcGetMemHandle**

`cupy.cuda.runtime.ipcGetMemHandle(intptr_t devPtr)`

**cupy.cuda.runtime.ipcOpenMemHandle**

`cupy.cuda.runtime.ipcOpenMemHandle(bytes handle, unsigned int flags=cudaIpcMemLazyEnablePeerAccess)`

**cupy.cuda.runtime.ipcCloseMemHandle**

```
cupy.cuda.runtime.ipcCloseMemHandle(intptr_t devPtr)
```

**cupy.cuda.runtime.ipcGetEventHandle**

```
cupy.cuda.runtime.ipcGetEventHandle(intptr_t event)
```

**cupy.cuda.runtime.ipcOpenEventHandle**

```
cupy.cuda.runtime.ipcOpenEventHandle(bytes handle)
```

## 5.7 Custom kernels

<code>cupy.ElementwiseKernel</code> (in_params, ..., [...])	User-defined elementwise kernel.
<code>cupy.ReductionKernel</code> (unicode in_params, ...)	User-defined reduction kernel.
<code>cupy.RawKernel</code> (unicode code, unicode name, ...)	User-defined custom kernel.
<code>cupy.RawModule</code> (unicode code=None, *, ..., [...])	User-defined custom module.
<code>cupy.fuse</code> (*args, **kwargs)	Decorator that fuses a function.

### 5.7.1 cupy.ElementwiseKernel

```
class cupy.ElementwiseKernel(in_params, out_params, operation, name='kernel', reduce_dims=True,
                             preamble="", no_return=False, return_tuple=False, **kwargs)
```

User-defined elementwise kernel.

This class can be used to define an elementwise kernel with or without broadcasting.

The kernel is compiled at an invocation of the `__call__()` method, which is cached for each device. The compiled binary is also cached into a file under the `$HOME/.cupy/kernel_cache/` directory with a hashed file name. The cached binary is reused by other processes.

**Parameters**

- **in\_params** (*str*) – Input argument list.
- **out\_params** (*str*) – Output argument list.
- **operation** (*str*) – The body in the loop written in CUDA-C/C++.
- **name** (*str*) – Name of the kernel function. It should be set for readability of the performance profiling.
- **reduce\_dims** (*bool*) – If `False`, the shapes of array arguments are kept within the kernel invocation. The shapes are reduced (i.e., the arrays are reshaped without copy to the minimum dimension) by default. It may make the kernel fast by reducing the index calculations.
- **options** (*tuple*) – Compile options passed to NVRTC. For details, see [https://docs.nvidia.com/cuda/nvrtc/index.html#group\\_\\_options](https://docs.nvidia.com/cuda/nvrtc/index.html#group__options).
- **preamble** (*str*) – Fragment of the CUDA-C/C++ code that is inserted at the top of the cu file.
- **no\_return** (*bool*) – If `True`, `__call__` returns `None`.

- **return\_tuple** (*bool*) – If True, `__call__` always returns tuple of array even if single value is returned.
- **loop\_prep** (*str*) – Fragment of the CUDA-C/C++ code that is inserted at the top of the kernel function definition and above the `for` loop.
- **after\_loop** (*str*) – Fragment of the CUDA-C/C++ code that is inserted at the bottom of the kernel function definition.

## Methods

### `__call__()`

Compiles and invokes the elementwise kernel.

The compilation runs only if the kernel is not cached. Note that the kernels with different argument dtypes or dimensions are not compatible. It means that single `ElementwiseKernel` object may be compiled into multiple kernel binaries.

#### Parameters

- **args** – Arguments of the kernel.
- **size** (*int*) – Range size of the indices. By default, the range size is automatically determined from the result of broadcasting. This parameter must be specified if and only if all ndarrays are *raw* and the range size cannot be determined automatically.
- **block\_size** (*int*) – Number of threads per block. By default, the value is set to 128.

**Returns** If `no_return` has not set, arrays are returned according to the `out_params` argument of the `__init__` method. If `no_return` has set, `None` is returned.

### `__eq__(value, /)`

Return `self==value`.

### `__ne__(value, /)`

Return `self!=value`.

### `__lt__(value, /)`

Return `self<value`.

### `__le__(value, /)`

Return `self<=value`.

### `__gt__(value, /)`

Return `self>value`.

### `__ge__(value, /)`

Return `self>=value`.

## Attributes

`in_params`

`kwargs`

`name`

`nargs`

`nin`

`no_return`



```

nout
operation
out_params
params
preamble
reduce_dims
return_tuple

```

## 5.7.2 cupy.ReductionKernel

```

class cupy.ReductionKernel(unicode in_params, unicode out_params, map_expr, reduce_expr,
                           post_map_expr, identity, name=u'reduce_kernel', reduce_type=None,
                           reduce_dims=True, preamble=u'', options=())

```

User-defined reduction kernel.

This class can be used to define a reduction kernel with or without broadcasting.

The kernel is compiled at an invocation of the `__call__()` method, which is cached for each device. The compiled binary is also cached into a file under the `$HOME/.cupy/kernel_cache/` directory with a hashed file name. The cached binary is reused by other processes.

### Parameters

- **in\_params** (*str*) – Input argument list.
- **out\_params** (*str*) – Output argument list.
- **map\_expr** (*str*) – Mapping expression for input values.
- **reduce\_expr** (*str*) – Reduction expression.
- **post\_map\_expr** (*str*) – Mapping expression for reduced values.
- **identity** (*str*) – Identity value for starting the reduction.
- **name** (*str*) – Name of the kernel function. It should be set for readability of the performance profiling.
- **reduce\_type** (*str*) – Type of values to be used for reduction. This type is used to store the special variables `a`.
- **reduce\_dims** (*bool*) – If True, input arrays are reshaped without copy to smaller dimensions for efficiency.
- **preamble** (*str*) – Fragment of the CUDA-C/C++ code that is inserted at the top of the cu file.
- **options** (*tuple of str*) – Additional compilation options.

## Methods

### `__call__()`

Compiles and invokes the reduction kernel.

The compilation runs only if the kernel is not cached. Note that the kernels with different argument dtypes, ndims, or axis are not compatible. It means that single ReductionKernel object may be compiled into multiple kernel binaries.

#### Parameters

- **args** – Arguments of the kernel.
- **out** (`cupy.ndarray`) – The output array. This can only be specified if args does not contain the output array.
- **axis** (*int or tuple of ints*) – Axis or axes along which the reduction is performed.
- **keepdims** (*bool*) – If True, the specified axes are remained as axes of length one.
- **stream** (`cupy.cuda.Stream`, *optional*) – The CUDA stream to launch the kernel on. If not given, the current stream will be used.

**Returns** Arrays are returned according to the `out_params` argument of the `__init__` method.

### `__eq__(value, /)`

Return self==value.

### `__ne__(value, /)`

Return self!=value.

### `__lt__(value, /)`

Return self<value.

### `__le__(value, /)`

Return self<=value.

### `__gt__(value, /)`

Return self>value.

### `__ge__(value, /)`

Return self>=value.

## Attributes

### `identity`

unicode

**Type** identity

### `in_params`

### `map_expr`

### `name`

### `nargs`

### `nin`

### `nout`

### `options`

### `out_params`

```

params
post_map_expr
preamble
reduce_dims
reduce_expr
reduce_type

```

### 5.7.3 cupy.RawKernel

```

class cupy.RawKernel(unicode code, unicode name, tuple options=(), unicode backend=u'nvrtc', bool
                      translate_cucomplex=False, *, bool enable_cooperative_groups=False, bool
                      jitify=False)

```

User-defined custom kernel.

This class can be used to define a custom kernel using raw CUDA source.

The kernel is compiled at an invocation of the `__call__()` method, which is cached for each device. The compiled binary is also cached into a file under the `$HOME/.cupy/kernel_cache/` directory with a hashed file name. The cached binary is reused by other processes.

#### Parameters

- **code** (*str*) – CUDA source code.
- **name** (*str*) – Name of the kernel function.
- **options** (*tuple of str*) – Compiler options passed to the backend (NVRTC or NVCC). For details, see [https://docs.nvidia.com/cuda/nvrtc/index.html#group\\_options](https://docs.nvidia.com/cuda/nvrtc/index.html#group_options) or <https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#command-option-description>
- **backend** (*str*) – Either *nvrtc* or *nvcc*. Defaults to *nvrtc*
- **translate\_cucomplex** (*bool*) – Whether the CUDA source includes the header *cuComplex.h* or not. If set to *True*, any code that uses the functions from *cuComplex.h* will be translated to its Thrust counterpart. Defaults to *False*.
- **enable\_cooperative\_groups** (*bool*) – Whether to enable cooperative groups in the CUDA source. If set to *True*, compile options are configured properly and the kernel is launched with `cuLaunchCooperativeKernel` so that cooperative groups can be used from the CUDA source. This feature is only supported in CUDA 9 or later.
- **jitify** (*bool*) – Whether or not to use *Jitify* to assist NVRTC to compile C++ kernels. Defaults to *False*.

#### Methods

```

__call__(self, grid, block, args, *, shared_mem=0)

```

Compiles and invokes the kernel.

The compilation runs only if the kernel is not cached.

#### Parameters

- **grid** (*tuple*) – Size of grid in blocks.
- **block** (*tuple*) – Dimensions of each thread block.

- **args** (*tuple*) – Arguments of the kernel.
- **shared\_mem** (*int*) – Dynamic shared-memory size per thread block in bytes.

**compile**(*self*, *log\_stream=None*)

Compile the current kernel.

In general, you don’t have to call this method; kernels are compiled implicitly on the first call.

**Parameters** **log\_stream** (*object*) – Pass either `sys.stdout` or a file object to which the compiler output will be written. Defaults to `None`.

**\_\_eq\_\_**(*value*, /)

Return `self==value`.

**\_\_ne\_\_**(*value*, /)

Return `self!=value`.

**\_\_lt\_\_**(*value*, /)

Return `self<value`.

**\_\_le\_\_**(*value*, /)

Return `self<=value`.

**\_\_gt\_\_**(*value*, /)

Return `self>value`.

**\_\_ge\_\_**(*value*, /)

Return `self>=value`.

## Attributes

### **attributes**

Returns a dictionary containing runtime kernel attributes. This is a read-only property; to overwrite the attributes, use

```
kernel = RawKernel(...) # arguments omitted
kernel.max_dynamic_shared_size_bytes = ...
kernel.preferred_shared_memory_carveout = ...
```

Note that the two attributes shown in the above example are the only two currently settable in CUDA.

Any attribute not existing in the present CUDA toolkit version will have the value -1.

**Returns** A dictionary containing the kernel’s attributes.

**Return type** `dict`

### **backend**

### **binary\_version**

The binary architecture version that was used during compilation, in the format: 10\*major + minor.

### **cache\_mode\_ca**

Indicates whether option “-Xptxas -dlcm=ca” was set during compilation.

### **code**

### **const\_size\_bytes**

The size in bytes of constant memory used by the function.

### **enable\_cooperative\_groups**

**file\_path****kernel****local\_size\_bytes**

The size in bytes of local memory used by the function.

**max\_dynamic\_shared\_size\_bytes**

The maximum dynamically-allocated shared memory size in bytes that can be used by the function. Can be set.

**max\_threads\_per\_block**

The maximum number of threads per block that can successfully launch the function on the device.

**name****num\_regs**

The number of registers used by the function.

**options****preferred\_shared\_memory\_carveout**On devices that have a unified L1 cache and shared memory, indicates the fraction to be used for shared memory as a *percentage* of the total. If the fraction does not exactly equal a supported shared memory capacity, then the next larger supported capacity is used. Can be set.**ptx\_version**

The PTX virtual architecture version that was used during compilation, in the format: 10\*major + minor.

**shared\_size\_bytes**

The size in bytes of the statically-allocated shared memory used by the function. This is separate from any dynamically-allocated shared memory, which must be specified when the function is called.

## 5.7.4 cupy.RawModule

```
class cupy.RawModule(unicode code=None, *, unicode path=None, tuple options=(), unicode backend=u'nvrtc',
                    bool translate_cucomplex=False, bool enable_cooperative_groups=False,
                    name_expressions=None, bool jitify=False)
```

User-defined custom module.

This class can be used to either compile raw CUDA sources or load CUDA modules (\*.cubin, \*.ptx). This class is useful when a number of CUDA kernels in the same source need to be retrieved.

For the former case, the CUDA source code is compiled when any method is called. For the latter case, an existing CUDA binary (\*.cubin) or a PTX file can be loaded by providing its path.

CUDA kernels in a *RawModule* can be retrieved by calling *get\_function()*, which will return an instance of *RawKernel*. (Same as in *RawKernel*, the generated binary is also cached.)

### Parameters

- **code** (*str*) – CUDA source code. Mutually exclusive with *path*.
- **path** (*str*) – Path to cubin/ptx. Mutually exclusive with *code*.
- **options** (*tuple of str*) – Compiler options passed to the backend (NVRTC or NVCC). For details, see [https://docs.nvidia.com/cuda/nvrtc/index.html#group\\_options](https://docs.nvidia.com/cuda/nvrtc/index.html#group_options) or <https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#command-option-description>.
- **backend** (*str*) – Either *nvrtc* or *nvcc*. Defaults to *nvrtc*.

- **translate\_cucomplex** (*bool*) – Whether the CUDA source includes the header *cuComplex.h* or not. If set to `True`, any code that uses the functions from *cuComplex.h* will be translated to its Thrust counterpart. Defaults to `False`.
- **enable\_cooperative\_groups** (*bool*) – Whether to enable cooperative groups in the CUDA source. If set to `True`, compile options are configured properly and the kernel is launched with `cuLaunchCooperativeKernel` so that cooperative groups can be used from the CUDA source. This feature is only supported in CUDA 9 or later.
- **name\_expressions** (*sequence of str*) – A sequence (e.g. list) of strings referring to the names of C++ global/template kernels. For example, `name_expressions=['func1<int>', 'func1<double>', 'func2']` for the template kernel `func1<T>` and non-template kernel `func2`. Strings in this tuple must then be passed, one at a time, to `get_function()` to retrieve the corresponding kernel.
- **jitify** (*bool*) – Whether or not to use `Jitify` to assist NVRTC to compile C++ kernels. Defaults to `False`.

---

**Note:** Each kernel in `RawModule` possesses independent function attributes.

---

---

**Note:** Before CuPy v8.0.0, the compilation happens at initialization. Now, it happens at the first time retrieving any object (kernels, pointers, or texrefs) from the module.

---

## Methods

**compile**(*self*, *log\_stream=None*)

Compile the current module.

In general, you don't have to call this method; kernels are compiled implicitly on the first call.

**Parameters** **log\_stream** (*object*) – Pass either `sys.stdout` or a file object to which the compiler output will be written. Defaults to `None`.

---

**Note:** Calling `compile()` will reset the internal state of a `RawKernel`.

---

**get\_function**(*self*, *unicode name*)

Retrieve a CUDA kernel by its name from the module.

**Parameters** **name** (*str*) – Name of the kernel function. For C++ global/template kernels, `name` refers to one of the name expressions specified when initializing the present `RawModule` instance.

**Returns** An `RawKernel` instance.

**Return type** `RawKernel`

---

**Note:** The following example shows how to retrieve one of the specialized C++ template kernels:

```
code = r'''
template<typename T>
__global__ void func(T* in_arr) { /* do something */ }
'''
```

(continues on next page)

(continued from previous page)

```

kers = ('func<int>', 'func<float>', 'func<double>')
mod = cupy.RawModule(code=code, options=('--std=c++11',),
                    name_expressions=kers)

// retrieve func<int>
ker_int = mod.get_function(kers[0])

```

**See also:**

`nVRTCAddNameExpression` and `nVRTCGetLoweredName` from [Accessing Lowered Names](#) of the NVRTC documentation.

**get\_global**(*self*, *name*)

Retrieve a pointer to a global symbol by its name from the module.

**Parameters** *name* (*str*) – Name of the global symbol.

**Returns** A handle to the global symbol.

**Return type** *MemoryPointer*

**Note:** This method can be used to access, for example, constant memory:

```

# to get a pointer to "arr" declared in the source like this:
# __constant__ float arr[10];
memptr = mod.get_global("arr")
# ...wrap it using cupy.ndarray with a known shape
arr_ndarray = cp.ndarray((10,), cp.float32, memptr)
# ...perform data transfer to initialize it
arr_ndarray[...] = cp.random.random((10,), dtype=cp.float32)
# ...and arr is ready to be accessed by RawKernels

```

**get\_texref**(*self*, *name*)

Retrieve a texture reference by its name from the module.

**Parameters** *name* (*str*) – Name of the texture reference.

**Returns** A CUtexref handle, to be passed to [TextureReference](#).

**Return type** `intptr_t`

**\_\_eq\_\_**(*value*, /)

Return `self==value`.

**\_\_ne\_\_**(*value*, /)

Return `self!=value`.

**\_\_lt\_\_**(*value*, /)

Return `self<value`.

**\_\_le\_\_**(*value*, /)

Return `self<=value`.

**\_\_gt\_\_**(*value*, /)

Return `self>value`.

```
__ge__(value, /)
    Return self>=value.
```

### Attributes

`backend`  
`code`  
`enable_cooperative_groups`  
`file_path`  
`module`  
`name_expressions`  
`options`

## 5.7.5 cupy.fuse

`cupy.fuse(*args, **kwargs)`  
Decorator that fuses a function.

This decorator can be used to define an elementwise or reduction kernel more easily than [ElementwiseKernel](#) or [ReductionKernel](#).

Since the fused kernels are cached and reused, it is recommended to reuse the same decorated functions instead of e.g. decorating local functions that are defined multiple times.

**Parameters** `kernel_name` (*str*) – Name of the fused kernel function. If omitted, the name of the decorated function is used.

### Example

```
>>> @cupy.fuse(kernel_name='squared_diff')
... def squared_diff(x, y):
...     return (x - y) * (x - y)
...
>>> x = cupy.arange(10)
>>> y = cupy.arange(10)[::-1]
>>> squared_diff(x, y)
array([81, 49, 25,  9,  1,  1,  9, 25, 49, 81])
```

## 5.7.6 JIT kernel definition

<code>cupyx.jit.rawkernel([mode])</code>	A decorator compiles a Python function into CUDA kernel.
<code>cupyx.jit.threadIdx</code>	dim3 threadIdx
<code>cupyx.jit.blockDim</code>	dim3 blockDim
<code>cupyx.jit.blockIdx</code>	dim3 blockIdx
<code>cupyx.jit.gridDim</code>	dim3 gridDim

continues on next page



Table 137 – continued from previous page

<code>cupyx.jit.syncthreads</code>	Calls <code>__syncthreads()</code>
<code>cupyx.jit.shared_memory</code>	Allocates shared memory and returns the 1-dim array.
<code>cupyx.jit._interface._JitRawKernel</code> (func, mode)	JIT CUDA kernel object.

## cupyx.jit.rawkernel

`cupyx.jit.rawkernel(mode='cuda')`

A decorator compiles a Python function into CUDA kernel.

## cupyx.jit.threadIdx

`cupyx.jit.threadIdx = <cupyx.jit._interface._Dim3 object>`

dim3 threadIdx

A namedtuple of three integers represents threadIdx.

### Variables

- `x(uint32)` – threadIdx.x
- `y(uint32)` – threadIdx.y
- `z(uint32)` – threadIdx.z

## cupyx.jit.blockDim

`cupyx.jit.blockDim = <cupyx.jit._interface._Dim3 object>`

dim3 blockDim

A namedtuple of three integers represents blockDim.

### Variables

- `x(uint32)` – blockDim.x
- `y(uint32)` – blockDim.y
- `z(uint32)` – blockDim.z

## cupyx.jit.blockIdx

`cupyx.jit.blockIdx = <cupyx.jit._interface._Dim3 object>`

dim3 blockIdx

A namedtuple of three integers represents blockIdx.

### Variables

- `x(uint32)` – blockIdx.x
- `y(uint32)` – blockIdx.y
- `z(uint32)` – blockIdx.z

### cupyx.jit.gridDim

`cupyx.jit.gridDim = <cupyx.jit._interface._Dim3 object>`  
dim3 gridDim

A namedtuple of three integers represents gridDim.

#### Variables

- **x** (*uint32*) – gridDim.x
- **y** (*uint32*) – gridDim.y
- **z** (*uint32*) – gridDim.z

### cupyx.jit.syncthreads

`cupyx.jit.syncthreads = <cupyx.jit._builtin_funcs.SyncThreads object>`  
Calls `__syncthreads()`

### cupyx.jit.shared\_memory

`cupyx.jit.shared_memory = <cupyx.jit._builtin_funcs.SharedMemory object>`  
Allocates shared memory and returns the 1-dim array.

#### Parameters

- **dtype** (*dtype*) – The dtype of the returned array.
- **size** (*int* or *None*) – If *int* type, the size of static shared memory. If *None*, declares the shared memory with extern specifier.

### cupyx.jit.\_interface.\_JitRawKernel

`class cupyx.jit._interface._JitRawKernel(func, mode)`  
JIT CUDA kernel object.

The decorator `:func:cupyx.jit.rawkernel` converts the target function to an object of this class. This class is not intended to be instantiated by users.

#### Methods

`__call__(grid, block, args, shared_mem=0, stream=None, enable_cooperative_groups=False)`  
Calls the CUDA kernel.

The compilation will be deferred until the first function call. CuPy's JIT compiler infers the types of arguments at the call time, and will cache the compiled kernels for speeding up any subsequent calls.

#### Parameters

- **grid** (*tuple of int*) – Size of grid in blocks.
- **block** (*tuple of int*) – Dimensions of each thread block.
- **args** (*tuple*) – Arguments of the kernel. The type of all elements must be *bool*, *int*, *float*, *complex*, NumPy scalar or *cupy.ndarray*.
- **shared\_mem** (*int*) – Dynamic shared-memory size per thread block in bytes.

- **stream** (`cupy.cuda.Stream`) – CUDA stream.

**See also:**

*JIT kernel definition*

**\_\_getitem\_\_** (*grid\_and\_block*)

Numba-style kernel call.

**See also:**

*JIT kernel definition*

**\_\_eq\_\_** (*value, /*)

Return self==value.

**\_\_ne\_\_** (*value, /*)

Return self!=value.

**\_\_lt\_\_** (*value, /*)

Return self<value.

**\_\_le\_\_** (*value, /*)

Return self<=value.

**\_\_gt\_\_** (*value, /*)

Return self>value.

**\_\_ge\_\_** (*value, /*)

Return self>=value.

## Attributes

### **cached\_code**

Returns `next(iter(self.cached_codes.values()))`.

This property method is for debugging purpose. The return value is not guaranteed to keep backward compatibility.

### **cached\_codes**

Returns a dict that has input types as keys and codes values.

This property method is for debugging purpose. The return value is not guaranteed to keep backward compatibility.

## 5.7.7 Kernel binary memoization

<code>cupy.memoize</code> (bool for_each_device=False)	Makes a function memoizing the result for each argument and device.
<code>cupy.clear_memo</code> ()	Clears the memoized results for all functions decorated by memoize.

## cupy.memoize

`cupy.memoize(bool for_each_device=False)`

Makes a function memoizing the result for each argument and device.

This decorator provides automatic memoization of the function result.

**Parameters** `for_each_device` (*bool*) – If True, it memoizes the results for each device. Otherwise, it memoizes the results only based on the arguments.

## cupy.clear\_memo

`cupy.clear_memo()`

Clears the memoized results for all functions decorated by memoize.

# 5.8 Profiling

## 5.8.1 Time range

<code>cupy.prof.TimeRangeDecorator([message, ...])</code>	Decorator to mark function calls with range in NVIDIA profiler
<code>cupy.prof.time_range(message[, color_id, ...])</code>	A context manager to describe the enclosed block as a nested range

## cupy.prof.TimeRangeDecorator

**class** `cupy.prof.TimeRangeDecorator(message=None, color_id=None, argb_color=None, sync=False)`

Decorator to mark function calls with range in NVIDIA profiler

Decorated function calls are marked as ranges in NVIDIA profiler timeline.

```
>>> from cupy import prof
>>> @cupy.prof.TimeRangeDecorator()
... def function_to_profile():
...     pass
```

### Parameters

- **message** (*str*) – Name of a range, default use `func.__name__`.
- **color\_id** – range color ID
- **argb\_color** – range color in ARGB (e.g. 0xFF00FF00 for green)
- **sync** (*bool*) – If True, waits for completion of all outstanding processing on GPU before calling `cupy.cuda.nvtx.RangePush()` or `cupy.cuda.nvtx.RangePop()`

See also:

`cupy.cuda.nvtx.RangePush()` `cupy.cuda.nvtx.RangePop()`

## Methods

```

__call__(func)
    Call self as a function.

__enter__()

__exit__(exc_type, exc_value, traceback)

__eq__(value, /)
    Return self==value.

__ne__(value, /)
    Return self!=value.

__lt__(value, /)
    Return self<value.

__le__(value, /)
    Return self<=value.

__gt__(value, /)
    Return self>value.

__ge__(value, /)
    Return self>=value.

```

## cupy.prof.time\_range

`cupy.prof.time_range(message, color_id=None, argb_color=None, sync=False)`

A context manager to describe the enclosed block as a nested range

```

>>> from cupy import prof
>>> with cupy.prof.time_range('some range in green', color_id=0):
...     # do something you want to measure
...     pass

```

### Parameters

- **message** – Name of a range.
- **color\_id** – range color ID
- **argb\_color** – range color in ARGB (e.g. 0xFF00FF00 for green)
- **sync** (*bool*) – If True, waits for completion of all outstanding processing on GPU before calling `cupy.cuda.nvtx.RangePush()` or `cupy.cuda.nvtx.RangePop()`

See also:

`cupy.cuda.nvtx.RangePush()` `cupy.cuda.nvtx.RangePop()`

## 5.8.2 Timing helper

---

<code>cupyx.time.repeat(func[, args, kwargs, ...])</code>	Timing utility for measuring time spent by both CPU and GPU.
---	--

---

### cupyx.time.repeat

`cupyx.time.repeat(func, args=(), kwargs={}, n_repeat=10000, *, name=None, n_warmup=10, max_duration=inf, devices=None)`

Timing utility for measuring time spent by both CPU and GPU.

This function is a very convenient helper for setting up a timing test. The GPU time is properly recorded by synchronizing internal streams. As a result, to time a multi-GPU function all participating devices must be passed as the `devices` argument so that this helper knows which devices to record. A simple example is given as follows:

```
import cupy as cp
from cupyx.time import repeat

def f(a, b):
    return 3 * cp.sin(-a) * b

a = 0.5 - cp.random.random((100,))
b = cp.random.random((100,))
print(repeat(f, (a, b), n_repeat=1000))
```

#### Parameters

- **func** (*callable*) – a callable object to be timed.
- **args** (*tuple*) – positional arguments to be passed to the callable.
- **kwargs** (*dict*) – keyword arguments to be passed to the callable.
- **n\_repeat** (*int*) – number of times the callable is called. Increasing this value would improve the collected statistics at the cost of longer test time.
- **name** (*str*) – the function name to be reported. If not given, the callable’s `__name__` attribute is used.
- **n\_warmup** (*int*) – number of times the callable is called. The warm-up runs are not timed.
- **max\_duration** (*float*) – the maximum time (in seconds) that the entire test can use. If the taken time is longer than this limit, the test is stopped and the statistics collected up to the breakpoint is reported.
- **devices** (*tuple*) – a tuple of device IDs (int) that will be timed during the timing test. If not given, the current device is used.

**Returns** an object collecting all test results.

**Return type** `_PerfCaseResult`

**Warning:** This API is currently experimental and subject to change in future releases.

### 5.8.3 Device synchronization detection

<code>cupyx.allow_synchronize(allow)</code>	Allows or disallows device synchronization temporarily in the current thread.
<code>cupyx.DeviceSynchronized([message])</code>	Raised when device synchronization is detected while disallowed.

#### `cupyx.allow_synchronize`

`cupyx.allow_synchronize(allow)`

Allows or disallows device synchronization temporarily in the current thread.

If device synchronization is detected, `cupyx.DeviceSynchronized` will be raised.

Note that there can be false negatives and positives. Device synchronization outside CuPy will not be detected.

#### `cupyx.DeviceSynchronized`

**exception** `cupyx.DeviceSynchronized(message=None)`

Raised when device synchronization is detected while disallowed.

See also:

`cupyx.allow_synchronize()`

## 5.9 Environment variables

### 5.9.1 For runtime

Here are the environment variables that CuPy uses at runtime.

#### `CUDA_PATH`

Path to the directory containing CUDA. The parent of the directory containing `nvcc` is used as default. When `nvcc` is not found, `/usr/local/cuda` is used. See [Working with Custom CUDA Installation](#) for details.

#### `CUPY_CACHE_DIR`

Default: `${HOME}/.cupy/kernel_cache`

Path to the directory to store kernel cache. See [Overview](#) for details.

#### `CUPY_CACHE_SAVE_CUDA_SOURCE`

Default: `0`

If set to 1, CUDA source file will be saved along with compiled binary in the cache directory for debug purpose.

Note: the source file will not be saved if the compiled binary is already stored in the cache.

#### `CUPY_CACHE_IN_MEMORY`

Default: `0`

If set to 1, `CUPY_CACHE_DIR` and `CUPY_CACHE_SAVE_CUDA_SOURCE` will be ignored, and the cache is in memory. This environment variable allows reducing disk I/O, but is ignored when `nvcc` is set to be the compiler backend.

**CUPY\_DUMP\_CUDA\_SOURCE\_ON\_ERROR**

Default: 0

If set to 1, when CUDA kernel compilation fails, CuPy dumps CUDA kernel code to standard error.

**CUPY\_CUDA\_COMPILE\_WITH\_DEBUG**

Default: 0

If set to 1, CUDA kernel will be compiled with debug information (`--device-debug` and `--generate-line-info`).

**CUPY\_GPU\_MEMORY\_LIMIT**

Default: 0 (unlimited)

The amount of memory that can be allocated for each device. The value can be specified in absolute bytes or fraction (e.g., "90%") of the total memory of each GPU. See [Memory Management](#) for details.

**CUPY\_SEED**

Set the seed for random number generators.

**CUPY\_EXPERIMENTAL\_SLICE\_COPY**

Default: 0

If set to 1, the following syntax is enabled:

```
cupy_ndarray[:] = numpy_ndarray
```

**CUPY\_ACCELERATORS**

Default: "" (no accelerators)

A comma-separated string of backend names (cub or cutensor) which indicates the acceleration backends used in CuPy operations and its priority. All accelerators are disabled by default.

**CUPY\_TF32**

Default: 0

If set to 1, it allows CUDA libraries to use Tensor Cores TF32 compute for 32-bit floating point compute.

**CUPY\_CUDA\_ARRAY\_INTERFACE\_SYNC**

Default: 1

This controls CuPy's behavior as a Consumer. If set to 0, a stream synchronization will *not* be performed when a device array provided by an external library that implements the CUDA Array Interface is being consumed by CuPy. For more detail, see the [Synchronization](#) requirement in the CUDA Array Interface v3 documentation.

**CUPY\_CUDA\_ARRAY\_INTERFACE\_EXPORT\_VERSION**

Default: 3

This controls CuPy's behavior as a Producer. If set to 2, the CuPy stream on which the data is being operated will not be exported and thus the Consumer (another library) will not perform any stream synchronization. For more detail, see the [Synchronization](#) requirement in the CUDA Array Interface v3 documentation.

**NVCC**

Default: nvcc

Define the compiler to use when compiling CUDA source. Note that most CuPy kernels are built with NVRTC; this environment variable is only effective for [RawKernel/RawModule](#) with the nvcc backend or when using cub as the accelerator.

**CUPY\_CUDA\_PER\_THREAD\_DEFAULT\_STREAM**

Default: 0

If set to 1, CuPy will use the CUDA per-thread default stream, effectively causing each host thread to automatically execute in its own stream, unless the CUDA default (null) stream or a user-created stream is specified.



If set to 0 (default), the CUDA default (null) stream is used, unless the per-thread default stream (ptds) or a user-created stream is specified.

#### CUPY\_COMPILE\_WITH\_PTX

Default: 0

By default, CuPy directly compiles kernels into SASS (CUBIN) to support [CUDA Enhanced Compatibility](#). If set to 1, CuPy instead compiles kernels into PTX and lets CUDA Driver assemble SASS from PTX. This option is only effective for CUDA 11.1 or later; CuPy always compiles into PTX on earlier CUDA versions. Also, this option only applies when NVRTC is selected as the compilation backend. NVCC backend always compiles into SASS (CUBIN).

**CUDA Toolkit Environment Variables** In addition to the environment variables listed above, as in any CUDA programs, all of the CUDA environment variables listed in the [CUDA Toolkit Documentation](#) will also be honored.

---

**Note:** When [CUPY\\_ACCELERATORS](#) or [NVCC](#) environment variables are set, g++-6 or later is required as the runtime host compiler. Please refer to [Installing CuPy from Source](#) for the details on how to install g++.

---

## 5.9.2 For installation

These environment variables are used during installation (building CuPy from source).

#### CUTENSOR\_PATH

Path to the cuTENSOR root directory that contains lib and include directories. (experimental)

#### CUPY\_INSTALL\_USE\_HIP

Default: 0

Build CuPy for AMD ROCm Platform (experimental). For building the ROCm support, see [Installing Binary Packages](#) for further detail.

#### CUPY\_NVCC\_GENERATE\_CODE

Build CuPy for a particular CUDA architecture. For example:

```
CUPY_NVCC_GENERATE_CODE="arch=compute_60,code=sm_60"
```

For specifying multiple archs, concatenate the arch=... strings with semicolons (;). If current is specified, then it will automatically detect the currently installed GPU architectures in build time. When this is not set, the default is to support all architectures.

#### CUPY\_NUM\_BUILD\_JOBS

Default: 4

To enable or disable parallel build, sets the number of processes used to build the extensions in parallel.

#### CUPY\_NUM\_NVCC\_THREADS

Default: 2

To enable or disable nvcc parallel compilation, sets the number of threads used to compile files using nvcc.

Additionally, the environment variables [CUDA\\_PATH](#) and [NVCC](#) are also respected at build time.

## 5.10 Comparison Table

Here is a list of NumPy / SciPy APIs and its corresponding CuPy implementations.

- in CuPy column denotes that CuPy implementation is not provided yet. We welcome contributions for these functions.

### 5.10.1 NumPy / CuPy APIs

#### Module-Level

NumPy	CuPy
<code>numpy.abs</code>	<code>cupy.abs</code>
<code>numpy.absolute</code>	<code>cupy.absolute</code>
<code>numpy.add</code>	<code>cupy.add</code>
<code>numpy.add_docstring</code>	-
<code>numpy.add_newdoc</code>	-
<code>numpy.add_newdoc_ufunc</code>	-
<code>numpy.alen</code>	-
<code>numpy.all</code>	<code>cupy.all</code>
<code>numpy.allclose</code>	<code>cupy.allclose</code>
<code>numpy.alltrue</code>	-
<code>numpy.amax</code>	<code>cupy.amax</code>
<code>numpy.amin</code>	<code>cupy.amin</code>
<code>numpy.angle</code>	<code>cupy.angle</code>
<code>numpy.any</code>	<code>cupy.any</code>
<code>numpy.append</code>	<code>cupy.append</code>
<code>numpy.apply_along_axis</code>	<code>cupy.apply_along_axis</code>
<code>numpy.apply_over_axes</code>	-
<code>numpy.arange</code>	<code>cupy.arange</code>
<code>numpy.arccos</code>	<code>cupy.arccos</code>
<code>numpy.arccosh</code>	<code>cupy.arccosh</code>
<code>numpy.arcsin</code>	<code>cupy.arcsin</code>
<code>numpy.arcsinh</code>	<code>cupy.arcsinh</code>
<code>numpy.arctan</code>	<code>cupy.arctan</code>
<code>numpy.arctan2</code>	<code>cupy.arctan2</code>
<code>numpy.arctanh</code>	<code>cupy.arctanh</code>
<code>numpy.argmax</code>	<code>cupy.argmax</code>
<code>numpy.argmin</code>	<code>cupy.argmin</code>
<code>numpy.argpartition</code>	<code>cupy.argpartition</code>
<code>numpy.argsort</code>	<code>cupy.argsort</code>
<code>numpy.argwhere</code>	<code>cupy.argwhere</code>
<code>numpy.around</code>	<code>cupy.around</code>
<code>numpy.array</code>	<code>cupy.array</code>
<code>numpy.array2string</code>	-
<code>numpy.array_equal</code>	<code>cupy.array_equal</code>
<code>numpy.array_equiv</code>	-
<code>numpy.array_repr</code>	<code>cupy.array_repr</code>
<code>numpy.array_split</code>	<code>cupy.array_split</code>
<code>numpy.array_str</code>	<code>cupy.array_str</code>
<code>numpy.asanyarray</code>	<code>cupy.asanyarray</code>

continues on next page

Table 142 – continued from previous page

NumPy	CuPy
<code>numpy.asarray</code>	<code>cupy.asarray</code>
<code>numpy.asarray_chkfinite</code>	-
<code>numpy.ascontiguousarray</code>	<code>cupy.ascontiguousarray</code>
<code>numpy.asfarray</code>	-
<code>numpy.asfortranarray</code>	<code>cupy.asfortranarray</code>
<code>numpy.asmatrix</code>	-
<code>numpy.asscalar</code>	-
<code>numpy.atleast_1d</code>	<code>cupy.atleast_1d</code>
<code>numpy.atleast_2d</code>	<code>cupy.atleast_2d</code>
<code>numpy.atleast_3d</code>	<code>cupy.atleast_3d</code>
<code>numpy.average</code>	<code>cupy.average</code>
<code>numpy.bartlett</code>	<code>cupy.bartlett</code>
<code>numpy.base_repr</code>	<code>cupy.base_repr</code>
<code>numpy.binary_repr</code>	<code>cupy.binary_repr</code>
<code>numpy.bincount</code>	<code>cupy.bincount</code>
<code>numpy.bitwise_and</code>	<code>cupy.bitwise_and</code>
<code>numpy.bitwise_not</code>	<code>cupy.bitwise_not</code>
<code>numpy.bitwise_or</code>	<code>cupy.bitwise_or</code>
<code>numpy.bitwise_xor</code>	<code>cupy.bitwise_xor</code>
<code>numpy.blackman</code>	<code>cupy.blackman</code>
<code>numpy.block</code>	-
<code>numpy.bmat</code>	-
<code>numpy.broadcast_arrays</code>	<code>cupy.broadcast_arrays</code>
<code>numpy.broadcast_shapes</code>	-
<code>numpy.broadcast_to</code>	<code>cupy.broadcast_to</code>
<code>numpy.busday_count</code>	-
<code>numpy.busday_offset</code>	-
<code>numpy.byte_bounds</code>	-
<code>numpy.can_cast</code>	<code>cupy.can_cast</code>
<code>numpy.cbrt</code>	<code>cupy.cbrt</code>
<code>numpy.ceil</code>	<code>cupy.ceil</code>
<code>numpy.choose</code>	<code>cupy.choose</code>
<code>numpy.clip</code>	<code>cupy.clip</code>
<code>numpy.column_stack</code>	<code>cupy.column_stack</code>
<code>numpy.common_type</code>	<code>cupy.common_type</code>
<code>numpy.compare_chararrays</code>	-
<code>numpy.compress</code>	<code>cupy.compress</code>
<code>numpy.concatenate</code>	<code>cupy.concatenate</code>
<code>numpy.conj</code>	<code>cupy.conj</code>
<code>numpy.conjugate</code>	<code>cupy.conjugate</code>
<code>numpy.convolve</code>	<code>cupy.convolve</code>
<code>numpy.copy</code>	<code>cupy.copy</code>
<code>numpy.copysign</code>	<code>cupy.copysign</code>
<code>numpy.copysign</code>	<code>cupy.copysign</code>
<code>numpy.corrcoef</code>	<code>cupy.corrcoef</code>
<code>numpy.correlate</code>	<code>cupy.correlate</code>
<code>numpy.cos</code>	<code>cupy.cos</code>
<code>numpy.cosh</code>	<code>cupy.cosh</code>
<code>numpy.count_nonzero</code>	<code>cupy.count_nonzero</code>

continues on next page

Table 142 – continued from previous page

NumPy	CuPy
<code>numpy.cov</code>	<code>cupy.cov</code>
<code>numpy.cross</code>	<code>cupy.cross</code>
<code>numpy.cumprod</code>	<code>cupy.cumprod</code>
<code>numpy.cumproduct</code>	-
<code>numpy.cumsum</code>	<code>cupy.cumsum</code>
<code>numpy.datetime_as_string</code>	-
<code>numpy.datetime_data</code>	-
<code>numpy.deg2rad</code>	<code>cupy.deg2rad</code>
<code>numpy.degrees</code>	<code>cupy.degrees</code>
<code>numpy.delete</code>	-
<code>numpy.deprecate</code>	-
<code>numpy.deprecate_with_doc</code>	-
<code>numpy.diag</code>	<code>cupy.diag</code>
<code>numpy.diag_indices</code>	<code>cupy.diag_indices</code>
<code>numpy.diag_indices_from</code>	<code>cupy.diag_indices_from</code>
<code>numpy.diagflat</code>	<code>cupy.diagflat</code>
<code>numpy.diagonal</code>	<code>cupy.diagonal</code>
<code>numpy.diff</code>	<code>cupy.diff</code>
<code>numpy.digitize</code>	<code>cupy.digitize</code>
<code>numpy.disp</code>	-
<code>numpy.divide</code>	<code>cupy.divide</code>
<code>numpy.divmod</code>	<code>cupy.divmod</code>
<code>numpy.dot</code>	<code>cupy.dot</code>
<code>numpy.dsplitt</code>	<code>cupy.dsplitt</code>
<code>numpy.dstack</code>	<code>cupy.dstack</code>
<code>numpy.ediff1d</code>	-
<code>numpy.einsum</code>	<code>cupy.einsum</code>
<code>numpy.einsum_path</code>	-
<code>numpy.empty</code>	<code>cupy.empty</code>
<code>numpy.empty_like</code>	<code>cupy.empty_like</code>
<code>numpy.equal</code>	<code>cupy.equal</code>
<code>numpy.exp</code>	<code>cupy.exp</code>
<code>numpy.exp2</code>	<code>cupy.exp2</code>
<code>numpy.expand_dims</code>	<code>cupy.expand_dims</code>
<code>numpy.expm1</code>	<code>cupy.expm1</code>
<code>numpy.extract</code>	<code>cupy.extract</code>
<code>numpy.eye</code>	<code>cupy.eye</code>
<code>numpy.fabs</code>	-
<code>numpy.fastCopyAndTranspose</code>	-
<code>numpy.fill_diagonal</code>	<code>cupy.fill_diagonal</code>
<code>numpy.find_common_type</code>	<code>cupy.find_common_type</code> ( <i>alias of</i> <code>numpy.find_common_type</code> )
<code>numpy.fix</code>	<code>cupy.fix</code>
<code>numpy.flatnonzero</code>	<code>cupy.flatnonzero</code>
<code>numpy.flip</code>	<code>cupy.flip</code>
<code>numpy.fliplr</code>	<code>cupy.fliplr</code>
<code>numpy.flipud</code>	<code>cupy.flipud</code>
<code>numpy.float_power</code>	-
<code>numpy.floor</code>	<code>cupy.floor</code>
<code>numpy.floor_divide</code>	<code>cupy.floor_divide</code>

continues on next page

Table 142 – continued from previous page

NumPy	CuPy
<code>numpy.fmax</code>	<code>cupy.fmax</code>
<code>numpy.fmin</code>	<code>cupy.fmin</code>
<code>numpy.fmod</code>	<code>cupy.fmod</code>
<code>numpy.format_float_positional</code>	-
<code>numpy.format_float_scientific</code>	-
<code>numpy.frexp</code>	<code>cupy.frexp</code>
<code>numpy.frombuffer</code>	-
<code>numpy.fromfile</code>	<code>cupy.fromfile</code>
<code>numpy.fromfunction</code>	-
<code>numpy.fromiter</code>	-
<code>numpy.frompyfunc</code>	-
<code>numpy.fromregex</code>	-
<code>numpy.fromstring</code>	-
<code>numpy.full</code>	<code>cupy.full</code>
<code>numpy.full_like</code>	<code>cupy.full_like</code>
<code>numpy.gcd</code>	<code>cupy.gcd</code>
<code>numpy.genfromtxt</code>	-
<code>numpy.geomspace</code>	-
<code>numpy.get_array_wrap</code>	-
<code>numpy.get_include</code>	-
<code>numpy.get_printoptions</code>	-
<code>numpy.getbufsize</code>	-
<code>numpy.geterr</code>	-
<code>numpy.geterrcall</code>	-
<code>numpy.geterrobj</code>	-
<code>numpy.gradient</code>	<code>cupy.gradient</code>
<code>numpy.greater</code>	<code>cupy.greater</code>
<code>numpy.greater_equal</code>	<code>cupy.greater_equal</code>
<code>numpy.hamming</code>	<code>cupy.hamming</code>
<code>numpy.hanning</code>	<code>cupy.hanning</code>
<code>numpy.heaviside</code>	-
<code>numpy.histogram</code>	<code>cupy.histogram</code>
<code>numpy.histogram2d</code>	<code>cupy.histogram2d</code>
<code>numpy.histogram_bin_edges</code>	-
<code>numpy.histogramdd</code>	<code>cupy.histogramdd</code>
<code>numpy.hsplitt</code>	<code>cupy.hsplitt</code>
<code>numpy.hstack</code>	<code>cupy.hstack</code>
<code>numpy.hypot</code>	<code>cupy.hypot</code>
<code>numpy.i0</code>	<code>cupy.i0</code>
<code>numpy.identity</code>	<code>cupy.identity</code>
<code>numpy.imag</code>	<code>cupy.imag</code>
<code>numpy.in1d</code>	<code>cupy.in1d</code>
<code>numpy.indices</code>	<code>cupy.indices</code>
<code>numpy.info</code>	-
<code>numpy.inner</code>	<code>cupy.inner</code>
<code>numpy.insert</code>	-
<code>numpy.interp</code>	<code>cupy.interp</code>
<code>numpy.intersect1d</code>	-
<code>numpy.invert</code>	<code>cupy.invert</code>

continues on next page

Table 142 – continued from previous page

NumPy	CuPy
<code>numpy.is_busday</code>	-
<code>numpy.isclose</code>	<code>cupy.isclose</code>
<code>numpy.iscomplex</code>	<code>cupy.iscomplex</code>
<code>numpy.iscomplexobj</code>	<code>cupy.iscomplexobj</code>
<code>numpy.isfinite</code>	<code>cupy.isfinite</code>
<code>numpy.isfortran</code>	<code>cupy.isfortran</code>
<code>numpy.isin</code>	<code>cupy.isin</code>
<code>numpy.isinf</code>	<code>cupy.isinf</code>
<code>numpy.isnan</code>	<code>cupy.isnan</code>
<code>numpy.isnat</code>	-
<code>numpy.isneginf</code>	-
<code>numpy.isposinf</code>	-
<code>numpy.isreal</code>	<code>cupy.isreal</code>
<code>numpy.isrealobj</code>	<code>cupy.isrealobj</code>
<code>numpy.isscalar</code>	<code>cupy.isscalar</code>
<code>numpy.issctype</code>	<code>cupy.issctype</code> ( <i>alias of</i> <code>numpy.issctype</code> )
<code>numpy.issubclass_</code>	<code>cupy.issubclass_</code> ( <i>alias of</i> <code>numpy.issubclass_</code> )
<code>numpy.issubdtype</code>	<code>cupy.issubdtype</code> ( <i>alias of</i> <code>numpy.issubdtype</code> )
<code>numpy.issubdtype</code>	<code>cupy.issubdtype</code> ( <i>alias of</i> <code>numpy.issubdtype</code> )
<code>numpy.iterable</code>	-
<code>numpy.ix_</code>	<code>cupy.ix_</code>
<code>numpy.kaiser</code>	<code>cupy.kaiser</code>
<code>numpy.kron</code>	<code>cupy.kron</code>
<code>numpy.lcm</code>	<code>cupy.lcm</code>
<code>numpy.ldexp</code>	<code>cupy.ldexp</code>
<code>numpy.left_shift</code>	<code>cupy.left_shift</code>
<code>numpy.less</code>	<code>cupy.less</code>
<code>numpy.less_equal</code>	<code>cupy.less_equal</code>
<code>numpy.lexsort</code>	<code>cupy.lexsort</code>
<code>numpy.linspace</code>	<code>cupy.linspace</code>
<code>numpy.load</code>	<code>cupy.load</code>
<code>numpy.loads</code>	-
<code>numpy.loadtxt</code>	-
<code>numpy.log</code>	<code>cupy.log</code>
<code>numpy.log10</code>	<code>cupy.log10</code>
<code>numpy.log1p</code>	<code>cupy.log1p</code>
<code>numpy.log2</code>	<code>cupy.log2</code>
<code>numpy.logaddexp</code>	<code>cupy.logaddexp</code>
<code>numpy.logaddexp2</code>	<code>cupy.logaddexp2</code>
<code>numpy.logical_and</code>	<code>cupy.logical_and</code>
<code>numpy.logical_not</code>	<code>cupy.logical_not</code>
<code>numpy.logical_or</code>	<code>cupy.logical_or</code>
<code>numpy.logical_xor</code>	<code>cupy.logical_xor</code>
<code>numpy.logspace</code>	<code>cupy.logspace</code>
<code>numpy.lookfor</code>	-
<code>numpy.mafromtxt</code>	-
<code>numpy.mask_indices</code>	-
<code>numpy.mat</code>	-
<code>numpy.matmul</code>	<code>cupy.matmul</code>

continues on next page

Table 142 – continued from previous page

NumPy	CuPy
<code>numpy.max</code>	<code>cupy.max</code>
<code>numpy.maximum</code>	<code>cupy.maximum</code>
<code>numpy.maximum_sctype</code>	-
<code>numpy.may_share_memory</code>	<code>cupy.may_share_memory</code>
<code>numpy.mean</code>	<code>cupy.mean</code>
<code>numpy.median</code>	<code>cupy.median</code>
<code>numpy.meshgrid</code>	<code>cupy.meshgrid</code>
<code>numpy.min</code>	<code>cupy.min</code>
<code>numpy.min_scalar_type</code>	<code>cupy.min_scalar_type</code> ( <i>alias of</i> <code>numpy.min_scalar_type</code> )
<code>numpy.minimum</code>	<code>cupy.minimum</code>
<code>numpy.mintypecode</code>	<code>cupy.mintypecode</code> ( <i>alias of</i> <code>numpy.mintypecode</code> )
<code>numpy.mod</code>	<code>cupy.mod</code>
<code>numpy.modf</code>	<code>cupy.modf</code>
<code>numpy.moveaxis</code>	<code>cupy.moveaxis</code>
<code>numpy.msort</code>	<code>cupy.msort</code>
<code>numpy.multiply</code>	<code>cupy.multiply</code>
<code>numpy.nan_to_num</code>	<code>cupy.nan_to_num</code>
<code>numpy.nanargmax</code>	<code>cupy.nanargmax</code>
<code>numpy.nanargmin</code>	<code>cupy.nanargmin</code>
<code>numpy.nancumprod</code>	<code>cupy.nancumprod</code>
<code>numpy.nancumsum</code>	<code>cupy.nancumsum</code>
<code>numpy.nanmax</code>	<code>cupy.nanmax</code>
<code>numpy.nanmean</code>	<code>cupy.nanmean</code>
<code>numpy.nanmedian</code>	<code>cupy.nanmedian</code>
<code>numpy.nanmin</code>	<code>cupy.nanmin</code>
<code>numpy.nanpercentile</code>	-
<code>numpy.nanprod</code>	<code>cupy.nanprod</code>
<code>numpy.nanquantile</code>	-
<code>numpy.nanstd</code>	<code>cupy.nanstd</code>
<code>numpy.nansum</code>	<code>cupy.nansum</code>
<code>numpy.nanvar</code>	<code>cupy.nanvar</code>
<code>numpy.ndfromtxt</code>	-
<code>numpy.ndim</code>	<code>cupy.ndim</code>
<code>numpy.negative</code>	<code>cupy.negative</code>
<code>numpy.nested_iters</code>	-
<code>numpy.nextafter</code>	<code>cupy.nextafter</code>
<code>numpy.nonzero</code>	<code>cupy.nonzero</code>
<code>numpy.not_equal</code>	<code>cupy.not_equal</code>
<code>numpy.obj2sctype</code>	<code>cupy.obj2sctype</code> ( <i>alias of</i> <code>numpy.obj2sctype</code> )
<code>numpy.ones</code>	<code>cupy.ones</code>
<code>numpy.ones_like</code>	<code>cupy.ones_like</code>
<code>numpy.outer</code>	<code>cupy.outer</code>
<code>numpy.packbits</code>	<code>cupy.packbits</code>
<code>numpy.pad</code>	<code>cupy.pad</code>
<code>numpy.partition</code>	<code>cupy.partition</code>
<code>numpy.percentile</code>	<code>cupy.percentile</code>
<code>numpy.piecewise</code>	<code>cupy.piecewise</code>
<code>numpy.place</code>	<code>cupy.place</code>
<code>numpy.poly</code>	-

continues on next page

Table 142 – continued from previous page

NumPy	CuPy
<code>numpy.polyadd</code>	<code>cupy.polyadd</code>
<code>numpy.polyder</code>	-
<code>numpy.polydiv</code>	-
<code>numpy.polyfit</code>	<code>cupy.polyfit</code>
<code>numpy.polyint</code>	-
<code>numpy.polymul</code>	<code>cupy.polymul</code>
<code>numpy.polysub</code>	<code>cupy.polysub</code>
<code>numpy.polyval</code>	<code>cupy.polyval</code>
<code>numpy.positive</code>	-
<code>numpy.power</code>	<code>cupy.power</code>
<code>numpy.printoptions</code>	-
<code>numpy.prod</code>	<code>cupy.prod</code>
<code>numpy.product</code>	-
<code>numpy.promote_types</code>	<code>cupy.promote_types</code> ( <i>alias of</i> <code>numpy.promote_types</code> )
<code>numpy.ptp</code>	<code>cupy.ptp</code>
<code>numpy.put</code>	<code>cupy.put</code>
<code>numpy.put_along_axis</code>	-
<code>numpy.putmask</code>	<code>cupy.putmask</code>
<code>numpy.quantile</code>	<code>cupy.quantile</code>
<code>numpy.rad2deg</code>	<code>cupy.rad2deg</code>
<code>numpy.radians</code>	<code>cupy.radians</code>
<code>numpy.ravel</code>	<code>cupy.ravel</code>
<code>numpy.ravel_multi_index</code>	<code>cupy.ravel_multi_index</code>
<code>numpy.real</code>	<code>cupy.real</code>
<code>numpy.real_if_close</code>	-
<code>numpy.recfromcsv</code>	-
<code>numpy.recfromtxt</code>	-
<code>numpy.reciprocal</code>	<code>cupy.reciprocal</code>
<code>numpy.remainder</code>	<code>cupy.remainder</code>
<code>numpy.repeat</code>	<code>cupy.repeat</code>
<code>numpy.require</code>	<code>cupy.require</code>
<code>numpy.reshape</code>	<code>cupy.reshape</code>
<code>numpy.resize</code>	<code>cupy.resize</code>
<code>numpy.result_type</code>	<code>cupy.result_type</code>
<code>numpy.right_shift</code>	<code>cupy.right_shift</code>
<code>numpy rint</code>	<code>cupy.rint</code>
<code>numpy.roll</code>	<code>cupy.roll</code>
<code>numpy.rollaxis</code>	<code>cupy.rollaxis</code>
<code>numpy.roots</code>	<code>cupy.roots</code>
<code>numpy.rot90</code>	<code>cupy.rot90</code>
<code>numpy.round</code>	<code>cupy.round</code>
<code>numpy.round_</code>	<code>cupy.round_</code>
<code>numpy.row_stack</code>	-
<code>numpy.safe_eval</code>	-
<code>numpy.save</code>	<code>cupy.save</code>
<code>numpy.savetxt</code>	-
<code>numpy.savez</code>	<code>cupy.savez</code>
<code>numpy.savez_compressed</code>	<code>cupy.savez_compressed</code>
<code>numpy.sctype2char</code>	<code>cupy.sctype2char</code> ( <i>alias of</i> <code>numpy.sctype2char</code> )

continues on next page



Table 142 – continued from previous page

NumPy	CuPy
<code>numpy.searchsorted</code>	<code>cupy.searchsorted</code>
<code>numpy.select</code>	<code>cupy.select</code>
<code>numpy.set_numeric_ops</code>	-
<code>numpy.set_printoptions</code>	-
<code>numpy.set_string_function</code>	-
<code>numpy.setbufsize</code>	-
<code>numpy.setdiff1d</code>	-
<code>numpy.seterr</code>	-
<code>numpy.seterrcall</code>	-
<code>numpy.seterrobj</code>	-
<code>numpy.setxor1d</code>	-
<code>numpy.shape</code>	<code>cupy.shape</code>
<code>numpy.shares_memory</code>	<code>cupy.shares_memory</code>
<code>numpy.show_config</code>	<code>cupy.show_config</code>
<code>numpy.sign</code>	<code>cupy.sign</code>
<code>numpy.signbit</code>	<code>cupy.signbit</code>
<code>numpy.sin</code>	<code>cupy.sin</code>
<code>numpy.sinc</code>	<code>cupy.sinc</code>
<code>numpy.sinh</code>	<code>cupy.sinh</code>
<code>numpy.size</code>	<code>cupy.size</code>
<code>numpy.sometrue</code>	-
<code>numpy.sort</code>	<code>cupy.sort</code>
<code>numpy.sort_complex</code>	<code>cupy.sort_complex</code>
<code>numpy.source</code>	-
<code>numpy.spacing</code>	-
<code>numpy.split</code>	<code>cupy.split</code>
<code>numpy.sqrt</code>	<code>cupy.sqrt</code>
<code>numpy.square</code>	<code>cupy.square</code>
<code>numpy.squeeze</code>	<code>cupy.squeeze</code>
<code>numpy.stack</code>	<code>cupy.stack</code>
<code>numpy.std</code>	<code>cupy.std</code>
<code>numpy.subtract</code>	<code>cupy.subtract</code>
<code>numpy.sum</code>	<code>cupy.sum</code>
<code>numpy.swapaxes</code>	<code>cupy.swapaxes</code>
<code>numpy.take</code>	<code>cupy.take</code>
<code>numpy.take_along_axis</code>	<code>cupy.take_along_axis</code>
<code>numpy.tan</code>	<code>cupy.tan</code>
<code>numpy.tanh</code>	<code>cupy.tanh</code>
<code>numpy.tensordot</code>	<code>cupy.tensordot</code>
<code>numpy.tile</code>	<code>cupy.tile</code>
<code>numpy.trace</code>	<code>cupy.trace</code>
<code>numpy.transpose</code>	<code>cupy.transpose</code>
<code>numpy.trapz</code>	-
<code>numpy.tri</code>	<code>cupy.tri</code>
<code>numpy.tril</code>	<code>cupy.tril</code>
<code>numpy.tril_indices</code>	-
<code>numpy.tril_indices_from</code>	-
<code>numpy.trim_zeros</code>	<code>cupy.trim_zeros</code>
<code>numpy.triu</code>	<code>cupy.triu</code>

continues on next page

Table 142 – continued from previous page

NumPy	CuPy
<code>numpy.triu_indices</code>	-
<code>numpy.triu_indices_from</code>	-
<code>numpy.true_divide</code>	<code>cupy.true_divide</code>
<code>numpy.trunc</code>	<code>cupy.trunc</code>
<code>numpy.typecode</code>	<code>cupy.typecode</code> ( <i>alias of</i> <code>numpy.typecode</code> )
<code>numpy.union1d</code>	-
<code>numpy.unique</code>	<code>cupy.unique</code>
<code>numpy.unpackbits</code>	<code>cupy.unpackbits</code>
<code>numpy.unravel_index</code>	<code>cupy.unravel_index</code>
<code>numpy.unwrap</code>	<code>cupy.unwrap</code>
<code>numpy.vander</code>	-
<code>numpy.var</code>	<code>cupy.var</code>
<code>numpy.vdot</code>	<code>cupy.vdot</code>
<code>numpy.vsplit</code>	<code>cupy.vsplit</code>
<code>numpy.vstack</code>	<code>cupy.vstack</code>
<code>numpy.where</code>	<code>cupy.where</code>
<code>numpy.who</code>	<code>cupy.who</code>
<code>numpy.zeros</code>	<code>cupy.zeros</code>
<code>numpy.zeros_like</code>	<code>cupy.zeros_like</code>

## Multi-Dimensional Array

NumPy	CuPy
<code>numpy.ndarray.all()</code>	<code>cupy.ndarray.all()</code>
<code>numpy.ndarray.any()</code>	<code>cupy.ndarray.any()</code>
<code>numpy.ndarray.argmax()</code>	<code>cupy.ndarray.argmax()</code>
<code>numpy.ndarray.argmin()</code>	<code>cupy.ndarray.argmin()</code>
<code>numpy.ndarray.argpartition()</code>	<code>cupy.ndarray.argpartition()</code>
<code>numpy.ndarray.argsort()</code>	<code>cupy.ndarray.argsort()</code>
<code>numpy.ndarray.astype()</code>	<code>cupy.ndarray.astype()</code>
<code>numpy.ndarray.byteswap()</code>	-
<code>numpy.ndarray.choose()</code>	<code>cupy.ndarray.choose()</code>
<code>numpy.ndarray.clip()</code>	<code>cupy.ndarray.clip()</code>
<code>numpy.ndarray.compress()</code>	<code>cupy.ndarray.compress()</code>
<code>numpy.ndarray.conj()</code>	<code>cupy.ndarray.conj()</code>
<code>numpy.ndarray.conjugate()</code>	<code>cupy.ndarray.conjugate()</code>
<code>numpy.ndarray.copy()</code>	<code>cupy.ndarray.copy()</code>
<code>numpy.ndarray.cumprod()</code>	<code>cupy.ndarray.cumprod()</code>
<code>numpy.ndarray.cumsum()</code>	<code>cupy.ndarray.cumsum()</code>
<code>numpy.ndarray.diagonal()</code>	<code>cupy.ndarray.diagonal()</code>
<code>numpy.ndarray.dot()</code>	<code>cupy.ndarray.dot()</code>
<code>numpy.ndarray.dump()</code>	<code>cupy.ndarray.dump()</code>
<code>numpy.ndarray.dumps()</code>	<code>cupy.ndarray.dumps()</code>
<code>numpy.ndarray.fill()</code>	<code>cupy.ndarray.fill()</code>
<code>numpy.ndarray.flatten()</code>	<code>cupy.ndarray.flatten()</code>
<code>numpy.ndarray.getfield()</code>	-
<code>numpy.ndarray.item()</code>	<code>cupy.ndarray.item()</code>
<code>numpy.ndarray.itemset()</code>	-

continues on next page

Table 143 – continued from previous page

NumPy	CuPy
<code>numpy.ndarray.max()</code>	<code>cupy.ndarray.max()</code>
<code>numpy.ndarray.mean()</code>	<code>cupy.ndarray.mean()</code>
<code>numpy.ndarray.min()</code>	<code>cupy.ndarray.min()</code>
<code>numpy.ndarray.newbyteorder()</code>	-
<code>numpy.ndarray.nonzero()</code>	<code>cupy.ndarray.nonzero()</code>
<code>numpy.ndarray.partition()</code>	<code>cupy.ndarray.partition()</code>
<code>numpy.ndarray.prod()</code>	<code>cupy.ndarray.prod()</code>
<code>numpy.ndarray.ptp()</code>	<code>cupy.ndarray.ptp()</code>
<code>numpy.ndarray.put()</code>	<code>cupy.ndarray.put()</code>
<code>numpy.ndarray.ravel()</code>	<code>cupy.ndarray.ravel()</code>
<code>numpy.ndarray.repeat()</code>	<code>cupy.ndarray.repeat()</code>
<code>numpy.ndarray.reshape()</code>	<code>cupy.ndarray.reshape()</code>
<code>numpy.ndarray.resize()</code>	-
<code>numpy.ndarray.round()</code>	<code>cupy.ndarray.round()</code>
<code>numpy.ndarray.searchsorted()</code>	-
<code>numpy.ndarray.setfield()</code>	-
<code>numpy.ndarray.setflags()</code>	-
<code>numpy.ndarray.sort()</code>	<code>cupy.ndarray.sort()</code>
<code>numpy.ndarray.squeeze()</code>	<code>cupy.ndarray.squeeze()</code>
<code>numpy.ndarray.std()</code>	<code>cupy.ndarray.std()</code>
<code>numpy.ndarray.sum()</code>	<code>cupy.ndarray.sum()</code>
<code>numpy.ndarray.swapaxes()</code>	<code>cupy.ndarray.swapaxes()</code>
<code>numpy.ndarray.take()</code>	<code>cupy.ndarray.take()</code>
<code>numpy.ndarray.tobytes()</code>	<code>cupy.ndarray.tobytes()</code>
<code>numpy.ndarray.tofile()</code>	<code>cupy.ndarray.tofile()</code>
<code>numpy.ndarray.tolist()</code>	<code>cupy.ndarray.tolist()</code>
<code>numpy.ndarray.tostring()</code>	-
<code>numpy.ndarray.trace()</code>	<code>cupy.ndarray.trace()</code>
<code>numpy.ndarray.transpose()</code>	<code>cupy.ndarray.transpose()</code>
<code>numpy.ndarray.var()</code>	<code>cupy.ndarray.var()</code>
<code>numpy.ndarray.view()</code>	<code>cupy.ndarray.view()</code>

## Linear Algebra

NumPy	CuPy
<code>numpy.linalg.cholesky</code>	<code>cupy.linalg.cholesky</code>
<code>numpy.linalg.cond</code>	-
<code>numpy.linalg.det</code>	<code>cupy.linalg.det</code>
<code>numpy.linalg.eig</code>	-
<code>numpy.linalg.eigh</code>	<code>cupy.linalg.eigh</code>
<code>numpy.linalg.eigvals</code>	-
<code>numpy.linalg.eigvalsh</code>	<code>cupy.linalg.eigvalsh</code>
<code>numpy.linalg.inv</code>	<code>cupy.linalg.inv</code>
<code>numpy.linalg.lstsq</code>	<code>cupy.linalg.lstsq</code>
<code>numpy.linalg.matrix_power</code>	<code>cupy.linalg.matrix_power</code>
<code>numpy.linalg.matrix_rank</code>	<code>cupy.linalg.matrix_rank</code>
<code>numpy.linalg.multi_dot</code>	-
<code>numpy.linalg.norm</code>	<code>cupy.linalg.norm</code>
<code>numpy.linalg.pinv</code>	<code>cupy.linalg.pinv</code>
<code>numpy.linalg.qr</code>	<code>cupy.linalg.qr</code>
<code>numpy.linalg.slogdet</code>	<code>cupy.linalg.slogdet</code>
<code>numpy.linalg.solve</code>	<code>cupy.linalg.solve</code>
<code>numpy.linalg.svd</code>	<code>cupy.linalg.svd</code>
<code>numpy.linalg.tensorinv</code>	<code>cupy.linalg.tensorinv</code>
<code>numpy.linalg.tensorsolve</code>	<code>cupy.linalg.tensorsolve</code>

## Discrete Fourier Transform

NumPy	CuPy
<code>numpy.fft.fft</code>	<code>cupy.fft.fft</code>
<code>numpy.fft.fft2</code>	<code>cupy.fft.fft2</code>
<code>numpy.fft.fftfreq</code>	<code>cupy.fft.fftfreq</code>
<code>numpy.fft.fftn</code>	<code>cupy.fft.fftn</code>
<code>numpy.fft.fftshift</code>	<code>cupy.fft.fftshift</code>
<code>numpy.fft.hfft</code>	<code>cupy.fft.hfft</code>
<code>numpy.fft.ifft</code>	<code>cupy.fft.ifft</code>
<code>numpy.fft.ifft2</code>	<code>cupy.fft.ifft2</code>
<code>numpy.fft.ifftn</code>	<code>cupy.fft.ifftn</code>
<code>numpy.fft.ifftshift</code>	<code>cupy.fft.ifftshift</code>
<code>numpy.fft.ihfft</code>	<code>cupy.fft.ihfft</code>
<code>numpy.fft.irfft</code>	<code>cupy.fft.irfft</code>
<code>numpy.fft.irfft2</code>	<code>cupy.fft.irfft2</code>
<code>numpy.fft.irfftn</code>	<code>cupy.fft.irfftn</code>
<code>numpy.fft.rfft</code>	<code>cupy.fft.rfft</code>
<code>numpy.fft.rfft2</code>	<code>cupy.fft.rfft2</code>
<code>numpy.fft.rfftfreq</code>	<code>cupy.fft.rfftfreq</code>
<code>numpy.fft.rfftn</code>	<code>cupy.fft.rfftn</code>

## Random Sampling

NumPy	CuPy
<code>numpy.random.beta</code>	<code>cupy.random.beta</code>
<code>numpy.random.binomial</code>	<code>cupy.random.binomial</code>
<code>numpy.random.bytes</code>	<code>cupy.random.bytes</code>
<code>numpy.random.chisquare</code>	<code>cupy.random.chisquare</code>
<code>numpy.random.choice</code>	<code>cupy.random.choice</code>
<code>numpy.random.default_rng</code>	<code>cupy.random.default_rng</code>
<code>numpy.random.dirichlet</code>	<code>cupy.random.dirichlet</code>
<code>numpy.random.exponential</code>	<code>cupy.random.exponential</code>
<code>numpy.random.f</code>	<code>cupy.random.f</code>
<code>numpy.random.gamma</code>	<code>cupy.random.gamma</code>
<code>numpy.random.geometric</code>	<code>cupy.random.geometric</code>
<code>numpy.random.get_state</code>	-
<code>numpy.random.gumbel</code>	<code>cupy.random.gumbel</code>
<code>numpy.random.hypergeometric</code>	<code>cupy.random.hypergeometric</code>
<code>numpy.random.laplace</code>	<code>cupy.random.laplace</code>
<code>numpy.random.logistic</code>	<code>cupy.random.logistic</code>
<code>numpy.random.lognormal</code>	<code>cupy.random.lognormal</code>
<code>numpy.random.logseries</code>	<code>cupy.random.logseries</code>
<code>numpy.random.multinomial</code>	<code>cupy.random.multinomial</code>
<code>numpy.random.multivariate_normal</code>	<code>cupy.random.multivariate_normal</code>
<code>numpy.random.negative_binomial</code>	<code>cupy.random.negative_binomial</code>
<code>numpy.random.noncentral_chisquare</code>	<code>cupy.random.noncentral_chisquare</code>
<code>numpy.random.noncentral_f</code>	<code>cupy.random.noncentral_f</code>
<code>numpy.random.normal</code>	<code>cupy.random.normal</code>
<code>numpy.random.pareto</code>	<code>cupy.random.pareto</code>
<code>numpy.random.permutation</code>	<code>cupy.random.permutation</code>
<code>numpy.random.poisson</code>	<code>cupy.random.poisson</code>
<code>numpy.random.power</code>	<code>cupy.random.power</code>
<code>numpy.random.rand</code>	<code>cupy.random.rand</code>
<code>numpy.random.randint</code>	<code>cupy.random.randint</code>
<code>numpy.random.randn</code>	<code>cupy.random.randn</code>
<code>numpy.random.random</code>	<code>cupy.random.random</code>
<code>numpy.random.random_integers</code>	<code>cupy.random.random_integers</code>
<code>numpy.random.random_sample</code>	<code>cupy.random.random_sample</code>
<code>numpy.random.ranf</code>	<code>cupy.random.ranf</code>
<code>numpy.random.rayleigh</code>	<code>cupy.random.rayleigh</code>
<code>numpy.random.sample</code>	<code>cupy.random.sample</code>
<code>numpy.random.seed</code>	<code>cupy.random.seed</code>
<code>numpy.random.set_state</code>	-
<code>numpy.random.shuffle</code>	<code>cupy.random.shuffle</code>
<code>numpy.random.standard_cauchy</code>	<code>cupy.random.standard_cauchy</code>
<code>numpy.random.standard_exponential</code>	<code>cupy.random.standard_exponential</code>
<code>numpy.random.standard_gamma</code>	<code>cupy.random.standard_gamma</code>
<code>numpy.random.standard_normal</code>	<code>cupy.random.standard_normal</code>
<code>numpy.random.standard_t</code>	<code>cupy.random.standard_t</code>
<code>numpy.random.triangular</code>	<code>cupy.random.triangular</code>
<code>numpy.random.uniform</code>	<code>cupy.random.uniform</code>

continues on next page

Table 144 – continued from previous page

NumPy	CuPy
<code>numpy.random.vonmises</code>	<code>cupy.random.vonmises</code>
<code>numpy.random.wald</code>	<code>cupy.random.wald</code>
<code>numpy.random.weibull</code>	<code>cupy.random.weibull</code>
<code>numpy.random.zipf</code>	<code>cupy.random.zipf</code>

## 5.10.2 SciPy / CuPy APIs

### Discrete Fourier Transform

SciPy	CuPy
<code>scipy.fft.dct</code>	-
<code>scipy.fft.dctn</code>	-
<code>scipy.fft.dst</code>	-
<code>scipy.fft.dstn</code>	-
<code>scipy.fft.fft</code>	<code>cupyx.scipy.fft.fft</code>
<code>scipy.fft.fft2</code>	<code>cupyx.scipy.fft.fft2</code>
<code>scipy.fft.fftfreq</code>	<code>cupyx.scipy.fft.fftfreq</code>
<code>scipy.fft.fftn</code>	<code>cupyx.scipy.fft.fftn</code>
<code>scipy.fft.fftshift</code>	<code>cupyx.scipy.fft.fftshift</code>
<code>scipy.fft.get_workers</code>	-
<code>scipy.fft.hfft</code>	<code>cupyx.scipy.fft.hfft</code>
<code>scipy.fft.hfft2</code>	-
<code>scipy.fft.hfftn</code>	-
<code>scipy.fft.idct</code>	-
<code>scipy.fft.idctn</code>	-
<code>scipy.fft.idst</code>	-
<code>scipy.fft.idstn</code>	-
<code>scipy.fft.iff</code>	<code>cupyx.scipy.fft.iff</code>
<code>scipy.fft.iff2</code>	<code>cupyx.scipy.fft.iff2</code>
<code>scipy.fft.iffn</code>	<code>cupyx.scipy.fft.iffn</code>
<code>scipy.fft.iffshift</code>	<code>cupyx.scipy.fft.iffshift</code>
<code>scipy.fft.ihfft</code>	<code>cupyx.scipy.fft.ihfft</code>
<code>scipy.fft.ihfft2</code>	-
<code>scipy.fft.ihfftn</code>	-
<code>scipy.fft.irfft</code>	<code>cupyx.scipy.fft.irfft</code>
<code>scipy.fft.irfft2</code>	<code>cupyx.scipy.fft.irfft2</code>
<code>scipy.fft.irfftn</code>	<code>cupyx.scipy.fft.irfftn</code>
<code>scipy.fft.next_fast_len</code>	<code>cupyx.scipy.fft.next_fast_len</code>
<code>scipy.fft.register_backend</code>	-
<code>scipy.fft.rfft</code>	<code>cupyx.scipy.fft.rfft</code>
<code>scipy.fft.rfft2</code>	<code>cupyx.scipy.fft.rfft2</code>
<code>scipy.fft.rfftfreq</code>	<code>cupyx.scipy.fft.rfftfreq</code>
<code>scipy.fft.rfftn</code>	<code>cupyx.scipy.fft.rfftn</code>
<code>scipy.fft.set_backend</code>	-
<code>scipy.fft.set_global_backend</code>	-
<code>scipy.fft.set_workers</code>	-
<code>scipy.fft.skip_backend</code>	-

## Discrete Fourier Transform (legacy fftpack module)

SciPy	CuPy
<code>scipy.fftpack.cc_diff</code>	-
<code>scipy.fftpack.cs_diff</code>	-
<code>scipy.fftpack.dct</code>	-
<code>scipy.fftpack.dctn</code>	-
<code>scipy.fftpack.diff</code>	-
<code>scipy.fftpack.dst</code>	-
<code>scipy.fftpack.dstn</code>	-
<code>scipy.fftpack.fft</code>	<code>cupyx.scipy.fftpack.fft</code>
<code>scipy.fftpack.fft2</code>	<code>cupyx.scipy.fftpack.fft2</code>
<code>scipy.fftpack.fftfreq</code>	-
<code>scipy.fftpack.fftn</code>	<code>cupyx.scipy.fftpack.fftn</code>
<code>scipy.fftpack.fftshift</code>	-
<code>scipy.fftpack.hilbert</code>	-
<code>scipy.fftpack.idct</code>	-
<code>scipy.fftpack.idctn</code>	-
<code>scipy.fftpack.idst</code>	-
<code>scipy.fftpack.idstn</code>	-
<code>scipy.fftpack.ifft</code>	<code>cupyx.scipy.fftpack.ifft</code>
<code>scipy.fftpack.ifft2</code>	<code>cupyx.scipy.fftpack.ifft2</code>
<code>scipy.fftpack.ifftn</code>	<code>cupyx.scipy.fftpack.ifftn</code>
<code>scipy.fftpack.ifftshift</code>	-
<code>scipy.fftpack.ihilbert</code>	-
<code>scipy.fftpack.irfft</code>	<code>cupyx.scipy.fftpack.irfft</code>
<code>scipy.fftpack.itylbert</code>	-
<code>scipy.fftpack.next_fast_len</code>	-
<code>scipy.fftpack.rfft</code>	<code>cupyx.scipy.fftpack.rfft</code>
<code>scipy.fftpack.rfftfreq</code>	-
<code>scipy.fftpack.sc_diff</code>	-
<code>scipy.fftpack.shift</code>	-
<code>scipy.fftpack.ss_diff</code>	-
<code>scipy.fftpack.tilbert</code>	-

## Sparse Matrices

SciPy	CuPy
<code>scipy.sparse.block_diag</code>	-
<code>scipy.sparse.bmat</code>	<code>cupyx.scipy.sparse.bmat</code>
<code>scipy.sparse.diags</code>	<code>cupyx.scipy.sparse.diags</code>
<code>scipy.sparse.eye</code>	<code>cupyx.scipy.sparse.eye</code>
<code>scipy.sparse.find</code>	<code>cupyx.scipy.sparse.find</code>
<code>scipy.sparse.hstack</code>	<code>cupyx.scipy.sparse.hstack</code>
<code>scipy.sparse.identity</code>	<code>cupyx.scipy.sparse.identity</code>
<code>scipy.sparse.issparse</code>	<code>cupyx.scipy.sparse.issparse</code>
<code>scipy.sparse.isspmatrix</code>	<code>cupyx.scipy.sparse.isspmatrix</code>
<code>scipy.sparse.isspmatrix_bsr</code>	-
<code>scipy.sparse.isspmatrix_coo</code>	<code>cupyx.scipy.sparse.isspmatrix_coo</code>
<code>scipy.sparse.isspmatrix_csc</code>	<code>cupyx.scipy.sparse.isspmatrix_csc</code>
<code>scipy.sparse.isspmatrix_csr</code>	<code>cupyx.scipy.sparse.isspmatrix_csr</code>
<code>scipy.sparse.isspmatrix_dia</code>	<code>cupyx.scipy.sparse.isspmatrix_dia</code>
<code>scipy.sparse.isspmatrix_dok</code>	-
<code>scipy.sparse.isspmatrix_lil</code>	-
<code>scipy.sparse.kron</code>	<code>cupyx.scipy.sparse.kron</code>
<code>scipy.sparse.kronsum</code>	-
<code>scipy.sparse.load_npz</code>	-
<code>scipy.sparse.rand</code>	<code>cupyx.scipy.sparse.rand</code>
<code>scipy.sparse.random</code>	<code>cupyx.scipy.sparse.random</code>
<code>scipy.sparse.save_npz</code>	-
<code>scipy.sparse.spdiags</code>	<code>cupyx.scipy.sparse.spdiags</code>
<code>scipy.sparse.tril</code>	<code>cupyx.scipy.sparse.tril</code>
<code>scipy.sparse.triu</code>	<code>cupyx.scipy.sparse.triu</code>
<code>scipy.sparse.vstack</code>	<code>cupyx.scipy.sparse.vstack</code>



## Sparse Linear Algebra

SciPy	CuPy
<code>scipy.sparse.linalg.aslinearoperator</code>	<code>cupyx.scipy.sparse.linalg.aslinearoperator</code>
<code>scipy.sparse.linalg.bicg</code>	-
<code>scipy.sparse.linalg.bicgstab</code>	-
<code>scipy.sparse.linalg.cg</code>	<code>cupyx.scipy.sparse.linalg.cg</code>
<code>scipy.sparse.linalg.cgs</code>	-
<code>scipy.sparse.linalg.eigs</code>	-
<code>scipy.sparse.linalg.eigsh</code>	<code>cupyx.scipy.sparse.linalg.eigsh</code>
<code>scipy.sparse.linalg.expm</code>	-
<code>scipy.sparse.linalg.expm_multiply</code>	-
<code>scipy.sparse.linalg.factorized</code>	<code>cupyx.scipy.sparse.linalg.factorized</code>
<code>scipy.sparse.linalg.gcrotmk</code>	-
<code>scipy.sparse.linalg.gmres</code>	<code>cupyx.scipy.sparse.linalg.gmres</code>
<code>scipy.sparse.linalg.inv</code>	-
<code>scipy.sparse.linalg.lgmres</code>	-
<code>scipy.sparse.linalg.lobpcg</code>	<code>cupyx.scipy.sparse.linalg.lobpcg</code>
<code>scipy.sparse.linalg.lsmr</code>	-
<code>scipy.sparse.linalg.lsqr</code>	<code>cupyx.scipy.sparse.linalg.lsqr</code>
<code>scipy.sparse.linalg.minres</code>	-
<code>scipy.sparse.linalg.norm</code>	<code>cupyx.scipy.sparse.linalg.norm</code>
<code>scipy.sparse.linalg.onenormest</code>	-
<code>scipy.sparse.linalg.qmr</code>	-
<code>scipy.sparse.linalg.spilu</code>	<code>cupyx.scipy.sparse.linalg.spilu</code>
<code>scipy.sparse.linalg.splu</code>	<code>cupyx.scipy.sparse.linalg.splu</code>
<code>scipy.sparse.linalg.spsolve</code>	<code>cupyx.scipy.sparse.linalg.spsolve</code>
<code>scipy.sparse.linalg.spsolve_triangular</code>	<code>cupyx.scipy.sparse.linalg.spsolve_triangular</code>
<code>scipy.sparse.linalg.svds</code>	<code>cupyx.scipy.sparse.linalg.svds</code>
<code>scipy.sparse.linalg.use_solver</code>	-

## Advanced Linear Algebra

SciPy	CuPy
<code>scipy.linalg.block_diag</code>	<code>cupyx.scipy.linalg.block_diag</code>
<code>scipy.linalg.cdf2rdf</code>	-
<code>scipy.linalg.cho_factor</code>	-
<code>scipy.linalg.cho_solve</code>	-
<code>scipy.linalg.cho_solve_banded</code>	-
<code>scipy.linalg.cholesky_banded</code>	-
<code>scipy.linalg.circulant</code>	<code>cupyx.scipy.linalg.circulant</code>
<code>scipy.linalg.clarkson_woodruff_transform</code>	-
<code>scipy.linalg.companion</code>	<code>cupyx.scipy.linalg.companion</code>
<code>scipy.linalg.convolution_matrix</code>	<code>cupyx.scipy.linalg.convolution_matrix</code>
<code>scipy.linalg.coshm</code>	-
<code>scipy.linalg.cosm</code>	-
<code>scipy.linalg.cossin</code>	-
<code>scipy.linalg.dft</code>	<code>cupyx.scipy.linalg.dft</code>
<code>scipy.linalg.diagsvd</code>	-

continues on next page

Table 147 – continued from previous page

SciPy	CuPy
<code>scipy.linalg.eig_banded</code>	-
<code>scipy.linalg.eigh_tridiagonal</code>	-
<code>scipy.linalg.eigvals_banded</code>	-
<code>scipy.linalg.eigvalsh_tridiagonal</code>	-
<code>scipy.linalg.expm</code>	-
<code>scipy.linalg.expm_cond</code>	-
<code>scipy.linalg.expm_frechet</code>	-
<code>scipy.linalg.fiedler</code>	<code>cupyx.scipy.linalg.fiedler</code>
<code>scipy.linalg.fiedler_companion</code>	<code>cupyx.scipy.linalg.fiedler_companion</code>
<code>scipy.linalg.find_best_blas_type</code>	-
<code>scipy.linalg.fractional_matrix_power</code>	-
<code>scipy.linalg.funm</code>	-
<code>scipy.linalg.get_blas_funcs</code>	-
<code>scipy.linalg.get_lapack_funcs</code>	-
<code>scipy.linalg.hadamard</code>	<code>cupyx.scipy.linalg.hadamard</code>
<code>scipy.linalg.hankel</code>	<code>cupyx.scipy.linalg.hankel</code>
<code>scipy.linalg.helmert</code>	<code>cupyx.scipy.linalg.helmert</code>
<code>scipy.linalg.hessenberg</code>	-
<code>scipy.linalg.hilbert</code>	<code>cupyx.scipy.linalg.hilbert</code>
<code>scipy.linalg.invhilbert</code>	-
<code>scipy.linalg.invpascal</code>	-
<code>scipy.linalg.khatri_rao</code>	-
<code>scipy.linalg.kron</code>	<code>cupyx.scipy.linalg.kron</code>
<code>scipy.linalg.ldl</code>	-
<code>scipy.linalg.leslie</code>	<code>cupyx.scipy.linalg.leslie</code>
<code>scipy.linalg.logm</code>	-
<code>scipy.linalg.lu</code>	<code>cupyx.scipy.linalg.lu</code>
<code>scipy.linalg.lu_factor</code>	<code>cupyx.scipy.linalg.lu_factor</code>
<code>scipy.linalg.lu_solve</code>	<code>cupyx.scipy.linalg.lu_solve</code>
<code>scipy.linalg.matmul_toeplitz</code>	-
<code>scipy.linalg.matrix_balance</code>	-
<code>scipy.linalg.null_space</code>	-
<code>scipy.linalg.ordqz</code>	-
<code>scipy.linalg.orth</code>	-
<code>scipy.linalg.orthogonal_procrustes</code>	-
<code>scipy.linalg.pascal</code>	-
<code>scipy.linalg.pinv2</code>	-
<code>scipy.linalg.pinvh</code>	-
<code>scipy.linalg.polar</code>	-
<code>scipy.linalg.qr_delete</code>	-
<code>scipy.linalg.qr_insert</code>	-
<code>scipy.linalg.qr_multiply</code>	-
<code>scipy.linalg.qr_update</code>	-
<code>scipy.linalg.qz</code>	-
<code>scipy.linalg.rq</code>	-
<code>scipy.linalg.rsf2csf</code>	-
<code>scipy.linalg.schur</code>	-
<code>scipy.linalg.signm</code>	-
<code>scipy.linalg.sinhm</code>	-

continues on next page

Table 147 – continued from previous page

SciPy	CuPy
<code>scipy.linalg.sinm</code>	-
<code>scipy.linalg.solve_banded</code>	-
<code>scipy.linalg.solve_circulant</code>	-
<code>scipy.linalg.solve_continuous_are</code>	-
<code>scipy.linalg.solve_continuous_lyapunov</code>	-
<code>scipy.linalg.solve_discrete_are</code>	-
<code>scipy.linalg.solve_discrete_lyapunov</code>	-
<code>scipy.linalg.solve_lyapunov</code>	-
<code>scipy.linalg.solve_sylvester</code>	-
<code>scipy.linalg.solve_toeplitz</code>	-
<code>scipy.linalg.solve_triangular</code>	<code>cupyx.scipy.linalg.solve_triangular</code>
<code>scipy.linalg.solveh_banded</code>	-
<code>scipy.linalg.sqrnm</code>	-
<code>scipy.linalg.subspace_angles</code>	-
<code>scipy.linalg.svdvals</code>	-
<code>scipy.linalg.tanhm</code>	-
<code>scipy.linalg.tanm</code>	-
<code>scipy.linalg.toeplitz</code>	<code>cupyx.scipy.linalg.toeplitz</code>
<code>scipy.linalg.tri</code>	<code>cupyx.scipy.linalg.tri</code>
<code>scipy.linalg.tril</code>	<code>cupyx.scipy.linalg.tril</code>
<code>scipy.linalg.triu</code>	<code>cupyx.scipy.linalg.triu</code>

## Multidimensional Image Processing

SciPy	CuPy
<code>scipy.ndimage.affine_transform</code>	<code>cupyx.scipy.ndimage.affine_transform</code>
<code>scipy.ndimage.binary_closing</code>	<code>cupyx.scipy.ndimage.binary_closing</code>
<code>scipy.ndimage.binary_dilation</code>	<code>cupyx.scipy.ndimage.binary_dilation</code>
<code>scipy.ndimage.binary_erosion</code>	<code>cupyx.scipy.ndimage.binary_erosion</code>
<code>scipy.ndimage.binary_fill_holes</code>	<code>cupyx.scipy.ndimage.binary_fill_holes</code>
<code>scipy.ndimage.binary_hit_or_miss</code>	<code>cupyx.scipy.ndimage.binary_hit_or_miss</code>
<code>scipy.ndimage.binary_opening</code>	<code>cupyx.scipy.ndimage.binary_opening</code>
<code>scipy.ndimage.binary_propagation</code>	<code>cupyx.scipy.ndimage.binary_propagation</code>
<code>scipy.ndimage.black_tophat</code>	<code>cupyx.scipy.ndimage.black_tophat</code>
<code>scipy.ndimage.center_of_mass</code>	<code>cupyx.scipy.ndimage.center_of_mass</code>
<code>scipy.ndimage.convolve</code>	<code>cupyx.scipy.ndimage.convolve</code>
<code>scipy.ndimage.convolve1d</code>	<code>cupyx.scipy.ndimage.convolve1d</code>
<code>scipy.ndimage.correlate</code>	<code>cupyx.scipy.ndimage.correlate</code>
<code>scipy.ndimage.correlate1d</code>	<code>cupyx.scipy.ndimage.correlate1d</code>
<code>scipy.ndimage.distance_transform_bf</code>	-
<code>scipy.ndimage.distance_transform_cdt</code>	-
<code>scipy.ndimage.distance_transform_edt</code>	-
<code>scipy.ndimage.extrema</code>	<code>cupyx.scipy.ndimage.extrema</code>
<code>scipy.ndimage.find_objects</code>	-
<code>scipy.ndimage.fourier_ellipsoid</code>	<code>cupyx.scipy.ndimage.fourier_ellipsoid</code>
<code>scipy.ndimage.fourier_gaussian</code>	<code>cupyx.scipy.ndimage.fourier_gaussian</code>
<code>scipy.ndimage.fourier_shift</code>	<code>cupyx.scipy.ndimage.fourier_shift</code>
<code>scipy.ndimage.fourier_uniform</code>	<code>cupyx.scipy.ndimage.fourier_uniform</code>

continues on next page

Table 148 – continued from previous page

SciPy	CuPy
<code>scipy.ndimage.gaussian_filter</code>	<code>cupyx.scipy.ndimage.gaussian_filter</code>
<code>scipy.ndimage.gaussian_filter1d</code>	<code>cupyx.scipy.ndimage.gaussian_filter1d</code>
<code>scipy.ndimage.gaussian_gradient_magnitude</code>	<code>cupyx.scipy.ndimage.gaussian_gradient_magnitude</code>
<code>scipy.ndimage.gaussian_laplace</code>	<code>cupyx.scipy.ndimage.gaussian_laplace</code>
<code>scipy.ndimage.generate_binary_structure</code>	<code>cupyx.scipy.ndimage.generate_binary_structure</code>
<code>scipy.ndimage.generic_filter</code>	<code>cupyx.scipy.ndimage.generic_filter</code>
<code>scipy.ndimage.generic_filter1d</code>	<code>cupyx.scipy.ndimage.generic_filter1d</code>
<code>scipy.ndimage.generic_gradient_magnitude</code>	<code>cupyx.scipy.ndimage.generic_gradient_magnitude</code>
<code>scipy.ndimage.generic_laplace</code>	<code>cupyx.scipy.ndimage.generic_laplace</code>
<code>scipy.ndimage.geometric_transform</code>	-
<code>scipy.ndimage.grey_closing</code>	<code>cupyx.scipy.ndimage.grey_closing</code>
<code>scipy.ndimage.grey_dilation</code>	<code>cupyx.scipy.ndimage.grey_dilation</code>
<code>scipy.ndimage.grey_erosion</code>	<code>cupyx.scipy.ndimage.grey_erosion</code>
<code>scipy.ndimage.grey_opening</code>	<code>cupyx.scipy.ndimage.grey_opening</code>
<code>scipy.ndimage.histogram</code>	<code>cupyx.scipy.ndimage.histogram</code>
<code>scipy.ndimage.iterate_structure</code>	<code>cupyx.scipy.ndimage.iterate_structure</code>
<code>scipy.ndimage.label</code>	<code>cupyx.scipy.ndimage.label</code>
<code>scipy.ndimage.labeled_comprehension</code>	<code>cupyx.scipy.ndimage.labeled_comprehension</code>
<code>scipy.ndimage.laplace</code>	<code>cupyx.scipy.ndimage.laplace</code>
<code>scipy.ndimage.map_coordinates</code>	<code>cupyx.scipy.ndimage.map_coordinates</code>
<code>scipy.ndimage.maximum</code>	<code>cupyx.scipy.ndimage.maximum</code>
<code>scipy.ndimage.maximum_filter</code>	<code>cupyx.scipy.ndimage.maximum_filter</code>
<code>scipy.ndimage.maximum_filter1d</code>	<code>cupyx.scipy.ndimage.maximum_filter1d</code>
<code>scipy.ndimage.maximum_position</code>	<code>cupyx.scipy.ndimage.maximum_position</code>
<code>scipy.ndimage.mean</code>	<code>cupyx.scipy.ndimage.mean</code>
<code>scipy.ndimage.median</code>	<code>cupyx.scipy.ndimage.median</code>
<code>scipy.ndimage.median_filter</code>	<code>cupyx.scipy.ndimage.median_filter</code>
<code>scipy.ndimage.minimum</code>	<code>cupyx.scipy.ndimage.minimum</code>
<code>scipy.ndimage.minimum_filter</code>	<code>cupyx.scipy.ndimage.minimum_filter</code>
<code>scipy.ndimage.minimum_filter1d</code>	<code>cupyx.scipy.ndimage.minimum_filter1d</code>
<code>scipy.ndimage.minimum_position</code>	<code>cupyx.scipy.ndimage.minimum_position</code>
<code>scipy.ndimage.morphological_gradient</code>	<code>cupyx.scipy.ndimage.morphological_gradient</code>
<code>scipy.ndimage.morphological_laplace</code>	<code>cupyx.scipy.ndimage.morphological_laplace</code>
<code>scipy.ndimage.percentile_filter</code>	<code>cupyx.scipy.ndimage.percentile_filter</code>
<code>scipy.ndimage.prewitt</code>	<code>cupyx.scipy.ndimage.prewitt</code>
<code>scipy.ndimage.rank_filter</code>	<code>cupyx.scipy.ndimage.rank_filter</code>
<code>scipy.ndimage.rotate</code>	<code>cupyx.scipy.ndimage.rotate</code>
<code>scipy.ndimage.shift</code>	<code>cupyx.scipy.ndimage.shift</code>
<code>scipy.ndimage.sobel</code>	<code>cupyx.scipy.ndimage.sobel</code>
<code>scipy.ndimage.spline_filter</code>	<code>cupyx.scipy.ndimage.spline_filter</code>
<code>scipy.ndimage.spline_filter1d</code>	<code>cupyx.scipy.ndimage.spline_filter1d</code>
<code>scipy.ndimage.standard_deviation</code>	<code>cupyx.scipy.ndimage.standard_deviation</code>
<code>scipy.ndimage.sum</code>	<code>cupyx.scipy.ndimage.sum</code>
<code>scipy.ndimage.sum_labels</code>	<code>cupyx.scipy.ndimage.sum_labels</code>
<code>scipy.ndimage.uniform_filter</code>	<code>cupyx.scipy.ndimage.uniform_filter</code>
<code>scipy.ndimage.uniform_filter1d</code>	<code>cupyx.scipy.ndimage.uniform_filter1d</code>
<code>scipy.ndimage.variance</code>	<code>cupyx.scipy.ndimage.variance</code>
<code>scipy.ndimage.watershed_ift</code>	-
<code>scipy.ndimage.white_tophat</code>	<code>cupyx.scipy.ndimage.white_tophat</code>

continues on next page

Table 148 – continued from previous page

SciPy	CuPy
<code>scipy.ndimage.zoom</code>	<code>cupyx.scipy.ndimage.zoom</code>

## Special Functions

SciPy	CuPy
<code>scipy.special.agm</code>	-
<code>scipy.special.ai_zeros</code>	-
<code>scipy.special.airy</code>	-
<code>scipy.special.airye</code>	-
<code>scipy.special.assoc_laguerre</code>	-
<code>scipy.special.bdtr</code>	-
<code>scipy.special.bdtrc</code>	-
<code>scipy.special.bdtri</code>	-
<code>scipy.special.bdtrik</code>	-
<code>scipy.special.bdtrln</code>	-
<code>scipy.special.bei</code>	-
<code>scipy.special.bei_zeros</code>	-
<code>scipy.special.beip</code>	-
<code>scipy.special.beip_zeros</code>	-
<code>scipy.special.ber</code>	-
<code>scipy.special.ber_zeros</code>	-
<code>scipy.special.bernoulli</code>	-
<code>scipy.special.berp</code>	-
<code>scipy.special.berp_zeros</code>	-
<code>scipy.special.besselpoly</code>	-
<code>scipy.special.beta</code>	-
<code>scipy.special.betainc</code>	-
<code>scipy.special.betaincinv</code>	-
<code>scipy.special.betaln</code>	-
<code>scipy.special.bi_zeros</code>	-
<code>scipy.special.binom</code>	-
<code>scipy.special.boxcox</code>	-
<code>scipy.special.boxcox1p</code>	-
<code>scipy.special.btdtr</code>	-
<code>scipy.special.btdtri</code>	-
<code>scipy.special.btdtria</code>	-
<code>scipy.special.btdtrib</code>	-
<code>scipy.special.c_roots</code>	-
<code>scipy.special.cbtr</code>	-
<code>scipy.special.cg_roots</code>	-
<code>scipy.special.chdtr</code>	-
<code>scipy.special.chdtrc</code>	-
<code>scipy.special.chdtri</code>	-
<code>scipy.special.chdtriv</code>	-
<code>scipy.special.chebyc</code>	-
<code>scipy.special.chebys</code>	-
<code>scipy.special.chebyt</code>	-
<code>scipy.special.chebyu</code>	-

continues on next page

Table 149 – continued from previous page

SciPy	CuPy
<code>scipy.special.chndtr</code>	-
<code>scipy.special.chndtridf</code>	-
<code>scipy.special.chndtrinc</code>	-
<code>scipy.special.chndtrix</code>	-
<code>scipy.special.clpmn</code>	-
<code>scipy.special.comb</code>	-
<code>scipy.special.cosdg</code>	-
<code>scipy.special.cosm1</code>	-
<code>scipy.special.cotdg</code>	-
<code>scipy.special.dawsn</code>	-
<code>scipy.special.digamma</code>	<code>cupyx.scipy.special.digamma</code>
<code>scipy.special.diric</code>	-
<code>scipy.special.ellip_harm</code>	-
<code>scipy.special.ellip_harm_2</code>	-
<code>scipy.special.ellip_normal</code>	-
<code>scipy.special.ellipe</code>	-
<code>scipy.special.ellipeinc</code>	-
<code>scipy.special.ellipj</code>	-
<code>scipy.special.ellipk</code>	-
<code>scipy.special.ellipkinc</code>	-
<code>scipy.special.ellipkm1</code>	-
<code>scipy.special.entr</code>	<code>cupyx.scipy.special.entr</code>
<code>scipy.special.erf</code>	<code>cupyx.scipy.special.erf</code>
<code>scipy.special.erf_zeros</code>	-
<code>scipy.special.erfc</code>	<code>cupyx.scipy.special.erfc</code>
<code>scipy.special.erfcinv</code>	<code>cupyx.scipy.special.erfcinv</code>
<code>scipy.special.erfcx</code>	<code>cupyx.scipy.special.erfcx</code>
<code>scipy.special.erfi</code>	-
<code>scipy.special.erfinv</code>	<code>cupyx.scipy.special.erfinv</code>
<code>scipy.special.euler</code>	-
<code>scipy.special.eval_chebyc</code>	-
<code>scipy.special.eval_chebys</code>	-
<code>scipy.special.eval_chebyt</code>	-
<code>scipy.special.eval_chebyu</code>	-
<code>scipy.special.eval_gegenbauer</code>	-
<code>scipy.special.eval_genlaguerre</code>	-
<code>scipy.special.eval_hermite</code>	-
<code>scipy.special.eval_hermitenorm</code>	-
<code>scipy.special.eval_jacobi</code>	-
<code>scipy.special.eval_laguerre</code>	-
<code>scipy.special.eval_legendre</code>	-
<code>scipy.special.eval_sh_chebyt</code>	-
<code>scipy.special.eval_sh_chebyu</code>	-
<code>scipy.special.eval_sh_jacobi</code>	-
<code>scipy.special.eval_sh_legendre</code>	-
<code>scipy.special.exp1</code>	-
<code>scipy.special.exp10</code>	-
<code>scipy.special.exp2</code>	-
<code>scipy.special.expi</code>	-

continues on next page

Table 149 – continued from previous page

SciPy	CuPy
<code>scipy.special.expit</code>	-
<code>scipy.special.expm1</code>	-
<code>scipy.special.expn</code>	-
<code>scipy.special.exprel</code>	-
<code>scipy.special.factorial</code>	-
<code>scipy.special.factorial2</code>	-
<code>scipy.special.factorialk</code>	-
<code>scipy.special.fdtr</code>	-
<code>scipy.special.fdtrc</code>	-
<code>scipy.special.fdtri</code>	-
<code>scipy.special.fdtridfd</code>	-
<code>scipy.special.fresnel</code>	-
<code>scipy.special.fresnel_zeros</code>	-
<code>scipy.special.fresnelc_zeros</code>	-
<code>scipy.special.fresnels_zeros</code>	-
<code>scipy.special.gamma</code>	<code>cupyx.scipy.special.gamma</code>
<code>scipy.special.gammainc</code>	-
<code>scipy.special.gammaincc</code>	-
<code>scipy.special.gammainccinv</code>	-
<code>scipy.special.gammaincinv</code>	-
<code>scipy.special.gammaln</code>	<code>cupyx.scipy.special.gammaln</code>
<code>scipy.special.gammasgn</code>	-
<code>scipy.special.gdtr</code>	-
<code>scipy.special.gdtrc</code>	-
<code>scipy.special.gdtria</code>	-
<code>scipy.special.gdtrib</code>	-
<code>scipy.special.gdtrix</code>	-
<code>scipy.special.gegenbauer</code>	-
<code>scipy.special.genlaguerre</code>	-
<code>scipy.special.geterr</code>	-
<code>scipy.special.h1vp</code>	-
<code>scipy.special.h2vp</code>	-
<code>scipy.special.h_roots</code>	-
<code>scipy.special.hankel1</code>	-
<code>scipy.special.hankel1e</code>	-
<code>scipy.special.hankel2</code>	-
<code>scipy.special.hankel2e</code>	-
<code>scipy.special.he_roots</code>	-
<code>scipy.special.hermite</code>	-
<code>scipy.special.hermitenorm</code>	-
<code>scipy.special.huber</code>	<code>cupyx.scipy.special.huber</code>
<code>scipy.special.hyp0f1</code>	-
<code>scipy.special.hyp1f1</code>	-
<code>scipy.special.hyp2f1</code>	-
<code>scipy.special.hyperu</code>	-
<code>scipy.special.i0</code>	<code>cupyx.scipy.special.i0</code>
<code>scipy.special.i0e</code>	-
<code>scipy.special.i1</code>	<code>cupyx.scipy.special.i1</code>
<code>scipy.special.i1e</code>	-

continues on next page

Table 149 – continued from previous page

SciPy	CuPy
<code>scipy.special.inv_boxcox</code>	-
<code>scipy.special.inv_boxcox1p</code>	-
<code>scipy.special.it2i0k0</code>	-
<code>scipy.special.it2j0y0</code>	-
<code>scipy.special.it2struve0</code>	-
<code>scipy.special.itairy</code>	-
<code>scipy.special.iti0k0</code>	-
<code>scipy.special.itj0y0</code>	-
<code>scipy.special.itmodstruve0</code>	-
<code>scipy.special.itstruve0</code>	-
<code>scipy.special.iv</code>	-
<code>scipy.special.ive</code>	-
<code>scipy.special.ivp</code>	-
<code>scipy.special.j0</code>	<code>cupyx.scipy.special.j0</code>
<code>scipy.special.j1</code>	<code>cupyx.scipy.special.j1</code>
<code>scipy.special.j_roots</code>	-
<code>scipy.special.jacobi</code>	-
<code>scipy.special.jn</code>	-
<code>scipy.special.jn_zeros</code>	-
<code>scipy.special.jnjnp_zeros</code>	-
<code>scipy.special.jnp_zeros</code>	-
<code>scipy.special.jnyn_zeros</code>	-
<code>scipy.special.js_roots</code>	-
<code>scipy.special.jv</code>	-
<code>scipy.special.jve</code>	-
<code>scipy.special.jvp</code>	-
<code>scipy.special.k0</code>	-
<code>scipy.special.k0e</code>	-
<code>scipy.special.k1</code>	-
<code>scipy.special.k1e</code>	-
<code>scipy.special.kei</code>	-
<code>scipy.special.kei_zeros</code>	-
<code>scipy.special.keip</code>	-
<code>scipy.special.keip_zeros</code>	-
<code>scipy.special.kelvin</code>	-
<code>scipy.special.kelvin_zeros</code>	-
<code>scipy.special.ker</code>	-
<code>scipy.special.ker_zeros</code>	-
<code>scipy.special.kerp</code>	-
<code>scipy.special.kerp_zeros</code>	-
<code>scipy.special.kl_div</code>	<code>cupyx.scipy.special.kl_div</code>
<code>scipy.special.kn</code>	-
<code>scipy.special.kolmogi</code>	-
<code>scipy.special.kolmogorov</code>	-
<code>scipy.special.kv</code>	-
<code>scipy.special.kve</code>	-
<code>scipy.special.kvp</code>	-
<code>scipy.special.l_roots</code>	-
<code>scipy.special.la_roots</code>	-

continues on next page



Table 149 – continued from previous page

SciPy	CuPy
<code>scipy.special.laguerre</code>	-
<code>scipy.special.lambertw</code>	-
<code>scipy.special.legendre</code>	-
<code>scipy.special.lmbda</code>	-
<code>scipy.special.log1p</code>	-
<code>scipy.special.log_ndtr</code>	-
<code>scipy.special.log_softmax</code>	-
<code>scipy.special.loggamma</code>	-
<code>scipy.special.logit</code>	-
<code>scipy.special.logsumexp</code>	-
<code>scipy.special.lpmn</code>	-
<code>scipy.special.lpmv</code>	-
<code>scipy.special.lpn</code>	-
<code>scipy.special.lqmn</code>	-
<code>scipy.special.lqn</code>	-
<code>scipy.special.mathieu_a</code>	-
<code>scipy.special.mathieu_b</code>	-
<code>scipy.special.mathieu_cem</code>	-
<code>scipy.special.mathieu_even_coef</code>	-
<code>scipy.special.mathieu_modcem1</code>	-
<code>scipy.special.mathieu_modcem2</code>	-
<code>scipy.special.mathieu_modsem1</code>	-
<code>scipy.special.mathieu_modsem2</code>	-
<code>scipy.special.mathieu_odd_coef</code>	-
<code>scipy.special.mathieu_sem</code>	-
<code>scipy.special.modfresnelm</code>	-
<code>scipy.special.modfresnelp</code>	-
<code>scipy.special.modstruve</code>	-
<code>scipy.special.multigammaln</code>	-
<code>scipy.special.nbdtr</code>	-
<code>scipy.special.nbdtrc</code>	-
<code>scipy.special.nbdtri</code>	-
<code>scipy.special.nbdtrik</code>	-
<code>scipy.special.nbdtrin</code>	-
<code>scipy.special.ncfdtr</code>	-
<code>scipy.special.ncfdtri</code>	-
<code>scipy.special.ncfdtridfd</code>	-
<code>scipy.special.ncfdtridfn</code>	-
<code>scipy.special.ncfdtrinc</code>	-
<code>scipy.special.nctdtr</code>	-
<code>scipy.special.nctdtridf</code>	-
<code>scipy.special.nctdtrinc</code>	-
<code>scipy.special.nctdtrit</code>	-
<code>scipy.special.ndtr</code>	<i><code>cupyx.scipy.special.ndtr</code></i>
<code>scipy.special.ndtri</code>	-
<code>scipy.special.nrdtrimn</code>	-
<code>scipy.special.nrdtrisd</code>	-
<code>scipy.special.obl_ang1</code>	-
<code>scipy.special.obl_ang1_cv</code>	-

continues on next page

Table 149 – continued from previous page

SciPy	CuPy
<code>scipy.special.obl_cv</code>	-
<code>scipy.special.obl_cv_seq</code>	-
<code>scipy.special.obl_rad1</code>	-
<code>scipy.special.obl_rad1_cv</code>	-
<code>scipy.special.obl_rad2</code>	-
<code>scipy.special.obl_rad2_cv</code>	-
<code>scipy.special.owens_t</code>	-
<code>scipy.special.p_roots</code>	-
<code>scipy.special.pbdn_seq</code>	-
<code>scipy.special.pbdv</code>	-
<code>scipy.special.pbdv_seq</code>	-
<code>scipy.special.pbvv</code>	-
<code>scipy.special.pbvv_seq</code>	-
<code>scipy.special.pbwa</code>	-
<code>scipy.special.pdtr</code>	-
<code>scipy.special.pdtrc</code>	-
<code>scipy.special.pdtri</code>	-
<code>scipy.special.pdtrik</code>	-
<code>scipy.special.perm</code>	-
<code>scipy.special.poch</code>	-
<code>scipy.special.polygamma</code>	<code>cupyx.scipy.special.polygamma</code>
<code>scipy.special.pro_ang1</code>	-
<code>scipy.special.pro_ang1_cv</code>	-
<code>scipy.special.pro_cv</code>	-
<code>scipy.special.pro_cv_seq</code>	-
<code>scipy.special.pro_rad1</code>	-
<code>scipy.special.pro_rad1_cv</code>	-
<code>scipy.special.pro_rad2</code>	-
<code>scipy.special.pro_rad2_cv</code>	-
<code>scipy.special.ps_roots</code>	-
<code>scipy.special.pseudo_huber</code>	<code>cupyx.scipy.special.pseudo_huber</code>
<code>scipy.special.psi</code>	-
<code>scipy.special.radian</code>	-
<code>scipy.special.rel_entr</code>	<code>cupyx.scipy.special.rel_entr</code>
<code>scipy.special.rgamma</code>	-
<code>scipy.special.riccati_jn</code>	-
<code>scipy.special.riccati_yn</code>	-
<code>scipy.special.roots_chebys</code>	-
<code>scipy.special.roots_chebys</code>	-
<code>scipy.special.roots_chebyt</code>	-
<code>scipy.special.roots_chebyu</code>	-
<code>scipy.special.roots_gegenbauer</code>	-
<code>scipy.special.roots_genlaguerre</code>	-
<code>scipy.special.roots_hermite</code>	-
<code>scipy.special.roots_hermitenorm</code>	-
<code>scipy.special.roots_jacobi</code>	-
<code>scipy.special.roots_laguerre</code>	-
<code>scipy.special.roots_legendre</code>	-
<code>scipy.special.roots_sh_chebyt</code>	-

continues on next page

Table 149 – continued from previous page

SciPy	CuPy
<code>scipy.special.roots_sh_chebyu</code>	-
<code>scipy.special.roots_sh_jacobi</code>	-
<code>scipy.special.roots_sh_legendre</code>	-
<code>scipy.special.round</code>	-
<code>scipy.special.s_roots</code>	-
<code>scipy.special.seterr</code>	-
<code>scipy.special.sh_chebyt</code>	-
<code>scipy.special.sh_chebyu</code>	-
<code>scipy.special.sh_jacobi</code>	-
<code>scipy.special.sh_legendre</code>	-
<code>scipy.special.shichi</code>	-
<code>scipy.special.sici</code>	-
<code>scipy.special.sinc</code>	-
<code>scipy.special.sindg</code>	-
<code>scipy.special.smirnov</code>	-
<code>scipy.special.smirnovi</code>	-
<code>scipy.special.softmax</code>	-
<code>scipy.special.spence</code>	-
<code>scipy.special.sph_harm</code>	-
<code>scipy.special.spherical_in</code>	-
<code>scipy.special.spherical_jn</code>	-
<code>scipy.special.spherical_kn</code>	-
<code>scipy.special.spherical_yn</code>	-
<code>scipy.special.stdtr</code>	-
<code>scipy.special.stdtridf</code>	-
<code>scipy.special.stdtrit</code>	-
<code>scipy.special.struve</code>	-
<code>scipy.special.t_roots</code>	-
<code>scipy.special.tandg</code>	-
<code>scipy.special.tklmbda</code>	-
<code>scipy.special.ts_roots</code>	-
<code>scipy.special.u_roots</code>	-
<code>scipy.special.us_roots</code>	-
<code>scipy.special.voigt_profile</code>	-
<code>scipy.special.wofz</code>	-
<code>scipy.special.wrightomega</code>	-
<code>scipy.special.xlog1py</code>	-
<code>scipy.special.xlogy</code>	-
<code>scipy.special.y0</code>	<code>cupyx.scipy.special.y0</code>
<code>scipy.special.y0_zeros</code>	-
<code>scipy.special.y1</code>	<code>cupyx.scipy.special.y1</code>
<code>scipy.special.y1_zeros</code>	-
<code>scipy.special.ylp_zeros</code>	-
<code>scipy.special.yn</code>	-
<code>scipy.special.yn_zeros</code>	-
<code>scipy.special.ynp_zeros</code>	-
<code>scipy.special.yv</code>	-
<code>scipy.special.yve</code>	-
<code>scipy.special.yvp</code>	-

continues on next page

Table 149 – continued from previous page

SciPy	CuPy
<code>scipy.special.zeta</code>	<code>cupyx.scipy.special.zeta</code>
<code>scipy.special.zetac</code>	-

## CONTRIBUTION GUIDE

This is a guide for all contributions to CuPy. The development of CuPy is running on [the official repository at GitHub](#). Anyone that wants to register an issue or to send a pull request should read through this document.

### 6.1 Classification of Contributions

There are several ways to contribute to CuPy community:

1. Registering an issue
2. Sending a pull request (PR)
3. Sending a question to [CuPy's Gitter channel](#), [CuPy User Group](#), or [StackOverflow](#)
4. Open-sourcing an external example
5. Writing a post about CuPy

This document mainly focuses on 1 and 2, though other contributions are also appreciated.

### 6.2 Development Cycle

This section explains the development process of CuPy. Before contributing to CuPy, it is strongly recommended to understand the development cycle.

#### 6.2.1 Versioning

The versioning of CuPy follows [PEP 440](#) and a part of [Semantic versioning](#). The version number consists of three or four parts: `X.Y.Zw` where `X` denotes the **major version**, `Y` denotes the **minor version**, `Z` denotes the **revision number**, and the optional `w` denotes the pre-release suffix. While the major, minor, and revision numbers follow the rule of semantic versioning, the pre-release suffix follows PEP 440 so that the version string is much friendly with Python eco-system.

**Note that a major update basically does not contain compatibility-breaking changes from the last release candidate (RC).** This is not a strict rule, though; if there is a critical API bug that we have to fix for the major version, we may add breaking changes to the major version up.

As for the backward compatibility, see [API Compatibility Policy](#).

## 6.2.2 Release Cycle

The first one is the track of **stable versions**, which is a series of revision updates for the latest major version. The second one is the track of **development versions**, which is a series of pre-releases for the upcoming major version.

Consider that `X.0.0` is the latest major version and `Y.0.0`, `Z.0.0` are the succeeding major versions. Then, the timeline of the updates is depicted by the following table.

Date	ver X	ver Y	ver Z
0 weeks	X.0.0rc1	–	–
4 weeks	X.0.0	Y.0.0a1	–
8 weeks	X.1.0*	Y.0.0b1	–
12 weeks	X.2.0*	Y.0.0rc1	–
16 weeks	–	Y.0.0	Z.0.0a1

(\* These might be revision releases)

The dates shown in the left-most column are relative to the release of `X.0.0rc1`. In particular, each revision/minor release is made four weeks after the previous one of the same major version, and the pre-release of the upcoming major version is made at the same time. Whether these releases are revision or minor is determined based on the contents of each update.

Note that there are only three stable releases for the versions `X.x.x`. During the parallel development of `Y.0.0` and `Z.0.0a1`, the version Y is treated as an **almost-stable version** and Z is treated as a development version.

If there is a critical bug found in `X.x.x` after stopping the development of version X, we may release a hot-fix for this version at any time.

We create a milestone for each upcoming release at GitHub. The GitHub milestone is basically used for collecting the issues and PRs resolved in the release.

## 6.2.3 Git Branches

The `master` branch is used to develop pre-release versions. It means that **alpha, beta, and RC updates are developed at the master branch**. This branch contains the most up-to-date source tree that includes features newly added after the latest major version.

The stable version is developed at the individual branch named as `vN` where “N” reflects the version number (we call it a *versioned branch*). For example, `v1.0.0`, `v1.0.1`, and `v1.0.2` will be developed at the `v1` branch.

**Notes for contributors:** When you send a pull request, you basically have to send it to the `master` branch. If the change can also be applied to the stable version, a core team member will apply the same change to the stable version so that the change is also included in the next revision update.

If the change is only applicable to the stable version and not to the `master` branch, please send it to the versioned branch. We basically only accept changes to the latest versioned branch (where the stable version is developed) unless the fix is critical.

If you want to make a new feature of the `master` branch available in the current stable version, please send a *backport PR* to the stable version (the latest `vN` branch). See the next section for details.

*Note: a change that can be applied to both branches should be sent to the master branch. Each release of the stable version is also merged to the development version so that the change is also reflected to the next major version.*

## 6.2.4 Feature Backport PRs

We basically do not backport any new features of the development version to the stable versions. If you desire to include the feature to the current stable version and you can work on the backport work, we welcome such a contribution. In such a case, you have to send a backport PR to the latest vN branch. **Note that we do not accept any feature backport PRs to older versions because we are not running quality assurance workflows (e.g. CI) for older versions so that we cannot ensure that the PR is correctly ported.**

There are some rules on sending a backport PR.

- Start the PR title from the prefix **[backport]**.
- Clarify the original PR number in the PR description (something like “This is a backport of #XXXX”).
- (optional) Write to the PR description the motivation of backporting the feature to the stable version.

Please follow these rules when you create a feature backport PR.

Note: PRs that do not include any changes/additions to APIs (e.g. bug fixes, documentation improvements) are usually backported by core dev members. It is also appreciated to make such a backport PR by any contributors, though, so that the overall development proceeds more smoothly!

## 6.3 Issues and Pull Requests

In this section, we explain how to file issues and send pull requests (PRs).

### 6.3.1 Issue/PR Labels

Issues and PRs are labeled by the following tags:

- **Bug:** bug reports (issues) and bug fixes (PRs)
- **Enhancement:** implementation improvements without breaking the interface
- **Feature:** feature requests (issues) and their implementations (PRs)
- **NoCompat:** disrupts backward compatibility
- **Test:** test fixes and updates
- **Document:** document fixes and improvements
- **Example:** fixes and improvements on the examples
- **Install:** fixes installation script
- **Contribution-Welcome:** issues that we request for contribution (only issues are categorized to this)
- **Other:** other issues and PRs

Multiple tags might be labeled to one issue/PR. **Note that revision releases cannot include PRs in Feature and NoCompat categories.**

### 6.3.2 How to File an Issue

On registering an issue, write precise explanations on how you want CuPy to be. Bug reports must include necessary and sufficient conditions to reproduce the bugs. Feature requests must include **what** you want to do (and **why** you want to do, if needed) with CuPy. You can contain your thoughts on **how** to realize it into the feature requests, though **what** part is most important for discussions.

**Warning:** If you have a question on usages of CuPy, it is highly recommended to send a post to [CuPy's Gitter channel](#), [CuPy User Group](#) or [StackOverflow](#) instead of the issue tracker. The issue tracker is not a place to share knowledge on practices. We may suggest these places and immediately close how-to question issues.

### 6.3.3 How to Send a Pull Request

If you can write code to fix an issue, we encourage to send a PR.

First of all, before starting to write any code, do not forget to confirm the following points.

- Read through the [Coding Guidelines](#) and [Unit Testing](#).
- Check the appropriate branch that you should send the PR following [Git Branches](#). If you do not have any idea about selecting a branch, please choose the `master` branch.

In particular, **check the branch before writing any code**. The current source tree of the chosen branch is the starting point of your change.

After writing your code (**including unit tests and hopefully documentations!**), send a PR on GitHub. You have to write a precise explanation of **what** and **how** you fix; it is the first documentation of your code that developers read, which is a very important part of your PR.

Once you send a PR, it is automatically tested on GitHub Actions. After the automatic test passes, core developers will start reviewing your code. Note that this automatic PR test only includes CPU tests.

---

**Note:** We are also running continuous integration with GPU tests for the `master` branch and the versioned branch of the latest major version. Since this service is currently running on our internal server, we do not use it for automatic PR tests to keep the server secure.

---

If you are planning to add a new feature or modify existing APIs, **it is recommended to open an issue and discuss the design first**. The design discussion needs lower cost for the core developers than code review. Following the consequences of the discussions, you can send a PR that is smoothly reviewed in a shorter time.

Even if your code is not complete, you can send a pull request as a *work-in-progress PR* by putting the [WIP] prefix to the PR title. If you write a precise explanation about the PR, core developers and other contributors can join the discussion about how to proceed the PR. WIP PR is also useful to have discussions based on a concrete code.



## 6.4 Coding Guidelines

**Note:** Coding guidelines are updated at v5.0. Those who have contributed to older versions should read the guidelines again.

We use [PEP8](#) and a part of [OpenStack Style Guidelines](#) related to general coding style as our basic style guidelines.

You can use `autopep8` and `flake8` commands to check your code.

In order to avoid confusion from using different tool versions, we pin the versions of those tools. Install them with the following command (from within the top directory of CuPy repository):

```
$ pip install -e '[stylecheck]'
```

And check your code with:

```
$ autopep8 path/to/your/code.py
$ flake8 path/to/your/code.py
```

To check Cython code, use `.flake8.cython` configuration file:

```
$ flake8 --config=.flake8.cython path/to/your/cython/code.pyx
```

The `autopep8` supports automatically correct Python code to conform to the PEP 8 style guide:

```
$ autopep8 --in-place path/to/your/code.py
```

The `flake8` command lets you know the part of your code not obeying our style guidelines. Before sending a pull request, be sure to check that your code passes the `flake8` checking.

Note that `flake8` command is not perfect. It does not check some of the style guidelines. Here is a (not-complete) list of the rules that `flake8` cannot check.

- Relative imports are prohibited. [H304]
- Importing non-module symbols is prohibited.
- Import statements must be organized into three parts: standard libraries, third-party libraries, and internal imports. [H306]

In addition, we restrict the usage of *shortcut symbols* in our code base. They are symbols imported by packages and sub-packages of `cupy`. For example, `cupy.cuda.Device` is a shortcut of `cupy.cuda.device.Device`. **It is not allowed to use such shortcuts in the ``cupy`` library implementation.** Note that you can still use them in `tests` and `examples` directories.

Once you send a pull request, your coding style is automatically checked by *GitHub Actions*. The reviewing process starts after the check passes.

The CuPy is designed based on NumPy's API design. CuPy's source code and documents contain the original NumPy ones. Please note the followings when writing the document.

- In order to identify overlapping parts, it is preferable to add some remarks that this document is just copied or altered from the original one. It is also preferable to briefly explain the specification of the function in a short paragraph, and refer to the corresponding function in NumPy so that users can read the detailed document. However, it is possible to include a complete copy of the document with such a remark if users cannot summarize in such a way.

- If a function in CuPy only implements a limited amount of features in the original one, users should explicitly describe only what is implemented in the document.

For changes that modify or add new Cython files, please make sure the pointer types follow these guidelines ([#1913](#)).

- Pointers should be `void*` if only used within Cython, or `intptr_t` if exposed to the Python space.
- Memory sizes should be `size_t`.
- Memory offsets should be `ptrdiff_t`.

---

**Note:** We are incrementally enforcing the above rules, so some existing code may not follow the above guidelines, but please ensure all new contributions do.

---

## 6.5 Unit Testing

Testing is one of the most important part of your code. You must write test cases and verify your implementation by following our testing guide.

Note that we are using `pytest` and `mock` package for testing, so install them before writing your code:

```
$ pip install pytest mock
```

### 6.5.1 How to Run Tests

In order to run unit tests at the repository root, you first have to build Cython files in place by running the following command:

```
$ pip install -e .
```

---

**Note:** When you modify `*.pxd` files, before running `pip install -e .`, you must clean `*.cpp` and `*.so` files once with the following command, because Cython does not automatically rebuild those files nicely:

```
$ git clean -fdx
```

---

Once Cython modules are built, you can run unit tests by running the following command at the repository root:

```
$ python -m pytest
```

CUDA must be installed to run unit tests.

Some GPU tests require cuDNN to run. In order to skip unit tests that require cuDNN, specify `-m='not cudnn'` option:

```
$ python -m pytest path/to/your/test.py -m='not cudnn'
```

Some GPU tests involve multiple GPUs. If you want to run GPU tests with insufficient number of GPUs, specify the number of available GPUs to `CUPY_TEST_GPU_LIMIT`. For example, if you have only one GPU, launch `pytest` by the following command to skip multi-GPU tests:

```
$ export CUPY_TEST_GPU_LIMIT=1
$ python -m pytest path/to/gpu/test.py
```

Following this naming convention, you can run all the tests by running the following command at the repository root:

```
$ python -m pytest
```

Or you can also specify a root directory to search test scripts from:

```
$ python -m pytest tests/cupy_tests      # to just run tests of CuPy
$ python -m pytest tests/install_tests  # to just run tests of installation modules
```

If you modify the code related to existing unit tests, you must run appropriate commands.

## 6.5.2 Test File and Directory Naming Conventions

Tests are put into the `tests/cupy_tests` directory. In order to enable test runner to find test scripts correctly, we are using special naming convention for the test subdirectories and the test scripts.

- The name of each subdirectory of `tests` must end with the `_tests` suffix.
- The name of each test script must start with the `test_` prefix.

When we write a test for a module, we use the appropriate path and file name for the test script whose correspondence to the tested module is clear. For example, if you want to write a test for a module `cupy.x.y.z`, the test script must be located at `tests/cupy_tests/x_tests/y_tests/test_z.py`.

## 6.5.3 How to Write Tests

There are many examples of unit tests under the `tests` directory, so reading some of them is a good and recommended way to learn how to write tests for CuPy. They simply use the `unittest` package of the standard library, while some tests are using utilities from `cupy.testing`.

In addition to the *Coding Guidelines* mentioned above, the following rules are applied to the test code:

- All test classes must inherit from `unittest.TestCase`.
- Use `unittest` features to write tests, except for the following cases:
  - Use `assert` statement instead of `self.assert*` methods (e.g., write `assert x == 1` instead of `self.assertEqual(x, 1)`).
  - Use `with pytest.raises(...):` instead of `with self.assertRaises(...):`.

---

**Note:** We are incrementally applying the above style. Some existing tests may be using the old style (`self.assertRaises`, etc.), but all newly written tests should follow the above style.

---

Even if your patch includes GPU-related code, your tests should not fail without GPU capability. Test functions that require CUDA must be tagged by the `cupy.testing.attr.gpu`:

```
import unittest
from cupy.testing import attr

class TestMyFunc(unittest.TestCase):
```

(continues on next page)

(continued from previous page)

```
...

@attr.gpu
def test_my_gpu_func(self):
    ...
```

The functions tagged by the `gpu` decorator are skipped if `CUPY_TEST_GPU_LIMIT=0` environment variable is set. We also have the `cupy.testing.attr.cudnn` decorator to let `pytest` know that the test depends on cuDNN. The test functions decorated by `cudnn` are skipped if `-m='not cudnn'` is given.

The test functions decorated by `gpu` must not depend on multiple GPUs. In order to write tests for multiple GPUs, use `cupy.testing.attr.multi_gpu()` decorators instead:

```
import unittest
from cupy.testing import attr

class TestMyFunc(unittest.TestCase):
    ...

    @attr.multi_gpu(2) # specify the number of required GPUs here
    def test_my_two_gpu_func(self):
        ...
```

If your test requires too much time, add `cupy.testing.attr.slow` decorator. The test functions decorated by `slow` are skipped if `-m='not slow'` is given:

```
import unittest
from cupy.testing import attr

class TestMyFunc(unittest.TestCase):
    ...

    @attr.slow
    def test_my_slow_func(self):
        ...
```

---

**Note:** If you want to specify more than two attributes, use and operator like `-m='not cudnn and not slow'`. See detail in the [document of pytest](#).

---

Once you send a pull request, [Travis-CI](#) automatically checks if your code meets our coding guidelines described above. Since Travis-CI does not support CUDA, we cannot run unit tests automatically. The reviewing process starts after the automatic check passes. Note that reviewers will test your code without the option to check CUDA-related code.

---

**Note:** Some of numerically unstable tests might cause errors irrelevant to your changes. In such a case, we ignore the failures and go on to the review process, so do not worry about it!

---

## 6.6 Documentation

When adding a new feature to the framework, you also need to document it in the reference.

---

**Note:** If you are unsure about how to fix the documentation, you can submit a pull request without doing so. Reviewers will help you fix the documentation appropriately.

---

The documentation source is stored under `docs` directory and written in `reStructuredText` format.

To build the documentation, you need to install `Sphinx`:

```
$ pip install -r docs/requirements.txt
```

Then you can build the documentation in HTML format locally:

```
$ cd docs
$ make html
```

HTML files are generated under `build/html` directory. Open `index.html` with the browser and see if it is rendered as expected.

---

**Note:** Docstrings (documentation comments in the source code) are collected from the installed CuPy module. If you modified docstrings, make sure to install the module (e.g., using `pip install -e .`) before building the documentation.

---

## 6.7 Tips for Developers

Here are some tips for developers hacking CuPy source code.

### 6.7.1 Install as Editable

During the development we recommend using `pip` with `-e` option to install as editable mode:

```
$ pip install -e .
```

Please note that even with `-e`, you will have to rerun `pip install -e .` to regenerate C++ sources using Cython if you modified Cython source files (e.g., `*.pyx` files).

### 6.7.2 Use `ccache`

NVCC environment variable can be specified at the build time to use the custom command instead of `nvcc`. You can speed up the rebuild using `ccache` (v3.4 or later) by:

```
$ export NVCC='ccache nvcc'
```

### 6.7.3 Limit Architecture

Use `CUPY_NVCC_GENERATE_CODE` environment variable to reduce the build time by limiting the target CUDA architectures. For example, if you only run your CuPy build with NVIDIA P100 and V100, you can use:

```
$ export CUPY_NVCC_GENERATE_CODE=arch=compute_60,code=sm_60;arch=compute_70,code=sm_70
```

See *Environment variables* for the description.

## UPGRADE GUIDE

This is a list of changes introduced in each release that users should be aware of when migrating from older versions.

### 7.1 CuPy v9

#### 7.1.1 Dropping Support of CUDA 9.0

CUDA 9.0 is no longer supported. Use CUDA 9.2 or later.

#### 7.1.2 Dropping Support of cuDNN v7.5 and NCCL v2.3

cuDNN v7.5 (or earlier) and NCCL v2.3 (or earlier) are no longer supported.

#### 7.1.3 Dropping Support of NumPy 1.16 and SciPy 1.3

NumPy 1.16 and SciPy 1.3 are no longer supported.

#### 7.1.4 Dropping Support of Python 3.5

Python 3.5 is no longer supported in CuPy v9.

#### 7.1.5 NCCL and cuDNN No Longer Included in Wheels

NCCL and cuDNN shared libraires are no longer included in wheels (see [#4850](#) for discussions). You can manually install them after installing wheel if you don't have a previous installation; see *Installation* for details.

#### 7.1.6 Baseline API Changes

Baseline API has been bumped from NumPy 1.19 and SciPy 1.5 to NumPy 1.20 and SciPy 1.6. CuPy v9 will follow the upstream products' specifications of these baseline versions.

Following NumPy 1.20, aliases for the Python scalar types (`cupy.bool`, `cupy.int`, `cupy.float`, and `cupy.complex`) are now deprecated. `cupy.bool_`, `cupy.int_`, `cupy.float_` and `cupy.complex_` should be used instead when required.

## 7.1.7 Update of Docker Images

CuPy official Docker images (see [Installation](#) for details) are now updated to use CUDA 11.2 and Python 3.8.

## 7.2 CuPy v8

### 7.2.1 Dropping Support of CUDA 8.0 and 9.1

CUDA 8.0 and 9.1 are no longer supported. Use CUDA 9.0, 9.2, 10.0, or later.

### 7.2.2 Dropping Support of NumPy 1.15 and SciPy 1.2

NumPy 1.15 (or earlier) and SciPy 1.2 (or earlier) are no longer supported.

### 7.2.3 Update of Docker Images

- CuPy official Docker images (see [Installation](#) for details) are now updated to use CUDA 10.2 and Python 3.6.
- SciPy and Optuna are now pre-installed.

### 7.2.4 CUB Support and Compiler Requirement

CUB module is now built by default. You can enable the use of CUB by setting `CUPY_ACCELERATORS="cub"` (see [Environment variables](#) for details).

Due to this change, g++-6 or later is required when building CuPy from the source. See [Installation](#) for details.

The following environment variables are no longer effective:

- `CUB_DISABLED`: Use `CUPY_ACCELERATORS` as aforementioned.
- `CUB_PATH`: No longer required as CuPy uses either the CUB source bundled with CUDA (only when using CUDA 11.0 or later) or the one in the CuPy distribution.

### 7.2.5 API Changes

- `cupy.scatter_add`, which was deprecated in CuPy v4, has been removed. Use `cupyx.scatter_add()` instead.
- `cupy.sparse` module has been deprecated and will be removed in future releases. Use `cupyx.scipy.sparse` instead.
- `dtype` argument of `cupy.ndarray.min()` and `cupy.ndarray.max()` has been removed to align with the NumPy specification.
- `cupy.allclose()` now returns the result as 0-dim GPU array instead of Python bool to avoid device synchronization.
- `cupy.RawModule` now delays the compilation to the time of the first call to align the behavior with `cupy.RawKernel`.
- `cupy.cuda.*_enabled` flags (`nccl_enabled`, `nvtx_enabled`, etc.) has been deprecated. Use `cupy.cuda.*.available` flag (`cupy.cuda.nccl.available`, `cupy.cuda.nvtx.available`, etc.) instead.



- CHAINER\_SEED environment variable is no longer effective. Use CUPY\_SEED instead.

## 7.3 CuPy v7

### 7.3.1 Dropping Support of Python 2.7 and 3.4

Starting from CuPy v7, Python 2.7 and 3.4 are no longer supported as it reaches its end-of-life (EOL) in January 2020 (2.7) and March 2019 (3.4). Python 3.5.1 is the minimum Python version supported by CuPy v7. Please upgrade the Python version if you are using affected versions of Python to any later versions listed under [Installation](#).

## 7.4 CuPy v6

### 7.4.1 Binary Packages Ignore LD\_LIBRARY\_PATH

Prior to CuPy v6, LD\_LIBRARY\_PATH environment variable can be used to override cuDNN / NCCL libraries bundled in the binary distribution (also known as wheels). In CuPy v6, LD\_LIBRARY\_PATH will be ignored during discovery of cuDNN / NCCL; CuPy binary distributions always use libraries that comes with the package to avoid errors caused by unexpected override.

## 7.5 CuPy v5

### 7.5.1 `cupyx.scipy` Namespace

`cupyx.scipy` namespace has been introduced to provide CUDA-enabled SciPy functions. `cupy.sparse` module has been renamed to `cupyx.scipy.sparse`; `cupy.sparse` will be kept as an alias for backward compatibility.

### 7.5.2 Dropped Support for CUDA 7.0 / 7.5

CuPy v5 no longer supports CUDA 7.0 / 7.5.

### 7.5.3 Update of Docker Images

CuPy official Docker images (see [Installation](#) for details) are now updated to use CUDA 9.2 and cuDNN 7.

To use these images, you may need to upgrade the NVIDIA driver on your host. See [Requirements of nvidia-docker](#) for details.

## 7.6 CuPy v4

**Note:** The version number has been bumped from v2 to v4 to align with the versioning of Chainer. Therefore, CuPy v3 does not exist.

---

### 7.6.1 Default Memory Pool

Prior to CuPy v4, memory pool was only enabled by default when CuPy is used with Chainer. In CuPy v4, memory pool is now enabled by default, even when you use CuPy without Chainer. The memory pool significantly improves the performance by mitigating the overhead of memory allocation and CPU/GPU synchronization.

**Attention:** When you monitor GPU memory usage (e.g., using `nvidia-smi`), you may notice that GPU memory not being freed even after the array instance become out of scope. This is expected behavior, as the default memory pool “caches” the allocated memory blocks.

To access the default memory pool instance, use `get_default_memory_pool()` and `get_default_pinned_memory_pool()`. You can access the statistics and free all unused memory blocks “cached” in the memory pool.

```
import cupy
a = cupy.ndarray(100, dtype=cupy.float32)
mempool = cupy.get_default_memory_pool()

# For performance, the size of actual allocation may become larger than the requested
# array size.
print(mempool.used_bytes())    # 512
print(mempool.total_bytes())   # 512

# Even if the array goes out of scope, its memory block is kept in the pool.
a = None
print(mempool.used_bytes())    # 0
print(mempool.total_bytes())   # 512

# You can clear the memory block by calling `free_all_blocks`.
mempool.free_all_blocks()
print(mempool.used_bytes())    # 0
print(mempool.total_bytes())   # 0
```

You can even disable the default memory pool by the code below. Be sure to do this before any other CuPy operations.

```
import cupy
cupy.cuda.set_allocator(None)
cupy.cuda.set_pinned_memory_allocator(None)
```

## 7.6.2 Compute Capability

CuPy v4 now requires NVIDIA GPU with Compute Capability 3.0 or larger. See the [List of CUDA GPUs](#) to check if your GPU supports Compute Capability 3.0.

## 7.6.3 CUDA Stream

As CUDA Stream is fully supported in CuPy v4, `cupy.cuda.RandomState.set_stream`, the function to change the stream used by the random number generator, has been removed. Please use `cupy.cuda.Stream.use()` instead.

See the discussion in [#306](#) for more details.

## 7.6.4 cupyx Namespace

`cupyx` namespace has been introduced to provide features specific to CuPy (i.e., features not provided in NumPy) while avoiding collision in future. See *CuPy-specific functions* for the list of such functions.

For this rule, `cupy.scatter_add()` has been moved to `cupyx.scatter_add()`. `cupy.scatter_add()` is still available as an alias, but it is encouraged to use `cupyx.scatter_add()` instead.

## 7.6.5 Update of Docker Images

CuPy official Docker images (see [Installation](#) for details) are now updated to use CUDA 8.0 and cuDNN 6.0. This change was introduced because CUDA 7.5 does not support NVIDIA Pascal GPUs.

To use these images, you may need to upgrade the NVIDIA driver on your host. See [Requirements of nvidia-docker](#) for details.

# 7.7 CuPy v2

## 7.7.1 Changed Behavior of `count_nonzero` Function

For performance reasons, `cupy.count_nonzero()` has been changed to return zero-dimensional ndarray instead of `int` when `axis=None`. See the discussion in [#154](#) for more details.



## LICENSE

Copyright (c) 2015 Preferred Infrastructure, Inc.

Copyright (c) 2015 Preferred Networks, Inc.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

### 8.1 NumPy

The CuPy is designed based on NumPy’s API. CuPy’s source code and documents contain the original NumPy ones.

Copyright (c) 2005-2016, NumPy Developers.

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the NumPy Developers nor the names of any contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,

PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

## 8.2 SciPy

The CuPy is designed based on SciPy's API. CuPy's source code and documents contain the original SciPy ones.

Copyright (c) 2001, 2002 Enthought, Inc.

All rights reserved.

Copyright (c) 2003-2016 SciPy Developers.

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- a. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- b. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- c. Neither the name of Enthought nor the names of the SciPy Developers may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDERS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

## PYTHON MODULE INDEX

### C

- `cupy`, ??
- `cupy.fft`, 118
- `cupy.linalg`, 148
- `cupy.polynomial.polynomial`, 184
- `cupy.polynomial.polyutils`, 185
- `cupy.random`, 191
- `cupy.testing`, 255
- `cupyx.optimizing`, 418
- `cupyx.scipy`, 272
  - `cupyx.scipy.fft`, 272
  - `cupyx.scipy.fftpack`, 283
  - `cupyx.scipy.linalg`, 290
  - `cupyx.scipy.ndimage`, 301
  - `cupyx.scipy.signal`, 404
  - `cupyx.scipy.sparse`, 344
    - `cupyx.scipy.sparse.linalg`, 387
  - `cupyx.scipy.special`, 398
  - `cupyx.scipy.stats`, 412





## Symbols

- `_JitRawKernel` (class in `cupyx.jit._interface`), 486
- `__bool__()` (`cupy.ndarray` method), 61
- `__bool__()` (`cupyx.scipy.sparse.coo_matrix` method), 351
- `__bool__()` (`cupyx.scipy.sparse.csc_matrix` method), 359
- `__bool__()` (`cupyx.scipy.sparse.csr_matrix` method), 368
- `__bool__()` (`cupyx.scipy.sparse.dia_matrix` method), 374
- `__bool__()` (`cupyx.scipy.sparse.spmatrix` method), 379
- `__call__()` (`cupy.ElementwiseKernel` method), 476
- `__call__()` (`cupy.RawKernel` method), 479
- `__call__()` (`cupy.ReductionKernel` method), 478
- `__call__()` (`cupy.poly1d` method), 187
- `__call__()` (`cupy.prof.TimeRangeDecorator` method), 489
- `__call__()` (`cupy.ufunc` method), 66
- `__call__()` (`cupy.vectorize` method), 131
- `__call__()` (`cupyx.jit._interface._JitRawKernel` method), 486
- `__call__()` (`cupyx.scipy.sparse.linalg.LinearOperator` method), 388
- `__copy__()` (`cupy.ndarray` method), 53
- `__enter__()` (`cupy.cuda.Device` method), 420
- `__enter__()` (`cupy.cuda.ExternalStream` method), 449
- `__enter__()` (`cupy.cuda.MemoryHook` method), 441
- `__enter__()` (`cupy.cuda.Stream` method), 447
- `__enter__()` (`cupy.cuda.memory_hooks.DebugPrintHook` method), 443
- `__enter__()` (`cupy.cuda.memory_hooks.LineProfileHook` method), 445
- `__enter__()` (`cupy.fft.config.set_cufft_callbacks` method), 129
- `__enter__()` (`cupy.prof.TimeRangeDecorator` method), 489
- `__eq__()` (`cupy.ElementwiseKernel` method), 476
- `__eq__()` (`cupy.RawKernel` method), 480
- `__eq__()` (`cupy.RawModule` method), 483
- `__eq__()` (`cupy.ReductionKernel` method), 478
- `__eq__()` (`cupy.broadcast` method), 103
- `__eq__()` (`cupy.cuda.CFunctionAllocator` method), 439
- `__eq__()` (`cupy.cuda.Device` method), 420
- `__eq__()` (`cupy.cuda.Event` method), 451
- `__eq__()` (`cupy.cuda.ExternalStream` method), 450
- `__eq__()` (`cupy.cuda.ManagedMemory` method), 425
- `__eq__()` (`cupy.cuda.Memory` method), 423
- `__eq__()` (`cupy.cuda.MemoryAsync` method), 424
- `__eq__()` (`cupy.cuda.MemoryAsyncPool` method), 437
- `__eq__()` (`cupy.cuda.MemoryHook` method), 442
- `__eq__()` (`cupy.cuda.MemoryPointer` method), 430
- `__eq__()` (`cupy.cuda.MemoryPool` method), 435
- `__eq__()` (`cupy.cuda.PinnedMemory` method), 427
- `__eq__()` (`cupy.cuda.PinnedMemoryPointer` method), 431
- `__eq__()` (`cupy.cuda.PinnedMemoryPool` method), 437
- `__eq__()` (`cupy.cuda.PythonFunctionAllocator` method), 438
- `__eq__()` (`cupy.cuda.Stream` method), 448
- `__eq__()` (`cupy.cuda.UnownedMemory` method), 426
- `__eq__()` (`cupy.cuda.memory_hooks.DebugPrintHook` method), 444
- `__eq__()` (`cupy.cuda.memory_hooks.LineProfileHook` method), 446
- `__eq__()` (`cupy.cuda.nccl.NcclCommunicator` method), 463
- `__eq__()` (`cupy.cuda.texture.CUDAarray` method), 454
- `__eq__()` (`cupy.cuda.texture.ChannelFormatDescriptor` method), 453
- `__eq__()` (`cupy.cuda.texture.ResourceDescriptor` method), 456
- `__eq__()` (`cupy.cuda.texture.SurfaceObject` method), 458
- `__eq__()` (`cupy.cuda.texture.TextureDescriptor` method), 457
- `__eq__()` (`cupy.cuda.texture.TextureObject` method), 458
- `__eq__()` (`cupy.cuda.texture.TextureReference` method), 459
- `__eq__()` (`cupy.fft.config.set_cufft_callbacks` method), 129
- `__eq__()` (`cupy.flatiter` method), 144
- `__eq__()` (`cupy.ndarray` method), 61

- `__eq__()` (*cupy.poly1d* method), 187
- `__eq__()` (*cupy.prof.TimeRangeDecorator* method), 489
- `__eq__()` (*cupy.random.BitGenerator* method), 196
- `__eq__()` (*cupy.random.Generator* method), 195
- `__eq__()` (*cupy.random.MRG32k3a* method), 199
- `__eq__()` (*cupy.random.Philox4x3210* method), 200
- `__eq__()` (*cupy.random.RandomState* method), 208
- `__eq__()` (*cupy.random.XORWOW* method), 197
- `__eq__()` (*cupy.ufunc* method), 66
- `__eq__()` (*cupy.vectorize* method), 131
- `__eq__()` (*cupyx.jit.\_interface.\_JitRawKernel* method), 487
- `__eq__()` (*cupyx.scipy.sparse.coo\_matrix* method), 350
- `__eq__()` (*cupyx.scipy.sparse.csc\_matrix* method), 359
- `__eq__()` (*cupyx.scipy.sparse.csr\_matrix* method), 368
- `__eq__()` (*cupyx.scipy.sparse.dia\_matrix* method), 374
- `__eq__()` (*cupyx.scipy.sparse.linalg.LinearOperator* method), 389
- `__eq__()` (*cupyx.scipy.sparse.spmatrix* method), 378
- `__exit__()` (*cupy.cuda.Device* method), 420
- `__exit__()` (*cupy.cuda.ExternalStream* method), 449
- `__exit__()` (*cupy.cuda.MemoryHook* method), 441
- `__exit__()` (*cupy.cuda.Stream* method), 447
- `__exit__()` (*cupy.cuda.memory\_hooks.DebugPrintHook* method), 443
- `__exit__()` (*cupy.cuda.memory\_hooks.LineProfileHook* method), 445
- `__exit__()` (*cupy.fft.config.set\_cufft\_callbacks* method), 129
- `__exit__()` (*cupy.prof.TimeRangeDecorator* method), 489
- `__ge__()` (*cupy.ElementwiseKernel* method), 476
- `__ge__()` (*cupy.RawKernel* method), 480
- `__ge__()` (*cupy.RawModule* method), 483
- `__ge__()` (*cupy.ReductionKernel* method), 478
- `__ge__()` (*cupy.broadcast* method), 103
- `__ge__()` (*cupy.cuda.CFunctionAllocator* method), 439
- `__ge__()` (*cupy.cuda.Device* method), 420
- `__ge__()` (*cupy.cuda.Event* method), 451
- `__ge__()` (*cupy.cuda.ExternalStream* method), 450
- `__ge__()` (*cupy.cuda.ManagedMemory* method), 425
- `__ge__()` (*cupy.cuda.Memory* method), 423
- `__ge__()` (*cupy.cuda.MemoryAsync* method), 424
- `__ge__()` (*cupy.cuda.MemoryAsyncPool* method), 437
- `__ge__()` (*cupy.cuda.MemoryHook* method), 442
- `__ge__()` (*cupy.cuda.MemoryPointer* method), 430
- `__ge__()` (*cupy.cuda.MemoryPool* method), 435
- `__ge__()` (*cupy.cuda.PinnedMemory* method), 427
- `__ge__()` (*cupy.cuda.PinnedMemoryPointer* method), 431
- `__ge__()` (*cupy.cuda.PinnedMemoryPool* method), 438
- `__ge__()` (*cupy.cuda.PythonFunctionAllocator* method), 438
- `__ge__()` (*cupy.cuda.Stream* method), 448
- `__ge__()` (*cupy.cuda.UnownedMemory* method), 426
- `__ge__()` (*cupy.cuda.memory\_hooks.DebugPrintHook* method), 444
- `__ge__()` (*cupy.cuda.memory\_hooks.LineProfileHook* method), 446
- `__ge__()` (*cupy.cuda.nccl.NcclCommunicator* method), 464
- `__ge__()` (*cupy.cuda.texture.CUDAarray* method), 454
- `__ge__()` (*cupy.cuda.texture.ChannelFormatDescriptor* method), 453
- `__ge__()` (*cupy.cuda.texture.ResourceDescriptor* method), 456
- `__ge__()` (*cupy.cuda.texture.SurfaceObject* method), 458
- `__ge__()` (*cupy.cuda.texture.TextureDescriptor* method), 457
- `__ge__()` (*cupy.cuda.texture.TextureObject* method), 458
- `__ge__()` (*cupy.cuda.texture.TextureReference* method), 459
- `__ge__()` (*cupy.fft.config.set\_cufft\_callbacks* method), 129
- `__ge__()` (*cupy.flatiter* method), 145
- `__ge__()` (*cupy.ndarray* method), 61
- `__ge__()` (*cupy.poly1d* method), 187
- `__ge__()` (*cupy.prof.TimeRangeDecorator* method), 489
- `__ge__()` (*cupy.random.BitGenerator* method), 196
- `__ge__()` (*cupy.random.Generator* method), 196
- `__ge__()` (*cupy.random.MRG32k3a* method), 199
- `__ge__()` (*cupy.random.Philox4x3210* method), 200
- `__ge__()` (*cupy.random.RandomState* method), 208
- `__ge__()` (*cupy.random.XORWOW* method), 198
- `__ge__()` (*cupy.ufunc* method), 66
- `__ge__()` (*cupy.vectorize* method), 132
- `__ge__()` (*cupyx.jit.\_interface.\_JitRawKernel* method), 487
- `__ge__()` (*cupyx.scipy.sparse.coo\_matrix* method), 351
- `__ge__()` (*cupyx.scipy.sparse.csc\_matrix* method), 359
- `__ge__()` (*cupyx.scipy.sparse.csr\_matrix* method), 368
- `__ge__()` (*cupyx.scipy.sparse.dia\_matrix* method), 374
- `__ge__()` (*cupyx.scipy.sparse.linalg.LinearOperator* method), 389
- `__ge__()` (*cupyx.scipy.sparse.spmatrix* method), 379
- `__getitem__()` (*cupy.flatiter* method), 144
- `__getitem__()` (*cupy.ndarray* method), 52
- `__getitem__()` (*cupy.poly1d* method), 187
- `__getitem__()` (*cupyx.jit.\_interface.\_JitRawKernel* method), 487
- `__getitem__()` (*cupyx.scipy.sparse.csc\_matrix* method), 352
- `__getitem__()` (*cupyx.scipy.sparse.csr\_matrix* method), 361
- `__gt__()` (*cupy.ElementwiseKernel* method), 476
- `__gt__()` (*cupy.RawKernel* method), 480

- `__gt__()` (*cupy.RawModule* method), 483
- `__gt__()` (*cupy.ReductionKernel* method), 478
- `__gt__()` (*cupy.broadcast* method), 103
- `__gt__()` (*cupy.cuda.CFunctionAllocator* method), 439
- `__gt__()` (*cupy.cuda.Device* method), 420
- `__gt__()` (*cupy.cuda.Event* method), 451
- `__gt__()` (*cupy.cuda.ExternalStream* method), 450
- `__gt__()` (*cupy.cuda.ManagedMemory* method), 425
- `__gt__()` (*cupy.cuda.Memory* method), 423
- `__gt__()` (*cupy.cuda.MemoryAsync* method), 424
- `__gt__()` (*cupy.cuda.MemoryAsyncPool* method), 437
- `__gt__()` (*cupy.cuda.MemoryHook* method), 442
- `__gt__()` (*cupy.cuda.MemoryPointer* method), 430
- `__gt__()` (*cupy.cuda.MemoryPool* method), 435
- `__gt__()` (*cupy.cuda.PinnedMemory* method), 427
- `__gt__()` (*cupy.cuda.PinnedMemoryPointer* method), 431
- `__gt__()` (*cupy.cuda.PinnedMemoryPool* method), 437
- `__gt__()` (*cupy.cuda.PythonFunctionAllocator* method), 438
- `__gt__()` (*cupy.cuda.Stream* method), 448
- `__gt__()` (*cupy.cuda.UnownedMemory* method), 426
- `__gt__()` (*cupy.cuda.memory\_hooks.DebugPrintHook* method), 444
- `__gt__()` (*cupy.cuda.memory\_hooks.LineProfileHook* method), 446
- `__gt__()` (*cupy.cuda.nccl.NcclCommunicator* method), 464
- `__gt__()` (*cupy.cuda.texture.CUDAarray* method), 454
- `__gt__()` (*cupy.cuda.texture.ChannelFormatDescriptor* method), 453
- `__gt__()` (*cupy.cuda.texture.ResourceDescriptor* method), 456
- `__gt__()` (*cupy.cuda.texture.SurfaceObject* method), 458
- `__gt__()` (*cupy.cuda.texture.TextureDescriptor* method), 457
- `__gt__()` (*cupy.cuda.texture.TextureObject* method), 458
- `__gt__()` (*cupy.cuda.texture.TextureReference* method), 459
- `__gt__()` (*cupy.fft.config.set\_cufft\_callbacks* method), 129
- `__gt__()` (*cupy.flatiter* method), 145
- `__gt__()` (*cupy.ndarray* method), 61
- `__gt__()` (*cupy.poly1d* method), 187
- `__gt__()` (*cupy.prof.TimeRangeDecorator* method), 489
- `__gt__()` (*cupy.random.BitGenerator* method), 196
- `__gt__()` (*cupy.random.Generator* method), 196
- `__gt__()` (*cupy.random.MRG32k3a* method), 199
- `__gt__()` (*cupy.random.Philox4x3210* method), 200
- `__gt__()` (*cupy.random.RandomState* method), 208
- `__gt__()` (*cupy.random.XORWOW* method), 198
- `__gt__()` (*cupy.ufunc* method), 66
- `__gt__()` (*cupy.vectorize* method), 131
- `__gt__()` (*cupyx.jit.\_interface.\_JitRawKernel* method), 487
- `__gt__()` (*cupyx.scipy.sparse.coo\_matrix* method), 351
- `__gt__()` (*cupyx.scipy.sparse.csc\_matrix* method), 359
- `__gt__()` (*cupyx.scipy.sparse.csr\_matrix* method), 368
- `__gt__()` (*cupyx.scipy.sparse.dia\_matrix* method), 374
- `__gt__()` (*cupyx.scipy.sparse.linalg.LinearOperator* method), 389
- `__gt__()` (*cupyx.scipy.sparse.spmatrix* method), 378
- `__iter__()` (*cupy.flatiter* method), 144
- `__iter__()` (*cupy.ndarray* method), 53
- `__iter__()` (*cupy.poly1d* method), 187
- `__iter__()` (*cupyx.scipy.sparse.coo\_matrix* method), 345
- `__iter__()` (*cupyx.scipy.sparse.csc\_matrix* method), 352
- `__iter__()` (*cupyx.scipy.sparse.csr\_matrix* method), 361
- `__iter__()` (*cupyx.scipy.sparse.dia\_matrix* method), 370
- `__iter__()` (*cupyx.scipy.sparse.spmatrix* method), 375
- `__le__()` (*cupy.ElementwiseKernel* method), 476
- `__le__()` (*cupy.RawKernel* method), 480
- `__le__()` (*cupy.RawModule* method), 483
- `__le__()` (*cupy.ReductionKernel* method), 478
- `__le__()` (*cupy.broadcast* method), 103
- `__le__()` (*cupy.cuda.CFunctionAllocator* method), 439
- `__le__()` (*cupy.cuda.Device* method), 420
- `__le__()` (*cupy.cuda.Event* method), 451
- `__le__()` (*cupy.cuda.ExternalStream* method), 450
- `__le__()` (*cupy.cuda.ManagedMemory* method), 425
- `__le__()` (*cupy.cuda.Memory* method), 423
- `__le__()` (*cupy.cuda.MemoryAsync* method), 424
- `__le__()` (*cupy.cuda.MemoryAsyncPool* method), 437
- `__le__()` (*cupy.cuda.MemoryHook* method), 442
- `__le__()` (*cupy.cuda.MemoryPointer* method), 430
- `__le__()` (*cupy.cuda.MemoryPool* method), 435
- `__le__()` (*cupy.cuda.PinnedMemory* method), 427
- `__le__()` (*cupy.cuda.PinnedMemoryPointer* method), 431
- `__le__()` (*cupy.cuda.PinnedMemoryPool* method), 437
- `__le__()` (*cupy.cuda.PythonFunctionAllocator* method), 438
- `__le__()` (*cupy.cuda.Stream* method), 448
- `__le__()` (*cupy.cuda.UnownedMemory* method), 426
- `__le__()` (*cupy.cuda.memory\_hooks.DebugPrintHook* method), 444
- `__le__()` (*cupy.cuda.memory\_hooks.LineProfileHook* method), 446
- `__le__()` (*cupy.cuda.nccl.NcclCommunicator* method), 464
- `__le__()` (*cupy.cuda.texture.CUDAarray* method), 454

- `__le__()` (`cupy.cuda.texture.ChannelFormatDescriptor` method), 453
- `__le__()` (`cupy.cuda.texture.ResourceDescriptor` method), 456
- `__le__()` (`cupy.cuda.texture.SurfaceObject` method), 458
- `__le__()` (`cupy.cuda.texture.TextureDescriptor` method), 457
- `__le__()` (`cupy.cuda.texture.TextureObject` method), 458
- `__le__()` (`cupy.cuda.texture.TextureReference` method), 459
- `__le__()` (`cupy.fft.config.set_cufft_callbacks` method), 129
- `__le__()` (`cupy.flatiter` method), 145
- `__le__()` (`cupy.ndarray` method), 61
- `__le__()` (`cupy.poly1d` method), 187
- `__le__()` (`cupy.prof.TimeRangeDecorator` method), 489
- `__le__()` (`cupy.random.BitGenerator` method), 196
- `__le__()` (`cupy.random.Generator` method), 196
- `__le__()` (`cupy.random.MRG32k3a` method), 199
- `__le__()` (`cupy.random.Philox4x3210` method), 200
- `__le__()` (`cupy.random.RandomState` method), 208
- `__le__()` (`cupy.random.XORWOW` method), 198
- `__le__()` (`cupy.ufunc` method), 66
- `__le__()` (`cupy.vectorize` method), 131
- `__le__()` (`cupyx.jit._interface._JitRawKernel` method), 487
- `__le__()` (`cupyx.scipy.sparse.coo_matrix` method), 351
- `__le__()` (`cupyx.scipy.sparse.csc_matrix` method), 359
- `__le__()` (`cupyx.scipy.sparse.csr_matrix` method), 368
- `__le__()` (`cupyx.scipy.sparse.dia_matrix` method), 374
- `__le__()` (`cupyx.scipy.sparse.linalg.LinearOperator` method), 389
- `__le__()` (`cupyx.scipy.sparse.spmatrix` method), 378
- `__len__()` (`cupy.flatiter` method), 144
- `__len__()` (`cupy.ndarray` method), 53
- `__len__()` (`cupy.poly1d` method), 187
- `__len__()` (`cupyx.scipy.sparse.coo_matrix` method), 345
- `__len__()` (`cupyx.scipy.sparse.csc_matrix` method), 352
- `__len__()` (`cupyx.scipy.sparse.csr_matrix` method), 361
- `__len__()` (`cupyx.scipy.sparse.dia_matrix` method), 370
- `__len__()` (`cupyx.scipy.sparse.spmatrix` method), 375
- `__lt__()` (`cupy.ElementwiseKernel` method), 476
- `__lt__()` (`cupy.RawKernel` method), 480
- `__lt__()` (`cupy.RawModule` method), 483
- `__lt__()` (`cupy.ReductionKernel` method), 478
- `__lt__()` (`cupy.broadcast` method), 103
- `__lt__()` (`cupy.cuda.CFunctionAllocator` method), 439
- `__lt__()` (`cupy.cuda.Device` method), 420
- `__lt__()` (`cupy.cuda.Event` method), 451
- `__lt__()` (`cupy.cuda.ExternalStream` method), 450
- `__lt__()` (`cupy.cuda.ManagedMemory` method), 425
- `__lt__()` (`cupy.cuda.Memory` method), 423
- `__lt__()` (`cupy.cuda.MemoryAsync` method), 424
- `__lt__()` (`cupy.cuda.MemoryAsyncPool` method), 437
- `__lt__()` (`cupy.cuda.MemoryHook` method), 442
- `__lt__()` (`cupy.cuda.MemoryPointer` method), 430
- `__lt__()` (`cupy.cuda.MemoryPool` method), 435
- `__lt__()` (`cupy.cuda.PinnedMemory` method), 427
- `__lt__()` (`cupy.cuda.PinnedMemoryPointer` method), 431
- `__lt__()` (`cupy.cuda.PinnedMemoryPool` method), 437
- `__lt__()` (`cupy.cuda.PythonFunctionAllocator` method), 438
- `__lt__()` (`cupy.cuda.Stream` method), 448
- `__lt__()` (`cupy.cuda.UnownedMemory` method), 426
- `__lt__()` (`cupy.cuda.memory_hooks.DebugPrintHook` method), 444
- `__lt__()` (`cupy.cuda.memory_hooks.LineProfileHook` method), 446
- `__lt__()` (`cupy.cuda.nccl.NcclCommunicator` method), 463
- `__lt__()` (`cupy.cuda.texture.CUDAarray` method), 454
- `__lt__()` (`cupy.cuda.texture.ChannelFormatDescriptor` method), 453
- `__lt__()` (`cupy.cuda.texture.ResourceDescriptor` method), 456
- `__lt__()` (`cupy.cuda.texture.SurfaceObject` method), 458
- `__lt__()` (`cupy.cuda.texture.TextureDescriptor` method), 457
- `__lt__()` (`cupy.cuda.texture.TextureObject` method), 458
- `__lt__()` (`cupy.cuda.texture.TextureReference` method), 459
- `__lt__()` (`cupy.fft.config.set_cufft_callbacks` method), 129
- `__lt__()` (`cupy.flatiter` method), 144
- `__lt__()` (`cupy.ndarray` method), 61
- `__lt__()` (`cupy.poly1d` method), 187
- `__lt__()` (`cupy.prof.TimeRangeDecorator` method), 489
- `__lt__()` (`cupy.random.BitGenerator` method), 196
- `__lt__()` (`cupy.random.Generator` method), 195
- `__lt__()` (`cupy.random.MRG32k3a` method), 199
- `__lt__()` (`cupy.random.Philox4x3210` method), 200
- `__lt__()` (`cupy.random.RandomState` method), 208
- `__lt__()` (`cupy.random.XORWOW` method), 197
- `__lt__()` (`cupy.ufunc` method), 66
- `__lt__()` (`cupy.vectorize` method), 131
- `__lt__()` (`cupyx.jit._interface._JitRawKernel` method), 487
- `__lt__()` (`cupyx.scipy.sparse.coo_matrix` method), 351
- `__lt__()` (`cupyx.scipy.sparse.csc_matrix` method), 359
- `__lt__()` (`cupyx.scipy.sparse.csr_matrix` method), 368
- `__lt__()` (`cupyx.scipy.sparse.dia_matrix` method), 374
- `__lt__()` (`cupyx.scipy.sparse.linalg.LinearOperator` method), 389



- `__lt__()` (*cupyx.scipy.sparse.spmatrix* method), 378
  - `__ne__()` (*cupy.ElementwiseKernel* method), 476
  - `__ne__()` (*cupy.RawKernel* method), 480
  - `__ne__()` (*cupy.RawModule* method), 483
  - `__ne__()` (*cupy.ReductionKernel* method), 478
  - `__ne__()` (*cupy.broadcast* method), 103
  - `__ne__()` (*cupy.cuda.CFunctionAllocator* method), 439
  - `__ne__()` (*cupy.cuda.Device* method), 420
  - `__ne__()` (*cupy.cuda.Event* method), 451
  - `__ne__()` (*cupy.cuda.ExternalStream* method), 450
  - `__ne__()` (*cupy.cuda.ManagedMemory* method), 425
  - `__ne__()` (*cupy.cuda.Memory* method), 423
  - `__ne__()` (*cupy.cuda.MemoryAsync* method), 424
  - `__ne__()` (*cupy.cuda.MemoryAsyncPool* method), 437
  - `__ne__()` (*cupy.cuda.MemoryHook* method), 442
  - `__ne__()` (*cupy.cuda.MemoryPointer* method), 430
  - `__ne__()` (*cupy.cuda.MemoryPool* method), 435
  - `__ne__()` (*cupy.cuda.PinnedMemory* method), 427
  - `__ne__()` (*cupy.cuda.PinnedMemoryPointer* method), 431
  - `__ne__()` (*cupy.cuda.PinnedMemoryPool* method), 437
  - `__ne__()` (*cupy.cuda.PythonFunctionAllocator* method), 438
  - `__ne__()` (*cupy.cuda.Stream* method), 448
  - `__ne__()` (*cupy.cuda.UnownedMemory* method), 426
  - `__ne__()` (*cupy.cuda.memory\_hooks.DebugPrintHook* method), 444
  - `__ne__()` (*cupy.cuda.memory\_hooks.LineProfileHook* method), 446
  - `__ne__()` (*cupy.cuda.nccl.NcclCommunicator* method), 463
  - `__ne__()` (*cupy.cuda.texture.CUDAarray* method), 454
  - `__ne__()` (*cupy.cuda.texture.ChannelFormatDescriptor* method), 453
  - `__ne__()` (*cupy.cuda.texture.ResourceDescriptor* method), 456
  - `__ne__()` (*cupy.cuda.texture.SurfaceObject* method), 458
  - `__ne__()` (*cupy.cuda.texture.TextureDescriptor* method), 457
  - `__ne__()` (*cupy.cuda.texture.TextureObject* method), 458
  - `__ne__()` (*cupy.cuda.texture.TextureReference* method), 459
  - `__ne__()` (*cupy.fft.config.set\_cufft\_callbacks* method), 129
  - `__ne__()` (*cupy.flatiter* method), 144
  - `__ne__()` (*cupy.ndarray* method), 61
  - `__ne__()` (*cupy.poly1d* method), 187
  - `__ne__()` (*cupy.prof.TimeRangeDecorator* method), 489
  - `__ne__()` (*cupy.random.BitGenerator* method), 196
  - `__ne__()` (*cupy.random.Generator* method), 195
  - `__ne__()` (*cupy.random.MRG32k3a* method), 199
  - `__ne__()` (*cupy.random.Philox4x3210* method), 200
  - `__ne__()` (*cupy.random.RandomState* method), 208
  - `__ne__()` (*cupy.random.XORWOW* method), 197
  - `__ne__()` (*cupy.ufunc* method), 66
  - `__ne__()` (*cupy.vectorize* method), 131
  - `__ne__()` (*cupyx.jit.\_interface.\_JitRawKernel* method), 487
  - `__ne__()` (*cupyx.scipy.sparse.coo\_matrix* method), 351
  - `__ne__()` (*cupyx.scipy.sparse.csc\_matrix* method), 359
  - `__ne__()` (*cupyx.scipy.sparse.csr\_matrix* method), 368
  - `__ne__()` (*cupyx.scipy.sparse.dia\_matrix* method), 374
  - `__ne__()` (*cupyx.scipy.sparse.linalg.LinearOperator* method), 389
  - `__ne__()` (*cupyx.scipy.sparse.spmatrix* method), 378
  - `__next__()` (*cupy.flatiter* method), 144
  - `__nonzero__()` (*cupyx.scipy.sparse.coo\_matrix* method), 351
  - `__nonzero__()` (*cupyx.scipy.sparse.csc\_matrix* method), 359
  - `__nonzero__()` (*cupyx.scipy.sparse.csr\_matrix* method), 368
  - `__nonzero__()` (*cupyx.scipy.sparse.dia\_matrix* method), 374
  - `__nonzero__()` (*cupyx.scipy.sparse.spmatrix* method), 379
  - `__setitem__()` (*cupy.flatiter* method), 144
  - `__setitem__()` (*cupy.ndarray* method), 52
  - `__setitem__()` (*cupy.poly1d* method), 187
  - `__setitem__()` (*cupyx.scipy.sparse.csc\_matrix* method), 352
  - `__setitem__()` (*cupyx.scipy.sparse.csr\_matrix* method), 361
- ## A
- `A` (*cupyx.scipy.sparse.coo\_matrix* attribute), 351
  - `A` (*cupyx.scipy.sparse.csc\_matrix* attribute), 360
  - `A` (*cupyx.scipy.sparse.csr\_matrix* attribute), 369
  - `A` (*cupyx.scipy.sparse.dia\_matrix* attribute), 375
  - `A` (*cupyx.scipy.sparse.spmatrix* attribute), 379
  - `abort()` (*cupy.cuda.nccl.NcclCommunicator* method), 463
  - `absolute` (in module *cupy*), 70
  - `add` (in module *cupy*), 67
  - `add_callback()` (*cupy.cuda.ExternalStream* method), 449
  - `add_callback()` (*cupy.cuda.Stream* method), 447
  - `adjoint()` (*cupyx.scipy.sparse.linalg.LinearOperator* method), 388
  - `advise()` (*cupy.cuda.ManagedMemory* method), 425
  - `affine_transform()` (in module *cupyx.scipy.ndimage*), 319
  - `all()` (*cupy.ndarray* method), 53
  - `all()` (in module *cupy*), 161
  - `allclose()` (in module *cupy*), 165

[allGather\(\)](#) (*cupy.cuda.nccl.NcclCommunicator method*), 463  
[alloc\(\)](#) (*in module cupy.cuda*), 432  
[alloc\\_pinned\\_memory\(\)](#) (*in module cupy.cuda*), 432  
[alloc\\_postprocess\(\)](#) (*cupy.cuda.memory\_hooks.DebugPrintHook method*), 443  
[alloc\\_postprocess\(\)](#) (*cupy.cuda.memory\_hooks.LineProfileHook method*), 445  
[alloc\\_postprocess\(\)](#) (*cupy.cuda.MemoryHook method*), 441  
[alloc\\_preprocess\(\)](#) (*cupy.cuda.memory\_hooks.DebugPrintHook method*), 443  
[alloc\\_preprocess\(\)](#) (*cupy.cuda.memory\_hooks.LineProfileHook method*), 445  
[alloc\\_preprocess\(\)](#) (*cupy.cuda.MemoryHook method*), 441  
[allow\\_synchronize\(\)](#) (*in module cupyx*), 491  
[allReduce\(\)](#) (*cupy.cuda.nccl.NcclCommunicator method*), 463  
[amax\(\)](#) (*in module cupy*), 243  
[amin\(\)](#) (*in module cupy*), 242  
[angle](#) (*in module cupy*), 176  
[any\(\)](#) (*cupy.ndarray method*), 53  
[any\(\)](#) (*in module cupy*), 161  
[append\(\)](#) (*in module cupy*), 111  
[apply\\_along\\_axis\(\)](#) (*in module cupy*), 131  
[arange\(\)](#) (*in module cupy*), 93  
[arccos](#) (*in module cupy*), 75  
[arccosh](#) (*in module cupy*), 77  
[arcsin](#) (*in module cupy*), 75  
[arcsin\(\)](#) (*cupyx.scipy.sparse.coo\_matrix method*), 345  
[arcsin\(\)](#) (*cupyx.scipy.sparse.csc\_matrix method*), 352  
[arcsin\(\)](#) (*cupyx.scipy.sparse.csr\_matrix method*), 361  
[arcsin\(\)](#) (*cupyx.scipy.sparse.dia\_matrix method*), 370  
[arcsinh](#) (*in module cupy*), 77  
[arcsinh\(\)](#) (*cupyx.scipy.sparse.coo\_matrix method*), 345  
[arcsinh\(\)](#) (*cupyx.scipy.sparse.csc\_matrix method*), 352  
[arcsinh\(\)](#) (*cupyx.scipy.sparse.csr\_matrix method*), 361  
[arcsinh\(\)](#) (*cupyx.scipy.sparse.dia\_matrix method*), 370  
[arctan](#) (*in module cupy*), 75  
[arctan\(\)](#) (*cupyx.scipy.sparse.coo\_matrix method*), 345  
[arctan\(\)](#) (*cupyx.scipy.sparse.csc\_matrix method*), 352  
[arctan\(\)](#) (*cupyx.scipy.sparse.csr\_matrix method*), 361  
[arctan\(\)](#) (*cupyx.scipy.sparse.dia\_matrix method*), 370  
[arctan2](#) (*in module cupy*), 76  
[arctanh](#) (*in module cupy*), 77  
[arctanh\(\)](#) (*cupyx.scipy.sparse.coo\_matrix method*), 345  
[arctanh\(\)](#) (*cupyx.scipy.sparse.csc\_matrix method*), 352  
[arctanh\(\)](#) (*cupyx.scipy.sparse.csr\_matrix method*), 361  
[arctanh\(\)](#) (*cupyx.scipy.sparse.dia\_matrix method*), 370  
[argmax\(\)](#) (*cupy.ndarray method*), 53  
[argmax\(\)](#) (*cupyx.scipy.sparse.csc\_matrix method*), 352  
[argmax\(\)](#) (*cupyx.scipy.sparse.csr\_matrix method*), 361  
[argmax\(\)](#) (*in module cupy*), 238  
[argmin\(\)](#) (*cupy.ndarray method*), 53  
[argmin\(\)](#) (*cupyx.scipy.sparse.csc\_matrix method*), 352  
[argmin\(\)](#) (*cupyx.scipy.sparse.csr\_matrix method*), 362  
[argmin\(\)](#) (*in module cupy*), 239  
[argpartition\(\)](#) (*cupy.ndarray method*), 53  
[argpartition\(\)](#) (*in module cupy*), 237  
[argsort\(\)](#) (*cupy.ndarray method*), 54  
[argsort\(\)](#) (*in module cupy*), 235  
[argwhere\(\)](#) (*in module cupy*), 240  
[around\(\)](#) (*in module cupy*), 168  
[autoextending\\_ptr\(\)](#) (*cupy.cuda.texture.ResourceDescriptor attribute*), 456  
[axis\\_hypot\(\)](#) (*in module cupy*), 63  
[array\\_equal\(\)](#) (*in module cupy*), 166  
[array\\_repr\(\)](#) (*in module cupy*), 147  
[array\\_split\(\)](#) (*in module cupy*), 109  
[array\\_str\(\)](#) (*in module cupy*), 147  
[as\\_series\(\)](#) (*in module cupy.polynomial.polyutils*), 185  
[as\\_strided\(\)](#) (*in module cupy.lib.stride\_tricks*), 141  
[asanyarray\(\)](#) (*in module cupy*), 91  
[asarray\(\)](#) (*in module cupy*), 63  
[ascontiguousarray\(\)](#) (*in module cupy*), 92  
[asformat\(\)](#) (*cupyx.scipy.sparse.coo\_matrix method*), 345  
[asformat\(\)](#) (*cupyx.scipy.sparse.csc\_matrix method*), 353  
[asformat\(\)](#) (*cupyx.scipy.sparse.csr\_matrix method*), 362  
[asformat\(\)](#) (*cupyx.scipy.sparse.dia\_matrix method*), 370  
[asformat\(\)](#) (*cupyx.scipy.sparse.spmatrix method*), 375  
[asfortranarray\(\)](#) (*in module cupy*), 105  
[asfptype\(\)](#) (*cupyx.scipy.sparse.coo\_matrix method*), 346  
[asfptype\(\)](#) (*cupyx.scipy.sparse.csc\_matrix method*), 353  
[asfptype\(\)](#) (*cupyx.scipy.sparse.csr\_matrix method*), 362  
[asfptype\(\)](#) (*cupyx.scipy.sparse.dia\_matrix method*), 370  
[asfptype\(\)](#) (*cupyx.scipy.sparse.spmatrix method*), 375  
[aslinearoperator\(\)](#) (*in module cupyx.scipy.sparse.linalg*), 389  
[asnumpy\(\)](#) (*in module cupy*), 64  
[assert\\_allclose\(\)](#) (*in module cupy.testing*), 257  
[assert\\_array\\_almost\\_equal\(\)](#) (*in module cupy.testing*), 256  
[assert\\_array\\_almost\\_equal\\_nulp\(\)](#) (*in module cupy.testing*), 257  
[assert\\_array\\_equal\(\)](#) (*in module cupy.testing*), 258  
[assert\\_array\\_less\(\)](#) (*in module cupy.testing*), 258

- `assert_array_list_equal()` (in module `cupy.testing`), 258  
`assert_array_max_ulp()` (in module `cupy.testing`), 257  
`astype()` (`cupy.ndarray` method), 54  
`astype()` (`cupyx.scipy.sparse.coo_matrix` method), 346  
`astype()` (`cupyx.scipy.sparse.csc_matrix` method), 353  
`astype()` (`cupyx.scipy.sparse.csr_matrix` method), 362  
`astype()` (`cupyx.scipy.sparse.dia_matrix` method), 370  
`astype()` (`cupyx.scipy.sparse.spmatrix` method), 375  
`atleast_1d()` (in module `cupy`), 102  
`atleast_2d()` (in module `cupy`), 102  
`atleast_3d()` (in module `cupy`), 102  
`attributes` (`cupy.cuda.Device` attribute), 421  
`attributes` (`cupy.RawKernel` attribute), 480  
`average()` (in module `cupy`), 247
- ## B
- `backend` (`cupy.RawKernel` attribute), 480  
`backend` (`cupy.RawModule` attribute), 484  
`bartlett()` (in module `cupy`), 269  
`base` (`cupy.flatiter` attribute), 145  
`base` (`cupy.ndarray` attribute), 61  
`base_repr()` (in module `cupy`), 148  
`bcast()` (`cupy.cuda.nccl.NcclCommunicator` method), 463  
`beta()` (`cupy.random.Generator` method), 192  
`beta()` (`cupy.random.RandomState` method), 201  
`beta()` (in module `cupy.random`), 210  
`binary_closing()` (in module `cupyx.scipy.ndimage`), 332  
`binary_dilation()` (in module `cupyx.scipy.ndimage`), 333  
`binary_erosion()` (in module `cupyx.scipy.ndimage`), 334  
`binary_fill_holes()` (in module `cupyx.scipy.ndimage`), 335  
`binary_hit_or_miss()` (in module `cupyx.scipy.ndimage`), 335  
`binary_opening()` (in module `cupyx.scipy.ndimage`), 336  
`binary_propagation()` (in module `cupyx.scipy.ndimage`), 337  
`binary_repr()` (in module `cupy`), 116  
`binary_version` (`cupy.RawKernel` attribute), 480  
`bincount()` (in module `cupy`), 255  
`binomial()` (`cupy.random.RandomState` method), 201  
`binomial()` (in module `cupy.random`), 210  
`BitGenerator` (class in `cupy.random`), 196  
`bitwise_and` (in module `cupy`), 78  
`bitwise_or` (in module `cupy`), 78  
`bitwise_xor` (in module `cupy`), 79  
`black_tophat()` (in module `cupyx.scipy.ndimage`), 337  
`blackman()` (in module `cupy`), 270  
`block_diag()` (in module `cupyx.scipy.linalg`), 294  
`blockDim` (in module `cupyx.jit`), 485  
`blockIdx` (in module `cupyx.jit`), 485  
`bmat()` (in module `cupyx.scipy.sparse`), 383  
`broadcast` (class in `cupy`), 103  
`broadcast()` (`cupy.cuda.nccl.NcclCommunicator` method), 463  
`broadcast_arrays()` (in module `cupy`), 104  
`broadcast_to()` (in module `cupy`), 104  
`bytes()` (in module `cupy.random`), 211
- ## C
- `c` (`cupy.poly1d` attribute), 188  
`c_` (in module `cupy`), 133  
`cache_mode_ca` (`cupy.RawKernel` attribute), 480  
`cached_code` (`cupyx.jit._interface._JitRawKernel` attribute), 487  
`cached_codes` (`cupyx.jit._interface._JitRawKernel` attribute), 487  
`can_cast()` (in module `cupy`), 117  
`cbrt` (in module `cupy`), 73  
`ceil` (in module `cupy`), 85  
`ceil()` (`cupyx.scipy.sparse.coo_matrix` method), 346  
`ceil()` (`cupyx.scipy.sparse.csc_matrix` method), 353  
`ceil()` (`cupyx.scipy.sparse.csr_matrix` method), 362  
`ceil()` (`cupyx.scipy.sparse.dia_matrix` method), 370  
`center_of_mass()` (in module `cupyx.scipy.ndimage`), 325  
`CFunctionAllocator` (class in `cupy.cuda`), 439  
`cg()` (in module `cupyx.scipy.sparse.linalg`), 392  
`ChannelFormatDescriptor` (class in `cupy.cuda.texture`), 452  
`chDesc` (`cupy.cuda.texture.ResourceDescriptor` attribute), 456  
`check_async_error()` (`cupy.cuda.nccl.NcclCommunicator` method), 463  
`chisquare()` (`cupy.random.RandomState` method), 201  
`chisquare()` (in module `cupy.random`), 211  
`choice()` (`cupy.random.RandomState` method), 201  
`choice()` (in module `cupy.random`), 212  
`cholesky()` (in module `cupy.linalg`), 152  
`choose()` (`cupy.ndarray` method), 54  
`choose()` (in module `cupy`), 139  
`choose_conv_method()` (in module `cupyx.scipy.signal`), 410  
`circulant()` (in module `cupyx.scipy.linalg`), 294  
`clear_memo()` (in module `cupy`), 488  
`clip()` (`cupy.ndarray` method), 54  
`clip()` (in module `cupy`), 178  
`code` (`cupy.RawKernel` attribute), 480  
`code` (`cupy.RawModule` attribute), 484  
`coef` (`cupy.poly1d` attribute), 188  
`coefficients` (`cupy.poly1d` attribute), 188

- `coeffs` (*cupy.poly1d* attribute), 188
- `column_stack()` (in module *cupy*), 108
- `common_type()` (in module *cupy*), 117
- `companion()` (in module *cupyx.scipy.linalg*), 295
- `compile()` (*cupy.RawKernel* method), 480
- `compile()` (*cupy.RawModule* method), 482
- `compress()` (*cupy.ndarray* method), 54
- `compress()` (in module *cupy*), 139
- `compute_capability` (*cupy.cuda.Device* attribute), 421
- `concatenate()` (in module *cupy*), 106
- `conj` (in module *cupy*), 71
- `conj()` (*cupy.ndarray* method), 55
- `conj()` (*cupyx.scipy.sparse.coo\_matrix* method), 346
- `conj()` (*cupyx.scipy.sparse.csc\_matrix* method), 353
- `conj()` (*cupyx.scipy.sparse.csr\_matrix* method), 362
- `conj()` (*cupyx.scipy.sparse.dia\_matrix* method), 370
- `conj()` (*cupyx.scipy.sparse.spmatrix* method), 375
- `conjugate` (in module *cupy*), 71
- `conjugate()` (*cupy.ndarray* method), 55
- `conjugate()` (*cupyx.scipy.sparse.coo\_matrix* method), 346
- `conjugate()` (*cupyx.scipy.sparse.csc\_matrix* method), 353
- `conjugate()` (*cupyx.scipy.sparse.csr\_matrix* method), 362
- `conjugate()` (*cupyx.scipy.sparse.dia\_matrix* method), 371
- `conjugate()` (*cupyx.scipy.sparse.spmatrix* method), 376
- `const_size_bytes` (*cupy.RawKernel* attribute), 480
- `convolution_matrix()` (in module *cupyx.scipy.linalg*), 295
- `convolve()` (in module *cupy*), 177
- `convolve()` (in module *cupyx.scipy.ndimage*), 302
- `convolve()` (in module *cupyx.scipy.signal*), 404
- `convolve1d()` (in module *cupyx.scipy.ndimage*), 302
- `convolve2d()` (in module *cupyx.scipy.signal*), 408
- `coo_matrix` (class in *cupyx.scipy.sparse*), 345
- `copy()` (*cupy.flatiter* method), 144
- `copy()` (*cupy.ndarray* method), 55
- `copy()` (*cupyx.scipy.sparse.coo\_matrix* method), 346
- `copy()` (*cupyx.scipy.sparse.csc\_matrix* method), 353
- `copy()` (*cupyx.scipy.sparse.csr\_matrix* method), 363
- `copy()` (*cupyx.scipy.sparse.dia\_matrix* method), 371
- `copy()` (*cupyx.scipy.sparse.spmatrix* method), 376
- `copy()` (in module *cupy*), 92
- `copy_from()` (*cupy.cuda.MemoryPointer* method), 428
- `copy_from()` (*cupy.cuda.texture.CUDAarray* method), 454
- `copy_from_async()` (*cupy.cuda.MemoryPointer* method), 428
- `copy_from_device()` (*cupy.cuda.MemoryPointer* method), 428
- `copy_from_device_async()` (*cupy.cuda.MemoryPointer* method), 428
- `copy_from_host()` (*cupy.cuda.MemoryPointer* method), 428
- `copy_from_host_async()` (*cupy.cuda.MemoryPointer* method), 429
- `copy_to()` (*cupy.cuda.texture.CUDAarray* method), 454
- `copy_to_host()` (*cupy.cuda.MemoryPointer* method), 429
- `copy_to_host_async()` (*cupy.cuda.MemoryPointer* method), 429
- `copysign` (in module *cupy*), 84
- `copyto()` (in module *cupy*), 98
- `corrcoef()` (in module *cupy*), 250
- `correlate()` (in module *cupy*), 251
- `correlate()` (in module *cupyx.scipy.ndimage*), 303
- `correlate()` (in module *cupyx.scipy.signal*), 405
- `correlate1d()` (in module *cupyx.scipy.ndimage*), 303
- `correlate2d()` (in module *cupyx.scipy.signal*), 409
- `cos` (in module *cupy*), 75
- `cosh` (in module *cupy*), 76
- `count_nonzero()` (*cupyx.scipy.sparse.coo\_matrix* method), 346
- `count_nonzero()` (*cupyx.scipy.sparse.csc\_matrix* method), 354
- `count_nonzero()` (*cupyx.scipy.sparse.csr\_matrix* method), 363
- `count_nonzero()` (*cupyx.scipy.sparse.dia\_matrix* method), 371
- `count_nonzero()` (*cupyx.scipy.sparse.spmatrix* method), 376
- `count_nonzero()` (in module *cupy*), 241
- `cov()` (in module *cupy*), 251
- `cross()` (in module *cupy*), 174
- `csc_matrix` (class in *cupyx.scipy.sparse*), 351
- `csr_matrix` (class in *cupyx.scipy.sparse*), 361
- `cstruct` (*cupy.ndarray* attribute), 61
- `cuArr` (*cupy.cuda.texture.ResourceDescriptor* attribute), 456
- `cublas_handle` (*cupy.cuda.Device* attribute), 421
- `CUDA_PATH`, 493
- `CUDAarray` (class in *cupy.cuda.texture*), 453
- `cumprod()` (*cupy.ndarray* method), 55
- `cumprod()` (in module *cupy*), 171
- `cumsum()` (*cupy.ndarray* method), 55
- `cumsum()` (in module *cupy*), 171
- `cupy`
  - module, 1
- `cupy.fft`
  - module, 118
- `cupy.linalg`
  - module, 148
- `cupy.polynomial.polynomial`
  - module, 184
- `cupy.polynomial.polyutils`
  - module, 185



cupy.random  
     module, 191  
 cupy.testing  
     module, 255  
 CUPY\_ACCELERATORS, 493  
 CUPY\_CACHE\_DIR, 491  
 CUPY\_CACHE\_SAVE\_CUDA\_SOURCE, 491  
 cupyx.optimizing  
     module, 418  
 cupyx.scipy  
     module, 272  
 cupyx.scipy.fft  
     module, 272  
 cupyx.scipy.fftpack  
     module, 283  
 cupyx.scipy.linalg  
     module, 290  
 cupyx.scipy.ndimage  
     module, 301  
 cupyx.scipy.signal  
     module, 404  
 cupyx.scipy.sparse  
     module, 344  
 cupyx.scipy.sparse.linalg  
     module, 387  
 cupyx.scipy.special  
     module, 398  
 cupyx.scipy.stats  
     module, 412  
 cusolver\_handle (*cupy.cuda.Device* attribute), 421  
 cusolver\_sp\_handle (*cupy.cuda.Device* attribute), 421  
 cusparse\_handle (*cupy.cuda.Device* attribute), 421

## D

data (*cupy.ndarray* attribute), 61  
 DebugPrintHook (class in *cupy.cuda.memory\_hooks*), 442  
 default\_rng() (in module *cupy.random*), 191  
 deg2rad (in module *cupy*), 78  
 deg2rad() (*cupyx.scipy.sparse.coo\_matrix* method), 346  
 deg2rad() (*cupyx.scipy.sparse.csc\_matrix* method), 354  
 deg2rad() (*cupyx.scipy.sparse.csr\_matrix* method), 363  
 deg2rad() (*cupyx.scipy.sparse.dia\_matrix* method), 371  
 degrees (in module *cupy*), 77  
 depth (*cupy.cuda.texture.CUDAArray* attribute), 455  
 deriv() (*cupy.poly1d* method), 187  
 desc (*cupy.cuda.texture.CUDAArray* attribute), 455  
 destroy() (*cupy.cuda.nccl.NcclCommunicator* method), 463  
 det() (in module *cupy.linalg*), 155  
 Device (class in *cupy.cuda*), 420  
 device (*cupy.cuda.ManagedMemory* attribute), 425  
 device (*cupy.cuda.Memory* attribute), 423  
 device (*cupy.cuda.MemoryAsync* attribute), 424

device (*cupy.cuda.MemoryPointer* attribute), 430  
 device (*cupy.cuda.UnownedMemory* attribute), 426  
 device (*cupy.ndarray* attribute), 61  
 device (*cupyx.scipy.sparse.coo\_matrix* attribute), 351  
 device (*cupyx.scipy.sparse.csc\_matrix* attribute), 360  
 device (*cupyx.scipy.sparse.csr\_matrix* attribute), 369  
 device (*cupyx.scipy.sparse.dia\_matrix* attribute), 375  
 device (*cupyx.scipy.sparse.spmatrix* attribute), 379  
 device\_id (*cupy.cuda.ManagedMemory* attribute), 425  
 device\_id (*cupy.cuda.Memory* attribute), 423  
 device\_id (*cupy.cuda.MemoryAsync* attribute), 424  
 device\_id (*cupy.cuda.MemoryPointer* attribute), 430  
 device\_id (*cupy.cuda.UnownedMemory* attribute), 426  
 device\_id() (*cupy.cuda.nccl.NcclCommunicator* method), 463  
 deviceCanAccessPeer() (in module *cupy.cuda.runtime*), 469  
 deviceEnablePeerAccess() (in module *cupy.cuda.runtime*), 469  
 deviceGetAttribute() (in module *cupy.cuda.runtime*), 468  
 deviceGetByPCIBusId() (in module *cupy.cuda.runtime*), 468  
 deviceGetDefaultMemPool() (in module *cupy.cuda.runtime*), 469  
 deviceGetLimit() (in module *cupy.cuda.runtime*), 470  
 deviceGetMemPool() (in module *cupy.cuda.runtime*), 469  
 deviceGetPCIBusId() (in module *cupy.cuda.runtime*), 468  
 deviceSetLimit() (in module *cupy.cuda.runtime*), 470  
 deviceSetMemPool() (in module *cupy.cuda.runtime*), 469  
 deviceSynchronize() (in module *cupy.cuda.runtime*), 469  
 DeviceSynchronized, 491  
 dft() (in module *cupyx.scipy.linalg*), 296  
 dia\_matrix (class in *cupyx.scipy.sparse*), 370  
 diag() (in module *cupy*), 96  
 diag\_indices() (in module *cupy*), 137  
 diag\_indices\_from() (in module *cupy*), 138  
 diagflat() (in module *cupy*), 96  
 diagonal() (*cupy.ndarray* method), 55  
 diagonal() (*cupyx.scipy.sparse.coo\_matrix* method), 346  
 diagonal() (*cupyx.scipy.sparse.csc\_matrix* method), 354  
 diagonal() (*cupyx.scipy.sparse.csr\_matrix* method), 363  
 diagonal() (*cupyx.scipy.sparse.dia\_matrix* method), 371  
 diagonal() (*cupyx.scipy.sparse.spmatrix* method), 376  
 diagonal() (in module *cupy*), 140  
 diags() (in module *cupyx.scipy.sparse*), 381

`diff()` (in module `cupy`), 172  
`digamma` (in module `cupyx.scipy.special`), 402  
`digitize()` (in module `cupy`), 255  
`dirichlet()` (`cupy.random.RandomState` method), 201  
`dirichlet()` (in module `cupy.random`), 212  
`divide` (in module `cupy`), 68  
`divmod` (in module `cupy`), 176  
`done` (`cupy.cuda.Event` attribute), 452  
`done` (`cupy.cuda.ExternalStream` attribute), 450  
`done` (`cupy.cuda.Stream` attribute), 449  
`dot()` (`cupy.ndarray` method), 55  
`dot()` (`cupyx.scipy.sparse.coo_matrix` method), 347  
`dot()` (`cupyx.scipy.sparse.csc_matrix` method), 354  
`dot()` (`cupyx.scipy.sparse.csr_matrix` method), 363  
`dot()` (`cupyx.scipy.sparse.dia_matrix` method), 371  
`dot()` (`cupyx.scipy.sparse.linalg.LinearOperator` method), 388  
`dot()` (`cupyx.scipy.sparse.spmatrix` method), 376  
`dot()` (in module `cupy`), 148  
`driverGetVersion()` (in module `cupy.cuda.runtime`), 468  
`dsplit()` (in module `cupy`), 109  
`dstack()` (in module `cupy`), 108  
`dtype` (`cupy.ndarray` attribute), 61  
`dtype` (`cupyx.scipy.sparse.coo_matrix` attribute), 351  
`dtype` (`cupyx.scipy.sparse.csc_matrix` attribute), 360  
`dtype` (`cupyx.scipy.sparse.csr_matrix` attribute), 369  
`dtype` (`cupyx.scipy.sparse.dia_matrix` attribute), 375  
`dump()` (`cupy.ndarray` method), 55  
`dumps()` (`cupy.ndarray` method), 55

## E

`eigh()` (in module `cupy.linalg`), 154  
`eigsh()` (in module `cupyx.scipy.sparse.linalg`), 394  
`eigvalsh()` (in module `cupy.linalg`), 154  
`einsum()` (in module `cupy`), 150  
`ElementwiseKernel` (class in `cupy`), 475  
`eliminate_zeros()` (`cupyx.scipy.sparse.coo_matrix` method), 347  
`eliminate_zeros()` (`cupyx.scipy.sparse.csc_matrix` method), 354  
`eliminate_zeros()` (`cupyx.scipy.sparse.csr_matrix` method), 363  
`empty()` (in module `cupy`), 87  
`empty_like()` (in module `cupy`), 87  
`empty_like_pinned()` (in module `cupyx`), 416  
`empty_pinned()` (in module `cupyx`), 416  
`enable_cooperative_groups` (`cupy.RawKernel` attribute), 480  
`enable_cooperative_groups` (`cupy.RawModule` attribute), 484  
`entr` (in module `cupyx.scipy.special`), 400  
`entropy()` (in module `cupyx.scipy.stats`), 412  
environment variable

`CUDA_PATH`, 491, 493  
`CUPY_ACCELERATORS`, 492, 493  
`CUPY_CACHE_DIR`, 491  
`CUPY_CACHE_IN_MEMORY`, 491  
`CUPY_CACHE_SAVE_CUDA_SOURCE`, 491  
`CUPY_COMPILE_WITH_PTX`, 493  
`CUPY_CUDA_ARRAY_INTERFACE_EXPORT_VERSION`, 492  
`CUPY_CUDA_ARRAY_INTERFACE_SYNC`, 492  
`CUPY_CUDA_COMPILE_WITH_DEBUG`, 492  
`CUPY_CUDA_PER_THREAD_DEFAULT_STREAM`, 492  
`CUPY_DUMP_CUDA_SOURCE_ON_ERROR`, 491  
`CUPY_EXPERIMENTAL_SLICE_COPY`, 492  
`CUPY_GPU_MEMORY_LIMIT`, 492  
`CUPY_INSTALL_USE_HIP`, 493  
`CUPY_NUM_BUILD_JOBS`, 493  
`CUPY_NUM_NVCC_THREADS`, 493  
`CUPY_NVCC_GENERATE_CODE`, 493  
`CUPY_SEED`, 492  
`CUPY_TF32`, 492  
`CUTENSOR_PATH`, 493  
`NVCC`, 492, 493  
`equal` (in module `cupy`), 81  
`erf` (in module `cupyx.scipy.special`), 402  
`erfc` (in module `cupyx.scipy.special`), 403  
`erfcinv` (in module `cupyx.scipy.special`), 403  
`erfcx` (in module `cupyx.scipy.special`), 403  
`erfinv` (in module `cupyx.scipy.special`), 403  
`Event` (class in `cupy.cuda`), 451  
`eventCreate()` (in module `cupy.cuda.runtime`), 474  
`eventCreateWithFlags()` (in module `cupy.cuda.runtime`), 474  
`eventDestroy()` (in module `cupy.cuda.runtime`), 474  
`eventElapsedTime()` (in module `cupy.cuda.runtime`), 474  
`eventQuery()` (in module `cupy.cuda.runtime`), 474  
`eventRecord()` (in module `cupy.cuda.runtime`), 474  
`eventSynchronize()` (in module `cupy.cuda.runtime`), 474  
`exp` (in module `cupy`), 71  
`exp2` (in module `cupy`), 72  
`expand_dims()` (in module `cupy`), 104  
`expm1` (in module `cupy`), 72  
`expm1()` (`cupyx.scipy.sparse.coo_matrix` method), 347  
`expm1()` (`cupyx.scipy.sparse.csc_matrix` method), 354  
`expm1()` (`cupyx.scipy.sparse.csr_matrix` method), 363  
`expm1()` (`cupyx.scipy.sparse.dia_matrix` method), 371  
`exponential()` (`cupy.random.Generator` method), 192  
`exponential()` (`cupy.random.RandomState` method), 201  
`exponential()` (in module `cupy.random`), 213  
`ExternalStream` (class in `cupy.cuda`), 449  
`extract()` (in module `cupy`), 241  
`extrema()` (in module `cupyx.scipy.ndimage`), 325

`eye()` (in module `cupy`), 88

`eye()` (in module `cupyx.scipy.sparse`), 380

## F

`f()` (`cupy.random.RandomState` method), 202

`f()` (in module `cupy.random`), 213

`factorized()` (in module `cupyx.scipy.sparse.linalg`), 391

`fft()` (in module `cupy.fft`), 119

`fft()` (in module `cupyx.scipy.fft`), 272

`fft()` (in module `cupyx.scipy.fftpack`), 284

`fft2()` (in module `cupy.fft`), 120

`fft2()` (in module `cupyx.scipy.fft`), 274

`fft2()` (in module `cupyx.scipy.fftpack`), 285

`fftconvolve()` (in module `cupyx.scipy.signal`), 406

`fftfreq()` (in module `cupy.fft`), 126

`fftfreq()` (in module `cupyx.scipy.fft`), 282

`fftn()` (in module `cupy.fft`), 121

`fftn()` (in module `cupyx.scipy.fft`), 275

`fftn()` (in module `cupyx.scipy.fftpack`), 286

`fftshift()` (in module `cupy.fft`), 126

`fftshift()` (in module `cupyx.scipy.fft`), 281

`fiedler()` (in module `cupyx.scipy.linalg`), 296

`fiedler_companion()` (in module `cupyx.scipy.linalg`), 297

`file_path` (`cupy.RawKernel` attribute), 480

`file_path` (`cupy.RawModule` attribute), 484

`fill()` (`cupy.ndarray` method), 55

`fill_diagonal()` (in module `cupy`), 143

`find()` (in module `cupyx.scipy.sparse`), 386

`fix` (in module `cupy`), 169

`flags` (`cupy.cuda.texture.CUDAArray` attribute), 455

`flags` (`cupy.ndarray` attribute), 61

`flat` (`cupy.ndarray` attribute), 61

`flatiter` (class in `cupy`), 144

`flatnonzero()` (in module `cupy`), 240

`flatten()` (`cupy.ndarray` method), 56

`flip()` (in module `cupy`), 113

`fliplr()` (in module `cupy`), 114

`flipud()` (in module `cupy`), 114

`floor` (in module `cupy`), 85

`floor()` (`cupyx.scipy.sparse.coo_matrix` method), 347

`floor()` (`cupyx.scipy.sparse.csc_matrix` method), 354

`floor()` (`cupyx.scipy.sparse.csr_matrix` method), 363

`floor()` (`cupyx.scipy.sparse.dia_matrix` method), 371

`floor_divide` (in module `cupy`), 69

`fmax` (in module `cupy`), 82

`fmin` (in module `cupy`), 83

`fmod` (in module `cupy`), 70

`for_all_dtypes()` (in module `cupy.testing`), 264

`for_all_dtypes_combination()` (in module `cupy.testing`), 267

`for_CF_orders()` (in module `cupy.testing`), 269

`for_complex_dtypes()` (in module `cupy.testing`), 266

`for_dtypes()` (in module `cupy.testing`), 264

`for_dtypes_combination()` (in module `cupy.testing`), 267

`for_float_dtypes()` (in module `cupy.testing`), 265

`for_int_dtypes()` (in module `cupy.testing`), 266

`for_int_dtypes_combination()` (in module `cupy.testing`), 268

`for_orders()` (in module `cupy.testing`), 269

`for_signed_dtypes()` (in module `cupy.testing`), 266

`for_signed_dtypes_combination()` (in module `cupy.testing`), 267

`for_unsigned_dtypes()` (in module `cupy.testing`), 266

`for_unsigned_dtypes_combination()` (in module `cupy.testing`), 268

`format` (`cupyx.scipy.sparse.coo_matrix` attribute), 351

`format` (`cupyx.scipy.sparse.csc_matrix` attribute), 360

`format` (`cupyx.scipy.sparse.csr_matrix` attribute), 369

`format` (`cupyx.scipy.sparse.dia_matrix` attribute), 375

`fourier_ellipsoid()` (in module `cupyx.scipy.ndimage`), 317

`fourier_gaussian()` (in module `cupyx.scipy.ndimage`), 318

`fourier_shift()` (in module `cupyx.scipy.ndimage`), 318

`fourier_uniform()` (in module `cupyx.scipy.ndimage`), 319

`free()` (`cupy.cuda.PinnedMemoryPool` method), 437

`free()` (in module `cupy.cuda.runtime`), 471

`free_all_blocks()` (`cupy.cuda.MemoryAsyncPool` method), 436

`free_all_blocks()` (`cupy.cuda.MemoryPool` method), 434

`free_all_blocks()` (`cupy.cuda.PinnedMemoryPool` method), 437

`free_all_free()` (`cupy.cuda.MemoryPool` method), 434

`free_bytes()` (`cupy.cuda.MemoryAsyncPool` method), 436

`free_bytes()` (`cupy.cuda.MemoryPool` method), 434

`free_postprocess()` (`cupy.cuda.memory_hooks.DebugPrintHook` method), 443

`free_postprocess()` (`cupy.cuda.memory_hooks.LineProfileHook` method), 445

`free_postprocess()` (`cupy.cuda.MemoryHook` method), 441

`free_preprocess()` (`cupy.cuda.memory_hooks.DebugPrintHook` method), 443

`free_preprocess()` (`cupy.cuda.memory_hooks.LineProfileHook` method), 446

`free_preprocess()` (`cupy.cuda.MemoryHook` method), 441

`freeArray()` (in module `cupy.cuda.runtime`), 471

`freeAsync()` (in module `cupy.cuda.runtime`), 471

`freeHost()` (in module `cupy.cuda.runtime`), 471

`frexp` (in module `cupy`), 85

from\_pci\_bus\_id() (*cupy.cuda.Device* method), 420  
 fromDlpack() (in module *cupy*), 418  
 fromfile() (in module *cupy*), 92  
 full() (in module *cupy*), 90  
 full\_like() (in module *cupy*), 91  
 fuse() (in module *cupy*), 484

## G

gamma (in module *cupyx.scipy.special*), 401  
 gamma() (*cupy.random.Generator* method), 193  
 gamma() (*cupy.random.RandomState* method), 202  
 gamma() (in module *cupy.random*), 214  
 gammaln (in module *cupyx.scipy.special*), 401  
 gaussian\_filter() (in module *cupyx.scipy.ndimage*), 304  
 gaussian\_filter1d() (in module *cupyx.scipy.ndimage*), 305  
 gaussian\_gradient\_magnitude() (in module *cupyx.scipy.ndimage*), 305  
 gaussian\_laplace() (in module *cupyx.scipy.ndimage*), 306  
 gcd (in module *cupy*), 74  
 generate\_binary\_structure() (in module *cupyx.scipy.ndimage*), 338  
 Generator (class in *cupy.random*), 192  
 generator (*cupy.random.MRG32k3a* attribute), 199  
 generator (*cupy.random.Philox4x3210* attribute), 200  
 generator (*cupy.random.XORWOW* attribute), 198  
 generic\_filter() (in module *cupyx.scipy.ndimage*), 307  
 generic\_filter1d() (in module *cupyx.scipy.ndimage*), 308  
 generic\_gradient\_magnitude() (in module *cupyx.scipy.ndimage*), 308  
 generic\_laplace() (in module *cupyx.scipy.ndimage*), 309  
 geometric() (*cupy.random.RandomState* method), 202  
 geometric() (in module *cupy.random*), 214  
 get() (*cupy.ndarray* method), 56  
 get() (*cupy.poly1d* method), 187  
 get() (*cupyx.scipy.sparse.coo\_matrix* method), 347  
 get() (*cupyx.scipy.sparse.csc\_matrix* method), 354  
 get() (*cupyx.scipy.sparse.csr\_matrix* method), 363  
 get() (*cupyx.scipy.sparse.dia\_matrix* method), 371  
 get() (*cupyx.scipy.sparse.spmatrix* method), 376  
 get\_allocator() (in module *cupy.cuda*), 433  
 get\_array\_module() (in module *cupy*), 64  
 get\_array\_module() (in module *cupyx.scipy*), 65  
 get\_build\_version() (in module *cupy.cuda.nccl*), 464  
 get\_channel\_format() (*cupy.cuda.texture.ChannelFormatDescriptor* method), 453  
 get\_current\_stream() (in module *cupy.cuda*), 451  
 get\_default\_memory\_pool() (in module *cupy*), 422

get\_default\_pinned\_memory\_pool() (in module *cupy*), 423  
 get\_elapsed\_time() (in module *cupy.cuda*), 452  
 get\_fft\_plan() (in module *cupyx.scipy.fftpack*), 289  
 get\_function() (*cupy.RawModule* method), 482  
 get\_global() (*cupy.RawModule* method), 483  
 get\_limit() (*cupy.cuda.MemoryAsyncPool* method), 436  
 get\_limit() (*cupy.cuda.MemoryPool* method), 434  
 get\_plan\_cache() (in module *cupy.fft.config*), 129  
 get\_random\_state() (in module *cupy.random*), 232  
 get\_resource\_desc() (*cupy.cuda.texture.ResourceDescriptor* method), 456  
 get\_shape() (*cupyx.scipy.sparse.coo\_matrix* method), 347  
 get\_shape() (*cupyx.scipy.sparse.csc\_matrix* method), 354  
 get\_shape() (*cupyx.scipy.sparse.csr\_matrix* method), 364  
 get\_shape() (*cupyx.scipy.sparse.dia\_matrix* method), 372  
 get\_shape() (*cupyx.scipy.sparse.spmatrix* method), 376  
 get\_texref() (*cupy.RawModule* method), 483  
 get\_texture\_desc() (*cupy.cuda.texture.TextureDescriptor* method), 457  
 get\_unique\_id() (in module *cupy.cuda.nccl*), 464  
 get\_version() (in module *cupy.cuda.nccl*), 464  
 getcol() (*cupyx.scipy.sparse.csc\_matrix* method), 355  
 getcol() (*cupyx.scipy.sparse.csr\_matrix* method), 364  
 getDevice() (in module *cupy.cuda.runtime*), 468  
 getDeviceCount() (in module *cupy.cuda.runtime*), 469  
 getDeviceProperties() (in module *cupy.cuda.runtime*), 468  
 getformat() (*cupyx.scipy.sparse.coo\_matrix* method), 347  
 getformat() (*cupyx.scipy.sparse.csc\_matrix* method), 355  
 getformat() (*cupyx.scipy.sparse.csr\_matrix* method), 364  
 getformat() (*cupyx.scipy.sparse.dia\_matrix* method), 372  
 getformat() (*cupyx.scipy.sparse.spmatrix* method), 376  
 getH() (*cupyx.scipy.sparse.coo\_matrix* method), 347  
 getH() (*cupyx.scipy.sparse.csc\_matrix* method), 354  
 getH() (*cupyx.scipy.sparse.csr\_matrix* method), 364  
 getH() (*cupyx.scipy.sparse.dia\_matrix* method), 372  
 getH() (*cupyx.scipy.sparse.spmatrix* method), 376  
 getmaxprint() (*cupyx.scipy.sparse.coo\_matrix* method), 347  
 getmaxprint() (*cupyx.scipy.sparse.csc\_matrix* method), 355  
 getmaxprint() (*cupyx.scipy.sparse.csr\_matrix* method), 364



- `getmaxprint()` (*cupyx.scipy.sparse.dia\_matrix method*), 372  
`getmaxprint()` (*cupyx.scipy.sparse.spmatrix method*), 376  
`getnnz()` (*cupyx.scipy.sparse.coo\_matrix method*), 347  
`getnnz()` (*cupyx.scipy.sparse.csc\_matrix method*), 355  
`getnnz()` (*cupyx.scipy.sparse.csr\_matrix method*), 364  
`getnnz()` (*cupyx.scipy.sparse.dia\_matrix method*), 372  
`getnnz()` (*cupyx.scipy.sparse.spmatrix method*), 376  
`getrow()` (*cupyx.scipy.sparse.csc\_matrix method*), 355  
`getrow()` (*cupyx.scipy.sparse.csr\_matrix method*), 364  
`gmres()` (*in module cupyx.scipy.sparse.linalg*), 392  
`gradient()` (*in module cupy*), 173  
`greater` (*in module cupy*), 80  
`greater_equal` (*in module cupy*), 80  
`grey_closing()` (*in module cupyx.scipy.ndimage*), 339  
`grey_dilation()` (*in module cupyx.scipy.ndimage*), 339  
`grey_erosion()` (*in module cupyx.scipy.ndimage*), 340  
`grey_opening()` (*in module cupyx.scipy.ndimage*), 341  
`gridDim` (*in module cupyx.jit*), 486  
`groupEnd()` (*in module cupy.cuda.nccl*), 465  
`groupStart()` (*in module cupy.cuda.nccl*), 464  
`gumbel()` (*cupy.random.RandomState method*), 202  
`gumbel()` (*in module cupy.random*), 215
- ## H
- `H` (*cupyx.scipy.sparse.coo\_matrix attribute*), 351  
`H` (*cupyx.scipy.sparse.csc\_matrix attribute*), 360  
`H` (*cupyx.scipy.sparse.csr\_matrix attribute*), 369  
`H` (*cupyx.scipy.sparse.dia\_matrix attribute*), 375  
`H` (*cupyx.scipy.sparse.linalg.LinearOperator attribute*), 389  
`H` (*cupyx.scipy.sparse.spmatrix attribute*), 379  
`hadamard()` (*in module cupyx.scipy.linalg*), 297  
`hamming()` (*in module cupy*), 270  
`hankel()` (*in module cupyx.scipy.linalg*), 298  
`hanning()` (*in module cupy*), 270  
`has_canonical_format` (*cupyx.scipy.sparse.csc\_matrix attribute*), 360  
`has_canonical_format` (*cupyx.scipy.sparse.csr\_matrix attribute*), 369  
`has_sorted_indices` (*cupyx.scipy.sparse.csc\_matrix attribute*), 360  
`has_sorted_indices` (*cupyx.scipy.sparse.csr\_matrix attribute*), 369  
`height` (*cupy.cuda.texture.CUDAArray attribute*), 455  
`helmert()` (*in module cupyx.scipy.linalg*), 298  
`hfft()` (*in module cupy.fft*), 125  
`hfft()` (*in module cupyx.scipy.fft*), 280  
`hilbert()` (*in module cupyx.scipy.linalg*), 299  
`histogram()` (*in module cupy*), 252  
`histogram()` (*in module cupyx.scipy.ndimage*), 326  
`histogram2d()` (*in module cupy*), 253  
`histogramdd()` (*in module cupy*), 254  
`hostAlloc()` (*in module cupy.cuda.runtime*), 470  
`hostRegister()` (*in module cupy.cuda.runtime*), 470  
`hostUnregister()` (*in module cupy.cuda.runtime*), 470  
`hsplit()` (*in module cupy*), 109  
`hstack()` (*in module cupy*), 107  
`hstack()` (*in module cupyx.scipy.sparse*), 383  
`huber` (*in module cupyx.scipy.special*), 401  
`hypergeometric()` (*cupy.random.RandomState method*), 202  
`hypergeometric()` (*in module cupy.random*), 215  
`hypot` (*in module cupy*), 76
- ## I
- `i0` (*in module cupy*), 175  
`i0` (*in module cupyx.scipy.special*), 399  
`i1` (*in module cupyx.scipy.special*), 399  
`id` (*cupy.cuda.Device attribute*), 421  
`identity` (*cupy.ReductionKernel attribute*), 478  
`identity()` (*in module cupy*), 88  
`identity()` (*in module cupyx.scipy.sparse*), 380  
`ifft()` (*in module cupy.fft*), 119  
`ifft()` (*in module cupyx.scipy.fft*), 273  
`ifft()` (*in module cupyx.scipy.fftpack*), 284  
`ifft2()` (*in module cupy.fft*), 120  
`ifft2()` (*in module cupyx.scipy.fft*), 274  
`ifft2()` (*in module cupyx.scipy.fftpack*), 286  
`ifftn()` (*in module cupy.fft*), 121  
`ifftn()` (*in module cupyx.scipy.fft*), 275  
`ifftn()` (*in module cupyx.scipy.fftpack*), 287  
`ifftshift()` (*in module cupy.fft*), 127  
`ifftshift()` (*in module cupyx.scipy.fft*), 281  
`ihfft()` (*in module cupy.fft*), 125  
`ihfft()` (*in module cupyx.scipy.fft*), 280  
`imag` (*cupy.ndarray attribute*), 61  
`imag()` (*in module cupy*), 177  
`in1d()` (*in module cupy*), 233  
`in_params` (*cupy.ElementwiseKernel attribute*), 476  
`in_params` (*cupy.ReductionKernel attribute*), 478  
`indices()` (*in module cupy*), 134  
`initAll()` (*cupy.cuda.nccl.NcclCommunicator static method*), 463  
`initialize()` (*in module cupy.cuda.profiler*), 460  
`inner()` (*in module cupy*), 149  
`integ()` (*cupy.poly1d method*), 187  
`integers()` (*cupy.random.Generator method*), 193  
`interp()` (*in module cupy*), 178  
`inv()` (*in module cupy.linalg*), 159  
`invert` (*in module cupy*), 79  
`ipCloseMemHandle()` (*in module cupy.cuda.runtime*), 475  
`ipcGetEventHandle()` (*in module cupy.cuda.runtime*), 475  
`ipcGetMemHandle()` (*in module cupy.cuda.runtime*), 474

- ipcOpenEventHandle() (in module *cupy.cuda.runtime*), 475  
 ipcOpenMemHandle() (in module *cupy.cuda.runtime*), 474  
 irfft() (in module *cupy.fft*), 122  
 irfft() (in module *cupyx.scipy.fft*), 277  
 irfft() (in module *cupyx.scipy.fftpack*), 288  
 irfft2() (in module *cupy.fft*), 123  
 irfft2() (in module *cupyx.scipy.fft*), 278  
 irfftn() (in module *cupy.fft*), 124  
 irfftn() (in module *cupyx.scipy.fft*), 279  
 isclose() (in module *cupy*), 166  
 iscomplex() (in module *cupy*), 162  
 iscomplexobj() (in module *cupy*), 162  
 isfinite (in module *cupy*), 83  
 isfortran() (in module *cupy*), 163  
 isin() (in module *cupy*), 234  
 isinf (in module *cupy*), 84  
 isnan (in module *cupy*), 84  
 isreal() (in module *cupy*), 164  
 isrealobj() (in module *cupy*), 164  
 isscalar() (in module *cupy*), 165  
 issparse() (in module *cupyx.scipy.sparse*), 386  
 isspmatrix() (in module *cupyx.scipy.sparse*), 387  
 isspmatrix\_coo() (in module *cupyx.scipy.sparse*), 387  
 isspmatrix\_csc() (in module *cupyx.scipy.sparse*), 387  
 isspmatrix\_csr() (in module *cupyx.scipy.sparse*), 387  
 isspmatrix\_dia() (in module *cupyx.scipy.sparse*), 387  
 item() (*cupy.ndarray* method), 56  
 itemsize (*cupy.ndarray* attribute), 61  
 iterate\_structure() (in module *cupyx.scipy.ndimage*), 341  
 ix\_() (in module *cupy*), 134
- ## J
- j0 (in module *cupyx.scipy.special*), 398  
 j1 (in module *cupyx.scipy.special*), 398
- ## K
- kaiser() (in module *cupy*), 271  
 kernel (*cupy.RawKernel* attribute), 481  
 kl\_div (in module *cupyx.scipy.special*), 400  
 kron() (in module *cupy*), 151  
 kron() (in module *cupyx.scipy.linalg*), 299  
 kron() (in module *cupyx.scipy.sparse*), 380  
 kwargs (*cupy.ElementwiseKernel* attribute), 476
- ## L
- label() (in module *cupyx.scipy.ndimage*), 326  
 labeled\_comprehension() (in module *cupyx.scipy.ndimage*), 327  
 laplace() (*cupy.random.RandomState* method), 202  
 laplace() (in module *cupy.random*), 216  
 laplace() (in module *cupyx.scipy.ndimage*), 310  
 launch\_host\_func() (*cupy.cuda.ExternalStream* method), 449  
 launch\_host\_func() (*cupy.cuda.Stream* method), 447  
 launchHostFunc() (in module *cupy.cuda.runtime*), 474  
 lcm (in module *cupy*), 74  
 ldexp (in module *cupy*), 85  
 left\_shift (in module *cupy*), 79  
 leslie() (in module *cupyx.scipy.linalg*), 299  
 less (in module *cupy*), 80  
 less\_equal (in module *cupy*), 81  
 lexsort() (in module *cupy*), 235  
 LinearOperator (class in *cupyx.scipy.sparse.linalg*), 388  
 LineProfileHook (class in *cupy.cuda.memory\_hooks*), 444  
 linspace() (in module *cupy*), 93  
 load() (in module *cupy*), 145  
 lobpcg() (in module *cupyx.scipy.sparse.linalg*), 395  
 local\_size\_bytes (*cupy.RawKernel* attribute), 481  
 log (in module *cupy*), 72  
 log10 (in module *cupy*), 72  
 log1p (in module *cupy*), 73  
 log1p() (*cupyx.scipy.sparse.coo\_matrix* method), 347  
 log1p() (*cupyx.scipy.sparse.csc\_matrix* method), 355  
 log1p() (*cupyx.scipy.sparse.csr\_matrix* method), 364  
 log1p() (*cupyx.scipy.sparse.dia\_matrix* method), 372  
 log2 (in module *cupy*), 72  
 logaddexp (in module *cupy*), 69  
 logaddexp2 (in module *cupy*), 69  
 logical\_and (in module *cupy*), 81  
 logical\_not (in module *cupy*), 82  
 logical\_or (in module *cupy*), 81  
 logical\_xor (in module *cupy*), 82  
 logistic() (*cupy.random.RandomState* method), 203  
 logistic() (in module *cupy.random*), 216  
 lognormal() (*cupy.random.RandomState* method), 203  
 lognormal() (in module *cupy.random*), 217  
 logseries() (*cupy.random.RandomState* method), 203  
 logseries() (in module *cupy.random*), 217  
 logspace() (in module *cupy*), 94  
 lsqr() (in module *cupyx.scipy.sparse.linalg*), 393  
 lstsq() (in module *cupy.linalg*), 158  
 lu() (in module *cupyx.scipy.linalg*), 292  
 lu\_factor() (in module *cupyx.scipy.linalg*), 292  
 lu\_solve() (in module *cupyx.scipy.linalg*), 293
- ## M
- malloc() (*cupy.cuda.CFunctionAllocator* method), 439  
 malloc() (*cupy.cuda.MemoryAsyncPool* method), 436  
 malloc() (*cupy.cuda.MemoryPool* method), 434  
 malloc() (*cupy.cuda.PinnedMemoryPool* method), 437  
 malloc() (*cupy.cuda.PythonFunctionAllocator* method), 438

malloc() (in module *cupy.cuda.runtime*), 470  
 malloc3DArray() (in module *cupy.cuda.runtime*), 470  
 malloc\_async() (in module *cupy.cuda*), 432  
 malloc\_managed() (in module *cupy.cuda*), 431  
 malloc\_postprocess()  
     (*cupy.cuda.memory\_hooks.DebugPrintHook*  
     *method*), 444  
 malloc\_postprocess()  
     (*cupy.cuda.memory\_hooks.LineProfileHook*  
     *method*), 446  
 malloc\_postprocess() (in module *cupy.cuda.MemoryHook*  
     *method*), 441  
 malloc\_preprocess()  
     (*cupy.cuda.memory\_hooks.DebugPrintHook*  
     *method*), 444  
 malloc\_preprocess()  
     (*cupy.cuda.memory\_hooks.LineProfileHook*  
     *method*), 446  
 malloc\_preprocess() (in module *cupy.cuda.MemoryHook*  
     *method*), 441  
 mallocArray() (in module *cupy.cuda.runtime*), 470  
 mallocAsync() (in module *cupy.cuda.runtime*), 470  
 mallocManaged() (in module *cupy.cuda.runtime*), 470  
 ManagedMemory (class in *cupy.cuda*), 425  
 map\_coordinates() (in module *cupyx.scipy.ndimage*),  
     320  
 map\_expr (*cupy.ReductionKernel* attribute), 478  
 Mark() (in module *cupy.cuda.nvtx*), 461  
 MarkC() (in module *cupy.cuda.nvtx*), 461  
 matmat() (*cupyx.scipy.sparse.linalg.LinearOperator*  
     *method*), 388  
 matmul() (in module *cupy*), 68  
 matrix\_power() (in module *cupy.linalg*), 151  
 matrix\_rank() (in module *cupy.linalg*), 156  
 matvec() (*cupyx.scipy.sparse.linalg.LinearOperator*  
     *method*), 388  
 max() (*cupy.ndarray* method), 56  
 max() (*cupyx.scipy.sparse.csc\_matrix* method), 355  
 max() (*cupyx.scipy.sparse.csr\_matrix* method), 364  
 max\_dynamic\_shared\_size\_bytes (*cupy.RawKernel*  
     attribute), 481  
 max\_threads\_per\_block (*cupy.RawKernel* attribute),  
     481  
 maximum (in module *cupy*), 82  
 maximum() (*cupyx.scipy.sparse.coo\_matrix* method), 347  
 maximum() (*cupyx.scipy.sparse.csc\_matrix* method), 355  
 maximum() (*cupyx.scipy.sparse.csr\_matrix* method), 365  
 maximum() (*cupyx.scipy.sparse.dia\_matrix* method), 372  
 maximum() (*cupyx.scipy.sparse.spmatrix* method), 377  
 maximum() (in module *cupyx.scipy.ndimage*), 327  
 maximum\_filter() (in module *cupyx.scipy.ndimage*),  
     310  
 maximum\_filter1d() (in module *cupyx.scipy.ndimage*),  
     311  
 maximum\_position() (in module *cupyx.scipy.ndimage*),  
     328  
 may\_share\_memory() (in module *cupy*), 179  
 mean() (*cupy.ndarray* method), 56  
 mean() (*cupyx.scipy.sparse.coo\_matrix* method), 347  
 mean() (*cupyx.scipy.sparse.csc\_matrix* method), 355  
 mean() (*cupyx.scipy.sparse.csr\_matrix* method), 365  
 mean() (*cupyx.scipy.sparse.dia\_matrix* method), 372  
 mean() (*cupyx.scipy.sparse.spmatrix* method), 377  
 mean() (in module *cupy*), 247  
 mean() (in module *cupyx.scipy.ndimage*), 328  
 medfilt() (in module *cupyx.scipy.signal*), 411  
 medfilt2d() (in module *cupyx.scipy.signal*), 411  
 median() (in module *cupy*), 246  
 median() (in module *cupyx.scipy.ndimage*), 329  
 median\_filter() (in module *cupyx.scipy.ndimage*), 312  
 mem (*cupy.cuda.MemoryPointer* attribute), 430  
 mem (*cupy.cuda.PinnedMemoryPointer* attribute), 431  
 mem\_info (*cupy.cuda.Device* attribute), 421  
 memAdvise() (in module *cupy.cuda.runtime*), 473  
 memcpy() (in module *cupy.cuda.runtime*), 471  
 memcpy2D() (in module *cupy.cuda.runtime*), 471  
 memcpy2DAsync() (in module *cupy.cuda.runtime*), 472  
 memcpy2DFromArray() (in module *cupy.cuda.runtime*),  
     472  
 memcpy2DFromArrayAsync() (in module  
     *cupy.cuda.runtime*), 472  
 memcpy2DToArray() (in module *cupy.cuda.runtime*),  
     472  
 memcpy2DToArrayAsync() (in module  
     *cupy.cuda.runtime*), 472  
 memcpy3D() (in module *cupy.cuda.runtime*), 472  
 memcpy3DAsync() (in module *cupy.cuda.runtime*), 472  
 memcpyAsync() (in module *cupy.cuda.runtime*), 471  
 memcpyPeer() (in module *cupy.cuda.runtime*), 471  
 memcpyPeerAsync() (in module *cupy.cuda.runtime*),  
     471  
 memGetInfo() (in module *cupy.cuda.runtime*), 471  
 memoize() (in module *cupy*), 488  
 Memory (class in *cupy.cuda*), 423  
 MemoryAsync (class in *cupy.cuda*), 424  
 MemoryAsyncPool (class in *cupy.cuda*), 436  
 MemoryHook (class in *cupy.cuda*), 440  
 MemoryPointer (class in *cupy.cuda*), 427  
 MemoryPool (class in *cupy.cuda*), 433  
 memPoolTrimTo() (in module *cupy.cuda.runtime*), 469  
 memPrefetchAsync() (in module *cupy.cuda.runtime*),  
     473  
 memset() (*cupy.cuda.MemoryPointer* method), 429  
 memset() (in module *cupy.cuda.runtime*), 472  
 memset\_async() (*cupy.cuda.MemoryPointer* method),  
     429  
 memsetAsync() (in module *cupy.cuda.runtime*), 472  
 meshgrid() (in module *cupy*), 94

- `mgrid` (in module `cupy`), 95
  - `min()` (`cupy.ndarray` method), 56
  - `min()` (`cupyx.scipy.sparse.csc_matrix` method), 356
  - `min()` (`cupyx.scipy.sparse.csr_matrix` method), 365
  - `minimum` (in module `cupy`), 82
  - `minimum()` (`cupyx.scipy.sparse.coo_matrix` method), 348
  - `minimum()` (`cupyx.scipy.sparse.csc_matrix` method), 356
  - `minimum()` (`cupyx.scipy.sparse.csr_matrix` method), 365
  - `minimum()` (`cupyx.scipy.sparse.dia_matrix` method), 372
  - `minimum()` (`cupyx.scipy.sparse.spmatrix` method), 377
  - `minimum()` (in module `cupyx.scipy.ndimage`), 329
  - `minimum_filter()` (in module `cupyx.scipy.ndimage`), 312
  - `minimum_filter1d()` (in module `cupyx.scipy.ndimage`), 313
  - `minimum_position()` (in module `cupyx.scipy.ndimage`), 330
  - `mod` (in module `cupy`), 70
  - `modf` (in module `cupy`), 85
  - module
    - `cupy`, 1
    - `cupy.fft`, 118
    - `cupy.linalg`, 148
    - `cupy.polynomial.polynomial`, 184
    - `cupy.polynomial.polyutils`, 185
    - `cupy.random`, 191
    - `cupy.testing`, 255
    - `cupyx.optimizing`, 418
    - `cupyx.scipy`, 272
    - `cupyx.scipy.fft`, 272
    - `cupyx.scipy.fftpack`, 283
    - `cupyx.scipy.linalg`, 290
    - `cupyx.scipy.ndimage`, 301
    - `cupyx.scipy.signal`, 404
    - `cupyx.scipy.sparse`, 344
    - `cupyx.scipy.sparse.linalg`, 387
    - `cupyx.scipy.special`, 398
    - `cupyx.scipy.stats`, 412
  - module (`cupy.RawModule` attribute), 484
  - `morphological_gradient()` (in module `cupyx.scipy.ndimage`), 342
  - `morphological_laplace()` (in module `cupyx.scipy.ndimage`), 342
  - `moveaxis()` (in module `cupy`), 100
  - `MRG32k3a` (class in `cupy.random`), 198
  - `msort()` (in module `cupy`), 236
  - `multinomial()` (in module `cupy.random`), 218
  - `multiply` (in module `cupy`), 68
  - `multiply()` (`cupyx.scipy.sparse.coo_matrix` method), 348
  - `multiply()` (`cupyx.scipy.sparse.csc_matrix` method), 356
  - `multiply()` (`cupyx.scipy.sparse.csr_matrix` method), 365
  - `multiply()` (`cupyx.scipy.sparse.dia_matrix` method), 372
  - `multiply()` (`cupyx.scipy.sparse.spmatrix` method), 377
  - `multivariate_normal()` (`cupy.random.RandomState` method), 203
  - `multivariate_normal()` (in module `cupy.random`), 218
- ## N
- `n_free_blocks()` (`cupy.cuda.MemoryAsyncPool` method), 436
  - `n_free_blocks()` (`cupy.cuda.MemoryPool` method), 434
  - `n_free_blocks()` (`cupy.cuda.PinnedMemoryPool` method), 437
  - `name` (`cupy.cuda.memory_hooks.DebugPrintHook` attribute), 444
  - `name` (`cupy.cuda.memory_hooks.LineProfileHook` attribute), 447
  - `name` (`cupy.cuda.MemoryHook` attribute), 442
  - `name` (`cupy.ElementwiseKernel` attribute), 476
  - `name` (`cupy.RawKernel` attribute), 481
  - `name` (`cupy.ReductionKernel` attribute), 478
  - `name` (`cupy.ufunc` attribute), 66
  - `name_expressions` (`cupy.RawModule` attribute), 484
  - `nan_to_num` (in module `cupy`), 178
  - `nanargmax()` (in module `cupy`), 238
  - `nanargmin()` (in module `cupy`), 239
  - `nancumprod()` (in module `cupy`), 172
  - `nancumsum()` (in module `cupy`), 172
  - `nanmax()` (in module `cupy`), 244
  - `nanmean()` (in module `cupy`), 249
  - `nanmedian()` (in module `cupy`), 248
  - `nanmin()` (in module `cupy`), 243
  - `nanprod()` (in module `cupy`), 170
  - `nanstd()` (in module `cupy`), 249
  - `nansum()` (in module `cupy`), 170
  - `nanvar()` (in module `cupy`), 250
  - `nargs` (`cupy.ElementwiseKernel` attribute), 476
  - `nargs` (`cupy.ReductionKernel` attribute), 478
  - `nargs` (`cupy.ufunc` attribute), 66
  - `nbytes` (`cupy.ndarray` attribute), 62
  - `NcclCommunicator` (class in `cupy.cuda.nccl`), 462
  - `nd` (`cupy.broadcast` attribute), 104
  - `ndarray` (class in `cupy`), 51
  - `ndim` (`cupy.cuda.texture.CUDAarray` attribute), 455
  - `ndim` (`cupy.ndarray` attribute), 62
  - `ndim` (`cupyx.scipy.sparse.coo_matrix` attribute), 351
  - `ndim` (`cupyx.scipy.sparse.csc_matrix` attribute), 360
  - `ndim` (`cupyx.scipy.sparse.csr_matrix` attribute), 369
  - `ndim` (`cupyx.scipy.sparse.dia_matrix` attribute), 375
  - `ndim` (`cupyx.scipy.sparse.linalg.LinearOperator` attribute), 389
  - `ndim` (`cupyx.scipy.sparse.spmatrix` attribute), 379



ndtr (in module *cupyx.scipy.special*), 400  
 negative (in module *cupy*), 69  
 negative\_binomial() (*cupy.random.RandomState* method), 203  
 negative\_binomial() (in module *cupy.random*), 219  
 next\_fast\_len() (in module *cupyx.scipy.fft*), 282  
 nextafter (in module *cupy*), 84  
 nin (*cupy.ElementwiseKernel* attribute), 476  
 nin (*cupy.ReductionKernel* attribute), 478  
 nin (*cupy.ufunc* attribute), 66  
 nnz (*cupyx.scipy.sparse.coo\_matrix* attribute), 351  
 nnz (*cupyx.scipy.sparse.csc\_matrix* attribute), 360  
 nnz (*cupyx.scipy.sparse.csr\_matrix* attribute), 369  
 nnz (*cupyx.scipy.sparse.dia\_matrix* attribute), 375  
 nnz (*cupyx.scipy.sparse.spmatrix* attribute), 379  
 no\_return (*cupy.ElementwiseKernel* attribute), 476  
 noncentral\_chisquare() (*cupy.random.RandomState* method), 204  
 noncentral\_chisquare() (in module *cupy.random*), 219  
 noncentral\_f() (*cupy.random.RandomState* method), 204  
 noncentral\_f() (in module *cupy.random*), 220  
 nonzero() (*cupy.ndarray* method), 57  
 nonzero() (in module *cupy*), 133  
 norm() (in module *cupy.linalg*), 155  
 norm() (in module *cupyx.scipy.sparse.linalg*), 390  
 normal() (*cupy.random.RandomState* method), 204  
 normal() (in module *cupy.random*), 220  
 not\_equal (in module *cupy*), 81  
 nout (*cupy.ElementwiseKernel* attribute), 476  
 nout (*cupy.ReductionKernel* attribute), 478  
 nout (*cupy.ufunc* attribute), 66  
 null (*cupy.cuda.ExternalStream* attribute), 450  
 null (*cupy.cuda.Stream* attribute), 449  
 num\_regs (*cupy.RawKernel* attribute), 481  
 numpy\_cupy\_allclose() (in module *cupy.testing*), 259  
 numpy\_cupy\_array\_almost\_equal() (in module *cupy.testing*), 260  
 numpy\_cupy\_array\_almost\_equal\_nulp() (in module *cupy.testing*), 261  
 numpy\_cupy\_array\_equal() (in module *cupy.testing*), 262  
 numpy\_cupy\_array\_less() (in module *cupy.testing*), 263  
 numpy\_cupy\_array\_list\_equal() (in module *cupy.testing*), 263  
 numpy\_cupy\_array\_max\_ulp() (in module *cupy.testing*), 261  
 NVCC, 493

## O

o (*cupy.poly1d* attribute), 188  
 oaconvolve() (in module *cupyx.scipy.signal*), 407

ogrid (in module *cupy*), 95  
 ones() (in module *cupy*), 88  
 ones\_like() (in module *cupy*), 89  
 operation (*cupy.ElementwiseKernel* attribute), 477  
 optimize() (in module *cupyx.optimizing*), 419  
 options (*cupy.RawKernel* attribute), 481  
 options (*cupy.RawModule* attribute), 484  
 options (*cupy.ReductionKernel* attribute), 478  
 order (*cupy.poly1d* attribute), 188  
 order\_filter() (in module *cupyx.scipy.signal*), 410  
 out\_params (*cupy.ElementwiseKernel* attribute), 477  
 out\_params (*cupy.ReductionKernel* attribute), 478  
 outer() (in module *cupy*), 150

## P

packbits() (in module *cupy*), 116  
 pad() (in module *cupy*), 181  
 params (*cupy.ElementwiseKernel* attribute), 477  
 params (*cupy.ReductionKernel* attribute), 479  
 pareto() (*cupy.random.RandomState* method), 204  
 pareto() (in module *cupy.random*), 221  
 partition() (*cupy.ndarray* method), 57  
 partition() (in module *cupy*), 236  
 pci\_bus\_id (*cupy.cuda.Device* attribute), 421  
 percentile() (in module *cupy*), 245  
 percentile\_filter() (in module *cupyx.scipy.ndimage*), 313  
 permutation() (*cupy.random.RandomState* method), 204  
 permutation() (in module *cupy.random*), 221  
 Philox4x3210 (class in *cupy.random*), 199  
 piecewise() (in module *cupy*), 132  
 PinnedMemory (class in *cupy.cuda*), 427  
 PinnedMemoryPointer (class in *cupy.cuda*), 430  
 PinnedMemoryPool (class in *cupy.cuda*), 437  
 pinv() (in module *cupy.linalg*), 159  
 place() (in module *cupy*), 142  
 pointerGetAttributes() (in module *cupy.cuda.runtime*), 473  
 poisson() (*cupy.random.Generator* method), 193  
 poisson() (*cupy.random.RandomState* method), 204  
 poisson() (in module *cupy.random*), 222  
 poly1d (class in *cupy*), 186  
 polyadd() (in module *cupy*), 190  
 polycompanion() (in module *cupy.polynomial.polynomial*), 184  
 polyfit() (in module *cupy*), 189  
 polygamma() (in module *cupyx.scipy.special*), 402  
 polymul() (in module *cupy*), 190  
 polysub() (in module *cupy*), 190  
 polyval() (in module *cupy*), 188  
 polyvalander() (in module *cupy.polynomial.polynomial*), 184  
 post\_map\_expr (*cupy.ReductionKernel* attribute), 479

- power (in module *cupy*), 70  
 power() (*cupy.random.RandomState* method), 205  
 power() (*cupyx.scipy.sparse.coo\_matrix* method), 348  
 power() (*cupyx.scipy.sparse.csc\_matrix* method), 356  
 power() (*cupyx.scipy.sparse.csr\_matrix* method), 365  
 power() (*cupyx.scipy.sparse.dia\_matrix* method), 372  
 power() (*cupyx.scipy.sparse.spmatrix* method), 377  
 power() (in module *cupy.random*), 222  
 preamble (*cupy.ElementwiseKernel* attribute), 477  
 preamble (*cupy.ReductionKernel* attribute), 479  
 preferred\_shared\_memory\_carveout  
     (*cupy.RawKernel* attribute), 481  
 prefetch() (*cupy.cuda.ManagedMemory* method), 425  
 prewitt() (in module *cupyx.scipy.ndimage*), 314  
 print\_report() (*cupy.cuda.memory\_hooks.LineProfileHook*  
     method), 446  
 prod() (*cupy.ndarray* method), 57  
 prod() (in module *cupy*), 169  
 profile() (in module *cupy.cuda*), 460  
 pseudo\_huber (in module *cupyx.scipy.special*), 401  
 ptds (*cupy.cuda.Stream* attribute), 449  
 ptp() (*cupy.ndarray* method), 57  
 ptp() (in module *cupy*), 244  
 ptr (*cupy.cuda.ManagedMemory* attribute), 425  
 ptr (*cupy.cuda.Memory* attribute), 423  
 ptr (*cupy.cuda.MemoryAsync* attribute), 424  
 ptr (*cupy.cuda.MemoryPointer* attribute), 430  
 ptr (*cupy.cuda.PinnedMemoryPointer* attribute), 431  
 ptr (*cupy.cuda.texture.ChannelFormatDescriptor* at-  
     tribute), 453  
 ptr (*cupy.cuda.texture.CUDAarray* attribute), 455  
 ptr (*cupy.cuda.texture.ResourceDescriptor* attribute),  
     456  
 ptr (*cupy.cuda.texture.SurfaceObject* attribute), 459  
 ptr (*cupy.cuda.texture.TextureDescriptor* attribute), 457  
 ptr (*cupy.cuda.texture.TextureObject* attribute), 458  
 ptr (*cupy.cuda.UnownedMemory* attribute), 426  
 ptx\_version (*cupy.RawKernel* attribute), 481  
 put() (*cupy.ndarray* method), 57  
 put() (in module *cupy*), 142  
 putmask() (in module *cupy*), 143  
 PythonFunctionAllocator (class in *cupy.cuda*), 438
- ## Q
- qr() (in module *cupy.linalg*), 152  
 quantile() (in module *cupy*), 245
- ## R
- r (*cupy.poly1d* attribute), 188  
 r\_ (in module *cupy*), 133  
 rad2deg (in module *cupy*), 78  
 rad2deg() (*cupyx.scipy.sparse.coo\_matrix* method), 348  
 rad2deg() (*cupyx.scipy.sparse.csc\_matrix* method), 356  
 rad2deg() (*cupyx.scipy.sparse.csr\_matrix* method), 365  
 rad2deg() (*cupyx.scipy.sparse.dia\_matrix* method), 372  
 radians (in module *cupy*), 77  
 rand() (*cupy.random.RandomState* method), 205  
 rand() (in module *cupy.random*), 223  
 rand() (in module *cupyx.scipy.sparse*), 385  
 randint() (*cupy.random.RandomState* method), 205  
 randint() (in module *cupy.random*), 223  
 randn() (*cupy.random.RandomState* method), 205  
 randn() (in module *cupy.random*), 224  
 random() (*cupy.random.Generator* method), 194  
 random() (in module *cupy.random*), 224  
 random() (in module *cupyx.scipy.sparse*), 385  
 random\_integers() (in module *cupy.random*), 225  
 random\_raw() (*cupy.random.BitGenerator* method), 196  
 random\_raw() (*cupy.random.MRG32k3a* method), 198  
 random\_raw() (*cupy.random.Philox4x3210* method),  
     199  
 random\_raw() (*cupy.random.XORWOW* method), 197  
 random\_sample() (*cupy.random.RandomState* method),  
     205  
 random\_sample() (in module *cupy.random*), 225  
 RandomState (class in *cupy.random*), 200  
 ranf() (in module *cupy.random*), 225  
 RangePop() (in module *cupy.cuda.nvtx*), 462  
 RangePush() (in module *cupy.cuda.nvtx*), 461  
 RangePushC() (in module *cupy.cuda.nvtx*), 462  
 rank\_filter() (in module *cupyx.scipy.ndimage*), 315  
 rank\_id() (*cupy.cuda.nccl.NcclCommunicator*  
     method), 463  
 ravel() (*cupy.ndarray* method), 57  
 ravel() (in module *cupy*), 99  
 ravel\_multi\_index() (in module *cupy*), 135  
 RawKernel (class in *cupy*), 479  
 rawkernel() (in module *cupyx.jit*), 485  
 RawModule (class in *cupy*), 481  
 rayleigh() (*cupy.random.RandomState* method), 205  
 rayleigh() (in module *cupy.random*), 226  
 real (*cupy.ndarray* attribute), 62  
 real() (in module *cupy*), 177  
 reciprocal (in module *cupy*), 73  
 record() (*cupy.cuda.Event* method), 451  
 record() (*cupy.cuda.ExternalStream* method), 449  
 record() (*cupy.cuda.Stream* method), 448  
 recv() (*cupy.cuda.nccl.NcclCommunicator* method),  
     463  
 reduce() (*cupy.cuda.nccl.NcclCommunicator* method),  
     463  
 reduce\_dims (*cupy.ElementwiseKernel* attribute), 477  
 reduce\_dims (*cupy.ReductionKernel* attribute), 479  
 reduce\_expr (*cupy.ReductionKernel* attribute), 479  
 reduce\_type (*cupy.ReductionKernel* attribute), 479  
 reduced\_view() (*cupy.ndarray* method), 57  
 reduceScatter() (*cupy.cuda.nccl.NcclCommunicator*  
     method), 463

ReductionKernel (class in cupy), 477  
 rel ENTR (in module cupyx.scipy.special), 400  
 remainder (in module cupy), 70  
 repeat() (cupy.ndarray method), 58  
 repeat() (in module cupy), 110  
 repeat() (in module cupyx.time), 490  
 require() (in module cupy), 106  
 ResDesc (cupy.cuda.texture.SurfaceObject attribute), 459  
 ResDesc (cupy.cuda.texture.TextureObject attribute), 458  
 ResDesc (cupy.cuda.texture.TextureReference attribute), 460  
 reshape() (cupy.ndarray method), 58  
 reshape() (cupyx.scipy.sparse.coo\_matrix method), 348  
 reshape() (cupyx.scipy.sparse.csc\_matrix method), 356  
 reshape() (cupyx.scipy.sparse.csr\_matrix method), 366  
 reshape() (cupyx.scipy.sparse.dia\_matrix method), 372  
 reshape() (cupyx.scipy.sparse.spmatrix method), 377  
 reshape() (in module cupy), 99  
 resize() (in module cupy), 111  
 ResourceDescriptor (class in cupy.cuda.texture), 455  
 result\_type() (in module cupy), 117  
 return\_tuple (cupy.ElementwiseKernel attribute), 477  
 rfft() (in module cupy.fft), 122  
 rfft() (in module cupyx.scipy.fft), 276  
 rfft() (in module cupyx.scipy.fftpack), 288  
 rfft2() (in module cupy.fft), 123  
 rfft2() (in module cupyx.scipy.fft), 277  
 rfftfreq() (in module cupy.fft), 126  
 rfftfreq() (in module cupyx.scipy.fft), 282  
 rfftn() (in module cupy.fft), 124  
 rfftn() (in module cupyx.scipy.fft), 278  
 right\_shift (in module cupy), 79  
 rint (in module cupy), 71  
 rint() (cupyx.scipy.sparse.coo\_matrix method), 348  
 rint() (cupyx.scipy.sparse.csc\_matrix method), 356  
 rint() (cupyx.scipy.sparse.csr\_matrix method), 366  
 rint() (cupyx.scipy.sparse.dia\_matrix method), 373  
 rmatmat() (cupyx.scipy.sparse.linalg.LinearOperator method), 389  
 rmatvec() (cupyx.scipy.sparse.linalg.LinearOperator method), 389  
 roll() (in module cupy), 114  
 rollaxis() (in module cupy), 100  
 roots (cupy.poly1d attribute), 188  
 roots() (in module cupy), 188  
 rot90() (in module cupy), 115  
 rotate() (in module cupyx.scipy.ndimage), 321  
 round() (cupy.ndarray method), 58  
 round\_() (in module cupy), 168  
 rsqrt (in module cupyx), 413  
 runtimeGetVersion() (in module cupy.cuda.runtime), 468

## S

sample() (in module cupy.random), 226  
 save() (in module cupy), 146  
 savez() (in module cupy), 146  
 savez\_compressed() (in module cupy), 147  
 scatter\_add() (cupy.ndarray method), 58  
 scatter\_add() (in module cupyx), 413  
 scatter\_max() (cupy.ndarray method), 58  
 scatter\_max() (in module cupyx), 414  
 scatter\_min() (cupy.ndarray method), 58  
 scatter\_min() (in module cupyx), 415  
 searchsorted() (in module cupy), 240  
 seed() (cupy.random.RandomState method), 206  
 seed() (in module cupy.random), 226  
 select() (in module cupy), 140  
 send() (cupy.cuda.nccl.NcclCommunicator method), 463  
 set() (cupy.ndarray method), 58  
 set() (cupy.poly1d method), 187  
 set\_allocator() (in module cupy.cuda), 433  
 set\_cufft\_callbacks (class in cupy.fft.config), 127  
 set\_cufft\_gpus() (in module cupy.fft.config), 129  
 set\_limit() (cupy.cuda.MemoryAsyncPool method), 436  
 set\_limit() (cupy.cuda.MemoryPool method), 435  
 set\_pinned\_memory\_allocator() (in module cupy.cuda), 433  
 set\_random\_state() (in module cupy.random), 233  
 set\_shape() (cupyx.scipy.sparse.coo\_matrix method), 348  
 set\_shape() (cupyx.scipy.sparse.csc\_matrix method), 356  
 set\_shape() (cupyx.scipy.sparse.csr\_matrix method), 366  
 set\_shape() (cupyx.scipy.sparse.dia\_matrix method), 373  
 set\_shape() (cupyx.scipy.sparse.spmatrix method), 377  
 setDevice() (in module cupy.cuda.runtime), 469  
 setdiag() (cupyx.scipy.sparse.coo\_matrix method), 348  
 setdiag() (cupyx.scipy.sparse.csc\_matrix method), 356  
 setdiag() (cupyx.scipy.sparse.csr\_matrix method), 366  
 setdiag() (cupyx.scipy.sparse.dia\_matrix method), 373  
 setdiag() (cupyx.scipy.sparse.spmatrix method), 377  
 shape (cupy.broadcast attribute), 104  
 shape (cupy.ndarray attribute), 62  
 shape (cupyx.scipy.sparse.coo\_matrix attribute), 351  
 shape (cupyx.scipy.sparse.csc\_matrix attribute), 360  
 shape (cupyx.scipy.sparse.csr\_matrix attribute), 369  
 shape (cupyx.scipy.sparse.dia\_matrix attribute), 375  
 shape (cupyx.scipy.sparse.spmatrix attribute), 379  
 shape() (in module cupy), 98  
 shared\_memory (in module cupyx.jit), 486  
 shared\_size\_bytes (cupy.RawKernel attribute), 481  
 shares\_memory() (in module cupy), 179



- `shift()` (in module `cupyx.scipy.ndimage`), 322
- `show_config()` (in module `cupy`), 179
- `show_plan_cache_info()` (in module `cupy.fft.config`), 130
- `shuffle()` (`cupy.random.RandomState` method), 206
- `shuffle()` (in module `cupy.random`), 227
- `sign` (in module `cupy`), 71
- `sign()` (`cupyx.scipy.sparse.coo_matrix` method), 348
- `sign()` (`cupyx.scipy.sparse.csc_matrix` method), 357
- `sign()` (`cupyx.scipy.sparse.csr_matrix` method), 366
- `sign()` (`cupyx.scipy.sparse.dia_matrix` method), 373
- `signbit` (in module `cupy`), 84
- `sin` (in module `cupy`), 74
- `sin()` (`cupyx.scipy.sparse.coo_matrix` method), 348
- `sin()` (`cupyx.scipy.sparse.csc_matrix` method), 357
- `sin()` (`cupyx.scipy.sparse.csr_matrix` method), 366
- `sin()` (`cupyx.scipy.sparse.dia_matrix` method), 373
- `sinc` (in module `cupy`), 175
- `sinh` (in module `cupy`), 76
- `sinh()` (`cupyx.scipy.sparse.coo_matrix` method), 348
- `sinh()` (`cupyx.scipy.sparse.csc_matrix` method), 357
- `sinh()` (`cupyx.scipy.sparse.csr_matrix` method), 366
- `sinh()` (`cupyx.scipy.sparse.dia_matrix` method), 373
- `size` (`cupy.broadcast` attribute), 104
- `size` (`cupy.cuda.ManagedMemory` attribute), 425
- `size` (`cupy.cuda.Memory` attribute), 423
- `size` (`cupy.cuda.MemoryAsync` attribute), 424
- `size` (`cupy.cuda.UnownedMemory` attribute), 426
- `size` (`cupy.ndarray` attribute), 62
- `size` (`cupyx.scipy.sparse.coo_matrix` attribute), 351
- `size` (`cupyx.scipy.sparse.csc_matrix` attribute), 360
- `size` (`cupyx.scipy.sparse.csr_matrix` attribute), 369
- `size` (`cupyx.scipy.sparse.dia_matrix` attribute), 375
- `size` (`cupyx.scipy.sparse.spmatrix` attribute), 379
- `size()` (`cupy.cuda.nccl.NcclCommunicator` method), 463
- `size()` (`cupy.cuda.PinnedMemoryPointer` method), 431
- `slogdet()` (in module `cupy.linalg`), 156
- `sobel()` (in module `cupyx.scipy.ndimage`), 315
- `solve()` (in module `cupy.linalg`), 157
- `solve_triangular()` (in module `cupyx.scipy.linalg`), 290
- `sort()` (`cupy.ndarray` method), 58
- `sort()` (in module `cupy`), 234
- `sort_complex()` (in module `cupy`), 236
- `sort_indices()` (`cupyx.scipy.sparse.csc_matrix` method), 357
- `sort_indices()` (`cupyx.scipy.sparse.csr_matrix` method), 366
- `sorted_indices()` (`cupyx.scipy.sparse.csc_matrix` method), 357
- `sorted_indices()` (`cupyx.scipy.sparse.csr_matrix` method), 366
- `spdiags()` (in module `cupyx.scipy.sparse`), 382
- `spilu()` (in module `cupyx.scipy.sparse.linalg`), 397
- `spline_filter()` (in module `cupyx.scipy.ndimage`), 322
- `spline_filter1d()` (in module `cupyx.scipy.ndimage`), 323
- `split()` (in module `cupy`), 109
- `splu()` (in module `cupyx.scipy.sparse.linalg`), 397
- `spmatrix` (class in `cupyx.scipy.sparse`), 375
- `spsolve()` (in module `cupyx.scipy.sparse.linalg`), 391
- `spsolve_triangular()` (in module `cupyx.scipy.sparse.linalg`), 391
- `sqrt` (in module `cupy`), 73
- `sqrt()` (`cupyx.scipy.sparse.coo_matrix` method), 348
- `sqrt()` (`cupyx.scipy.sparse.csc_matrix` method), 357
- `sqrt()` (`cupyx.scipy.sparse.csr_matrix` method), 366
- `sqrt()` (`cupyx.scipy.sparse.dia_matrix` method), 373
- `square` (in module `cupy`), 73
- `squeeze()` (`cupy.ndarray` method), 59
- `squeeze()` (in module `cupy`), 105
- `stack()` (in module `cupy`), 107
- `standard_cauchy()` (`cupy.random.RandomState` method), 206
- `standard_cauchy()` (in module `cupy.random`), 227
- `standard_deviation()` (in module `cupyx.scipy.ndimage`), 331
- `standard_exponential()` (`cupy.random.Generator` method), 194
- `standard_exponential()` (`cupy.random.RandomState` method), 206
- `standard_exponential()` (in module `cupy.random`), 227
- `standard_gamma()` (`cupy.random.Generator` method), 195
- `standard_gamma()` (`cupy.random.RandomState` method), 206
- `standard_gamma()` (in module `cupy.random`), 228
- `standard_normal()` (`cupy.random.Generator` method), 195
- `standard_normal()` (`cupy.random.RandomState` method), 206
- `standard_normal()` (in module `cupy.random`), 228
- `standard_t()` (`cupy.random.RandomState` method), 207
- `standard_t()` (in module `cupy.random`), 229
- `start()` (in module `cupy.cuda.profiler`), 461
- `state()` (`cupy.random.MRG32k3a` method), 198
- `state()` (`cupy.random.Philox4x3210` method), 200
- `state()` (`cupy.random.XORWOW` method), 197
- `std()` (`cupy.ndarray` method), 59
- `std()` (in module `cupy`), 248
- `stop()` (in module `cupy.cuda.profiler`), 461
- `Stream` (class in `cupy.cuda`), 447
- `stream_ref` (`cupy.cuda.MemoryAsync` attribute), 424
- `streamAddCallback()` (in module `cupy.cuda.runtime`), 473
- `streamCreate()` (in module `cupy.cuda.runtime`), 473

[streamCreateWithFlags\(\)](#) (in [module cupy.cuda.runtime](#)), 473  
[streamDestroy\(\)](#) (in [module cupy.cuda.runtime](#)), 473  
[streamQuery\(\)](#) (in [module cupy.cuda.runtime](#)), 473  
[streamSynchronize\(\)](#) (in [module cupy.cuda.runtime](#)), 473  
[streamWaitEvent\(\)](#) (in [module cupy.cuda.runtime](#)), 473  
[strides](#) ([cupy.ndarray](#) attribute), 62  
[subtract](#) (in [module cupy](#)), 67  
[sum\(\)](#) ([cupy.ndarray](#) method), 59  
[sum\(\)](#) ([cupyx.scipy.sparse.coo\\_matrix](#) method), 348  
[sum\(\)](#) ([cupyx.scipy.sparse.csc\\_matrix](#) method), 357  
[sum\(\)](#) ([cupyx.scipy.sparse.csr\\_matrix](#) method), 366  
[sum\(\)](#) ([cupyx.scipy.sparse.dia\\_matrix](#) method), 373  
[sum\(\)](#) ([cupyx.scipy.sparse.spmatrix](#) method), 378  
[sum\(\)](#) (in [module cupy](#)), 170  
[sum\\_duplicates\(\)](#) ([cupyx.scipy.sparse.coo\\_matrix](#) method), 349  
[sum\\_duplicates\(\)](#) ([cupyx.scipy.sparse.csc\\_matrix](#) method), 357  
[sum\\_duplicates\(\)](#) ([cupyx.scipy.sparse.csr\\_matrix](#) method), 366  
[sum\\_labels\(\)](#) (in [module cupyx.scipy.ndimage](#)), 331  
[SurfaceObject](#) (class in [cupy.cuda.texture](#)), 458  
[svd\(\)](#) (in [module cupy.linalg](#)), 153  
[svds\(\)](#) (in [module cupyx.scipy.sparse.linalg](#)), 396  
[swapaxes\(\)](#) ([cupy.ndarray](#) method), 59  
[swapaxes\(\)](#) (in [module cupy](#)), 101  
[synchronize\(\)](#) ([cupy.cuda.Device](#) method), 420  
[synchronize\(\)](#) ([cupy.cuda.Event](#) method), 451  
[synchronize\(\)](#) ([cupy.cuda.ExternalStream](#) method), 450  
[synchronize\(\)](#) ([cupy.cuda.Stream](#) method), 448  
[syncthreads](#) (in [module cupyx.jit](#)), 486

## T

[T](#) ([cupy.ndarray](#) attribute), 61  
[T](#) ([cupyx.scipy.sparse.coo\\_matrix](#) attribute), 351  
[T](#) ([cupyx.scipy.sparse.csc\\_matrix](#) attribute), 360  
[T](#) ([cupyx.scipy.sparse.csr\\_matrix](#) attribute), 369  
[T](#) ([cupyx.scipy.sparse.dia\\_matrix](#) attribute), 375  
[T](#) ([cupyx.scipy.sparse.linalg.LinearOperator](#) attribute), 389  
[T](#) ([cupyx.scipy.sparse.spmatrix](#) attribute), 379  
[take\(\)](#) ([cupy.ndarray](#) method), 59  
[take\(\)](#) (in [module cupy](#)), 138  
[take\\_along\\_axis\(\)](#) (in [module cupy](#)), 139  
[tan](#) (in [module cupy](#)), 75  
[tan\(\)](#) ([cupyx.scipy.sparse.coo\\_matrix](#) method), 349  
[tan\(\)](#) ([cupyx.scipy.sparse.csc\\_matrix](#) method), 358  
[tan\(\)](#) ([cupyx.scipy.sparse.csr\\_matrix](#) method), 367  
[tan\(\)](#) ([cupyx.scipy.sparse.dia\\_matrix](#) method), 373  
[tanh](#) (in [module cupy](#)), 76  
[tanh\(\)](#) ([cupyx.scipy.sparse.coo\\_matrix](#) method), 349  
[tanh\(\)](#) ([cupyx.scipy.sparse.csc\\_matrix](#) method), 358  
[tanh\(\)](#) ([cupyx.scipy.sparse.csr\\_matrix](#) method), 367  
[tanh\(\)](#) ([cupyx.scipy.sparse.dia\\_matrix](#) method), 373  
[tensordot\(\)](#) (in [module cupy](#)), 150  
[tensorinv\(\)](#) (in [module cupy.linalg](#)), 160  
[tensorsolve\(\)](#) (in [module cupy.linalg](#)), 158  
[TexDesc](#) ([cupy.cuda.texture.TextureObject](#) attribute), 458  
[TexDesc](#) ([cupy.cuda.texture.TextureReference](#) attribute), 460  
[texref](#) ([cupy.cuda.texture.TextureReference](#) attribute), 460  
[TextureDescriptor](#) (class in [cupy.cuda.texture](#)), 456  
[TextureObject](#) (class in [cupy.cuda.texture](#)), 457  
[TextureReference](#) (class in [cupy.cuda.texture](#)), 459  
[threadIdx](#) (in [module cupyx.jit](#)), 485  
[tile\(\)](#) (in [module cupy](#)), 110  
[time\\_range\(\)](#) (in [module cupy.prof](#)), 489  
[TimeRangeDecorator](#) (class in [cupy.prof](#)), 488  
[toarray\(\)](#) ([cupyx.scipy.sparse.coo\\_matrix](#) method), 349  
[toarray\(\)](#) ([cupyx.scipy.sparse.csc\\_matrix](#) method), 358  
[toarray\(\)](#) ([cupyx.scipy.sparse.csr\\_matrix](#) method), 367  
[toarray\(\)](#) ([cupyx.scipy.sparse.dia\\_matrix](#) method), 373  
[toarray\(\)](#) ([cupyx.scipy.sparse.spmatrix](#) method), 378  
[tobsr\(\)](#) ([cupyx.scipy.sparse.coo\\_matrix](#) method), 350  
[tobsr\(\)](#) ([cupyx.scipy.sparse.csc\\_matrix](#) method), 358  
[tobsr\(\)](#) ([cupyx.scipy.sparse.csr\\_matrix](#) method), 367  
[tobsr\(\)](#) ([cupyx.scipy.sparse.dia\\_matrix](#) method), 373  
[tobsr\(\)](#) ([cupyx.scipy.sparse.spmatrix](#) method), 378  
[tobytes\(\)](#) ([cupy.ndarray](#) method), 60  
[tocoo\(\)](#) ([cupyx.scipy.sparse.coo\\_matrix](#) method), 350  
[tocoo\(\)](#) ([cupyx.scipy.sparse.csc\\_matrix](#) method), 358  
[tocoo\(\)](#) ([cupyx.scipy.sparse.csr\\_matrix](#) method), 367  
[tocoo\(\)](#) ([cupyx.scipy.sparse.dia\\_matrix](#) method), 373  
[tocoo\(\)](#) ([cupyx.scipy.sparse.spmatrix](#) method), 378  
[tocsc\(\)](#) ([cupyx.scipy.sparse.coo\\_matrix](#) method), 350  
[tocsc\(\)](#) ([cupyx.scipy.sparse.csc\\_matrix](#) method), 358  
[tocsc\(\)](#) ([cupyx.scipy.sparse.csr\\_matrix](#) method), 367  
[tocsc\(\)](#) ([cupyx.scipy.sparse.dia\\_matrix](#) method), 374  
[tocsc\(\)](#) ([cupyx.scipy.sparse.spmatrix](#) method), 378  
[tocsr\(\)](#) ([cupyx.scipy.sparse.coo\\_matrix](#) method), 350  
[tocsr\(\)](#) ([cupyx.scipy.sparse.csc\\_matrix](#) method), 358  
[tocsr\(\)](#) ([cupyx.scipy.sparse.csr\\_matrix](#) method), 367  
[tocsr\(\)](#) ([cupyx.scipy.sparse.dia\\_matrix](#) method), 374  
[tocsr\(\)](#) ([cupyx.scipy.sparse.spmatrix](#) method), 378  
[todense\(\)](#) ([cupyx.scipy.sparse.coo\\_matrix](#) method), 350  
[todense\(\)](#) ([cupyx.scipy.sparse.csc\\_matrix](#) method), 359  
[todense\(\)](#) ([cupyx.scipy.sparse.csr\\_matrix](#) method), 368  
[todense\(\)](#) ([cupyx.scipy.sparse.dia\\_matrix](#) method), 374  
[todense\(\)](#) ([cupyx.scipy.sparse.spmatrix](#) method), 378  
[todia\(\)](#) ([cupyx.scipy.sparse.coo\\_matrix](#) method), 350  
[todia\(\)](#) ([cupyx.scipy.sparse.csc\\_matrix](#) method), 359  
[todia\(\)](#) ([cupyx.scipy.sparse.csr\\_matrix](#) method), 368  
[todia\(\)](#) ([cupyx.scipy.sparse.dia\\_matrix](#) method), 374

todia() (*cupyx.scipy.sparse.spmatrix method*), 378  
 toDlpack() (*cupy.ndarray method*), 59  
 todok() (*cupyx.scipy.sparse.coo\_matrix method*), 350  
 todok() (*cupyx.scipy.sparse.csc\_matrix method*), 359  
 todok() (*cupyx.scipy.sparse.csr\_matrix method*), 368  
 todok() (*cupyx.scipy.sparse.dia\_matrix method*), 374  
 todok() (*cupyx.scipy.sparse.spmatrix method*), 378  
 toeplitz() (*in module cupyx.scipy.linalg*), 300  
 tofile() (*cupy.ndarray method*), 60  
 tolil() (*cupyx.scipy.sparse.coo\_matrix method*), 350  
 tolil() (*cupyx.scipy.sparse.csc\_matrix method*), 359  
 tolil() (*cupyx.scipy.sparse.csr\_matrix method*), 368  
 tolil() (*cupyx.scipy.sparse.dia\_matrix method*), 374  
 tolil() (*cupyx.scipy.sparse.spmatrix method*), 378  
 tolist() (*cupy.ndarray method*), 60  
 tomaxint() (*cupy.random.RandomState method*), 207  
 total\_bytes() (*cupy.cuda.MemoryAsyncPool method*), 437  
 total\_bytes() (*cupy.cuda.MemoryPool method*), 435  
 trace() (*cupy.ndarray method*), 60  
 trace() (*in module cupy*), 157  
 transpose() (*cupy.ndarray method*), 60  
 transpose() (*cupyx.scipy.sparse.coo\_matrix method*), 350  
 transpose() (*cupyx.scipy.sparse.csc\_matrix method*), 359  
 transpose() (*cupyx.scipy.sparse.csr\_matrix method*), 368  
 transpose() (*cupyx.scipy.sparse.dia\_matrix method*), 374  
 transpose() (*cupyx.scipy.sparse.linalg.LinearOperator method*), 389  
 transpose() (*cupyx.scipy.sparse.spmatrix method*), 378  
 transpose() (*in module cupy*), 101  
 tri() (*in module cupy*), 97  
 tri() (*in module cupyx.scipy.linalg*), 300  
 triangular() (*cupy.random.RandomState method*), 207  
 triangular() (*in module cupy.random*), 229  
 tril() (*in module cupy*), 97  
 tril() (*in module cupyx.scipy.linalg*), 291  
 tril() (*in module cupyx.scipy.sparse*), 382  
 trim\_zeros() (*in module cupy*), 113  
 trimcoef() (*in module cupy.polynomial.polyutils*), 186  
 trimseq() (*in module cupy.polynomial.polyutils*), 185  
 triu() (*in module cupy*), 97  
 triu() (*in module cupyx.scipy.linalg*), 291  
 triu() (*in module cupyx.scipy.sparse*), 382  
 true\_divide() (*in module cupy*), 69  
 trunc() (*in module cupy*), 86  
 trunc() (*cupyx.scipy.sparse.coo\_matrix method*), 350  
 trunc() (*cupyx.scipy.sparse.csc\_matrix method*), 359  
 trunc() (*cupyx.scipy.sparse.csr\_matrix method*), 368  
 trunc() (*cupyx.scipy.sparse.dia\_matrix method*), 374  
 types (*cupy.ufunc attribute*), 66

## U

ufunc (*class in cupy*), 65  
 uniform() (*cupy.random.RandomState method*), 207  
 uniform() (*in module cupy.random*), 230  
 uniform\_filter() (*in module cupyx.scipy.ndimage*), 316  
 uniform\_filter1d() (*in module cupyx.scipy.ndimage*), 316  
 unique() (*in module cupy*), 112  
 UnownedMemory (*class in cupy.cuda*), 426  
 unpackbits() (*in module cupy*), 116  
 unravel\_index() (*in module cupy*), 136  
 unwrap() (*in module cupy*), 167  
 use() (*cupy.cuda.Device method*), 420  
 use() (*cupy.cuda.ExternalStream method*), 450  
 use() (*cupy.cuda.Stream method*), 448  
 used\_bytes() (*cupy.cuda.MemoryAsyncPool method*), 437  
 used\_bytes() (*cupy.cuda.MemoryPool method*), 435  
 using\_allocator() (*in module cupy.cuda*), 433

## V

values (*cupy.broadcast attribute*), 104  
 var() (*cupy.ndarray method*), 60  
 var() (*in module cupy*), 248  
 variable (*cupy.poly1d attribute*), 188  
 variance() (*in module cupyx.scipy.ndimage*), 331  
 vdot() (*in module cupy*), 149  
 vectorize (*class in cupy*), 131  
 view() (*cupy.ndarray method*), 60  
 vonmises() (*cupy.random.RandomState method*), 207  
 vonmises() (*in module cupy.random*), 230  
 vsplit() (*in module cupy*), 110  
 vstack() (*in module cupy*), 107  
 vstack() (*in module cupyx.scipy.sparse*), 384

## W

wait\_event() (*cupy.cuda.ExternalStream method*), 450  
 wait\_event() (*cupy.cuda.Stream method*), 448  
 wald() (*cupy.random.RandomState method*), 208  
 wald() (*in module cupy.random*), 231  
 weibull() (*cupy.random.RandomState method*), 208  
 weibull() (*in module cupy.random*), 231  
 where() (*in module cupy*), 133  
 white\_tophat() (*in module cupyx.scipy.ndimage*), 343  
 who() (*in module cupy*), 180  
 width (*cupy.cuda.texture.CUDAArray attribute*), 455  
 wiener() (*in module cupyx.scipy.signal*), 412

## X

XORWOW (*class in cupy.random*), 197

## Y

y0 (*in module cupyx.scipy.special*), 399

[y1 \(in module cupyx.scipy.special\)](#), [399](#)

## Z

[zeros\(\)](#) (in module *cupy*), [89](#)

[zeros\\_like\(\)](#) (in module *cupy*), [90](#)

[zeros\\_like\\_pinned\(\)](#) (in module *cupyx*), [417](#)

[zeros\\_pinned\(\)](#) (in module *cupyx*), [417](#)

[zeta \(in module cupyx.scipy.special\)](#), [404](#)

[zipf\(\)](#) (*cupy.random.RandomState* method), [208](#)

[zipf\(\)](#) (in module *cupy.random*), [232](#)

[zoom\(\)](#) (in module *cupyx.scipy.ndimage*), [323](#)