# CONTENTS

## 1 Overview

## 2 Installation Guide

2.1 Requirements ................................................. 3
2.2 Installing CuPy ............................................... 4
2.3 Installing CuPy from conda-forge ......................... 4
2.4 Installing CuPy from Source ............................... 5
2.5 Uninstalling CuPy ........................................... 6
2.6 Upgrading CuPy ............................................... 6
2.7 Reinstalling CuPy ............................................ 6
2.8 Using CuPy inside Docker ................................. 6
2.9 FAQ .......................................................... 7

## 3 Tutorial

3.1 Basics of CuPy ............................................... 9
3.2 User-Defined Kernels ....................................... 12

## 4 API Reference

4.1 Multi-Dimensional Array (ndarray) ......................... 19
4.2 Universal Functions (ufunc) ............................... 32
4.3 Routines ...................................................... 49
4.4 SciPy-compatible Routines ............................... 200
4.5 NumPy-CuPy Generic Code Support .................... 297
4.6 Memory Management ..................................... 297
4.7 Low-Level CUDA Support ............................... 300
4.8 Kernel binary memoization ............................... 338
4.9 Custom kernels ............................................. 338
4.10 Automatic Kernel Parameters Optimizations .......... 347
4.11 Interoperability .......................................... 347
4.12 Testing Modules ......................................... 350
4.13 Profiling .................................................. 363
4.14 Environment variables ................................. 365
4.15 Difference between CuPy and NumPy ................. 366
4.16 Comparison Table ....................................... 368
4.17 Miscellaneous functions ............................... 394

## 5 API Compatibility Policy

5.1 Versioning and Backward Compatibilities ............. 397
5.2 Processes to Break Backward Compatibilities ......... 397
5.3 Supported Backward Compatibility ..................... 398
5.4 Installation Compatibility .................................................. 399

6 Contribution Guide ......................................................... 401
   6.1 Classification of Contributions ....................................... 401
   6.2 Development Cycle ..................................................... 401
   6.3 Issues and Pull Requests ............................................... 403
   6.4 Coding Guidelines ...................................................... 404
   6.5 Unit Testing .............................................................. 406
   6.6 Documentation .......................................................... 408
   6.7 Tips for Developers .................................................... 409

7 Using CuPy on AMD GPU (experimental) ................................. 411
   7.1 Requirements ............................................................ 411
   7.2 Building CuPy for ROCm ............................................... 411

8 Upgrade Guide .............................................................. 413
   8.1 CuPy v8 ................................................................. 413
   8.2 CuPy v7 ................................................................. 414
   8.3 CuPy v6 ................................................................. 414
   8.4 CuPy v5 ................................................................. 414
   8.5 CuPy v4 ................................................................. 415
   8.6 CuPy v2 ................................................................. 416

9 License ......................................................................... 417
   9.1 NumPy ................................................................. 417
   9.2 SciPy ................................................................. 418

Python Module Index ........................................................................ 419

Index ....................................................................................... 421
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.
- **Reduction along axes** (`sum`, `max`, `argmax`, etc.)

CuPy also includes the following features for performance:

- User-defined elementwise CUDA kernels
- User-defined reduction CUDA kernels
- Fusing CUDA kernels to optimize user-defined calculation
- Customizable memory allocator and memory pool
- `cuDNN` utilities

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.
2.1 Requirements

The following Linux distributions are recommended.

- **Ubuntu** 18.04 LTS (x86_64)
- **CentOS** 7 (x86_64)

These components must be installed to use CuPy:

- **NVIDIA CUDA GPU** with the Compute Capability 3.0 or larger.
- **CUDA Toolkit**: v9.0 / v9.2 / v10.0 / v10.1 / v10.2 / v11.0 / v11.1
  - 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.
- **Python**: v3.5.1+ / v3.6.0+ / v3.7.0+ / v3.8.0+

**Note**: On Windows, CuPy only supports Python 3.6.0 or later.

2.1.1 Python Dependencies

NumPy/SciPy-compatible API in CuPy v8 is based on NumPy 1.19 and SciPy 1.5, and has been tested against the following versions:

- **NumPy**: v1.16 / v1.17 / v1.18 / v1.19
- **SciPy (optional)**: v1.3 / v1.4 / v1.5
  - Required only when using [SciPy-compatible Routines](#) (cupyx.scipy).
- **Optuna (optional)**: v2.x
  - Required only when using [Automatic Kernel Parameters Optimizations](#).

**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
  - The library to accelerate tensor operations. See *Environment variables* for the details.
- **NCCL**: v2.0 / v2.1 / v2.2 / v2.3 / v2.4 / v2.5 / v2.6 / v2.7
  - The library to perform collective multi-GPU / multi-node computations.
- **cuDNN**: v7.0 / v7.1 / v7.2 / v7.3 / v7.4 / v7.5 / v7.6 / v8.0
  - The library to accelerate deep neural network computations.

2.2 Installing CuPy

Wheels (precompiled binary packages) are available for Linux (Python 3.5+) and Windows (Python 3.6+). Package names are different depending on your CUDA Toolkit version.

<table>
<thead>
<tr>
<th>CUDA</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>v9.0</td>
<td>$ pip install cupy-cuda90</td>
</tr>
<tr>
<td>v9.2</td>
<td>$ pip install cupy-cuda92</td>
</tr>
<tr>
<td>v10.0</td>
<td>$ pip install cupy-cuda100</td>
</tr>
<tr>
<td>v10.1</td>
<td>$ pip install cupy-cuda101</td>
</tr>
<tr>
<td>v10.2</td>
<td>$ pip install cupy-cuda102</td>
</tr>
<tr>
<td>v11.0</td>
<td>$ pip install cupy-cuda110</td>
</tr>
<tr>
<td>v11.1</td>
<td>$ pip install cupy-cuda111</td>
</tr>
</tbody>
</table>

**Note:** Wheel packages are built with NCCL (Linux only) and cuDNN support enabled.

- NCCL library is bundled with these packages. You don’t have to install it manually.
- cuDNN library is bundled with these packages except for CUDA 11.0+. For CUDA 11.0+, you need to manually download and install cuDNN v8.0.x library to use cuDNN features.

**Note:** Use `pip install --pre cupy-cudaXXX` if you want to install prerelease (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:

```bash
$ 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 Linux 64 systems once the CUDA driver is correctly set up, you can install CuPy from the `conda-forge` channel:

```bash
```
and conda will install pre-built CuPy and most of the optional dependencies for you, including CUDA runtime libraries (cudatoolkit), NCCL, and cuDNN. It is not necessary to install CUDA Toolkit in advance. If you need to enforce the installation of a particular CUDA version (say 10.0) for driver compatibility, you can do:

```bash
$ conda install -c conda-forge cupy cudatoolkit=10.0
```

**Note:** Currently cuTENSOR is not yet available on conda-forge.

**Note:** If you encounter any problem with CuPy 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 yourselves, the nvcc compiler might not be available. The cudatoolkit package from Anaconda does not have nvcc included.

### 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.

```bash
$ pip install cupy
```

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

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

**Note:** To build the source tree downloaded from GitHub, you need to install Cython 0.28.0 or later (`pip install cython`). You don’t have to install Cython to build source packages hosted on PyPI as they include pre-generated...
C++ source files.

## 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:

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`, `LDFLAGS` and `LD_LIBRARY_PATH` environment variables before installing CuPy:

```
$ export CFLAGS=-I/path/to/cudnn/include  
$ export LDFLAGS=-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:

```bash
$ 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-9.0:

```bash
$ export CUDA_PATH=/usr/local/cuda-9.0
$ 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:**

```bash
$ 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:**

```bash
$ sudo yum install centos-release-scl
$ sudo yum install devtoolset-7-gcc-c++
$ source /opt/rh/devtoolset-7/enable
$ export NVCC="nvcc --compiler-bindir gcc-7"
```
3.1 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

3.1.1 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`, as `np` is `numpy` as is customarily done:

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

The `cupy.ndarray` class is in its core, which is a compatible GPU alternative of `numpy.ndarray`.

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

`x_gpu` in the above example is an instance of `cupy.ndarray`. You can see its creation of identical to NumPy’s one, except that `numpy` is replaced with `cupy`. The main difference of `cupy.ndarray` from `numpy.ndarray` is that the content is allocated on the device memory. Its data is allocated on the `current device`, which will be explained 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()` to calculate it on CPU.

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

We can calculate it on GPU with CuPy in a similar way:

```python
>>> 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. Understanding NumPy might help utilizing most features of CuPy. So, we recommend you to read the NumPy documentation.
3.1.2 Current Device

CuPy has a concept of the *current device*, which is the default device on which the allocation, manipulation, calculation etc. of arrays are taken place. Suppose the ID of current device is 0. The following code allocates array contents on GPU 0.

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

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

```python
>>> 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])
```

If you switch the current GPU temporarily, *with* statement comes in handy.

```python
>>> 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 operations of CuPy is done on the current device. Be careful that if processing of an array on a non-current device will cause an error:

```python
>>> 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):
...   current = 1
```

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

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

**Note:** If the environment has only one device, such explicit device switching is not needed.

3.1.3 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:

```python
>>> 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:

```python
>>> 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()`:

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

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

The compatibility of CuPy with NumPy enables us to write CPU/GPU generic code. It can be made easy by the `cupy.get_array_module()` function. This function returns the `numpy` or `cupy` module based on arguments.

A CPU/GPU generic function is defined using it like follows:

```python
>>> # Stable implementation of log(1 + exp(x))
>>> def softplus(x):
...     xp = cp.get_array_module(x)
...     return xp.maximum(0, x) + xp.log1p(xp.exp(-abs(x)))
```

Sometimes, an explicit conversion to a host or device array may be required. `cupy.asarray()` and `cupy.asnumpy()` can be used in agnostic implementations to get host or device arrays from either CuPy or NumPy arrays.

```python
>>> y_cpu = np.array([4, 5, 6])
>>> x_cpu + y_cpu
Traceback (most recent call last):
...
TypeError: Unsupported type <class 'numpy.ndarray'>
>>> cp.asnumpy(x_gpu) + y_cpu
array([5, 7, 9])
>>> x_gpu + cp.asnumpy(y_cpu)
array([5, 7, 9])
>>> cp.asarray(x_gpu) + cp.asarray(y_cpu)
array([5, 7, 9])
```
3.2 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.

3.2.1 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:

```python
>>> 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:

```python
>>> 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.],
>>> 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):

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

3.2.2 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:

```python
>>> 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:

```python
>>> 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:

```python
>>> 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.

### 3.2.3 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:

```python
>>> 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 stoded.

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()`.

### 3.2. User-Defined Kernels
3.2.4 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:

```python
>>> 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=cp.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.

3.2.5 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.

```python
>>> 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
```

(continues on next page)
Raw kernels operating on complex-valued arrays can be created as well:

```python
>>> 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)))

array([[ 0. +0.j, 1. +2.j, 2. +4.j, 3. +6.j, 4. +8.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:

```python
>>> 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.

**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.

### 3.2.6 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. 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.

```python
>>> 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]; }
...   }
... }"
```

(continues on next page)
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.)

### 3.2.7 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:

```python
>>> @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.

```python
>>> 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.
CuPy Documentation, Release 8.2.0

```python
>>> @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:

```python
>>> @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.
This is the official reference of CuPy, a multi-dimensional array on CUDA with a subset of NumPy interface.

- genindex
- modindex

4.1 Multi-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.

```python
class cupy.ndarray(shape, dtype=float, memptr=None, strides=None, order=u'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 (l'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__()**

```python
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

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

**__setitem__()**

```python
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.

```python
>>> 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,))
>>> i = cupy.arange(10000) % 2
>>> v = cupy.arange(10000).astype(cupy.float)
>>> a[i] = v
>>> a  # doctest: +SKIP
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__(self)

Return len(self).

__iter__(self)

Implement iter(self).

__copy__(self)

all(self, axis=None, out=None, keepdims=False) \to ndarray

any(self, axis=None, out=None, keepdims=False) \to ndarray

argmax(self, axis=None, out=None, dtype=None, keepdims=False) \to 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) \to 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=u'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:
```python
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:
```python
numpy.nonzero()
```

```python
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:
```python
cupy.partition() for full documentation, numpy.ndarray.partition()
```

```python
prod(self, axis=None, dtype=None, out=None, keepdims=None) → ndarray
Returns the product along a given axis.

See also:
```python
cupy.prod() for full documentation, numpy.ndarray.prod()
```

```python
ptp(self, axis=None, out=None, keepdims=False) → ndarray
Returns (maximum - minimum) along a given axis.

See also:
```python
cupy.ptp() for full documentation, numpy.ndarray.ptp()
```

```python
put(self, indices, values, mode='wrap')
Replaces specified elements of an array with given values.

See also:
```python
cupy.put() for full documentation, numpy.ndarray.put()
```

```python
ravel(self, order='C') → ndarray
Returns an array flattened into one dimension.

See also:
```python
cupy.ravel() for full documentation, numpy.ndarray.ravel()
```

```python
reduced_view(self, dtype=None) → ndarray
Returns a view of the array with minimum number of dimensions.
```

4.1. Multi-Dimensional Array (ndarray)
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.

`sor{t (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** `dltensor` *(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`

**Example**
```python
>>> 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=u'C') → bytes

Turns the array into a Python bytes object.

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

Writes the array to a file.

See also:

tobytes ()

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:

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 reinterpret 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 ()
Attributes

\textbf{T}
Shape-reversed view of the array.

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

\textbf{base}
\textbf{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 \texttt{cupy/carray.cuh}.

\textbf{data}
\textbf{device}
CUDA device on which this array resides.

\textbf{dtype}
\textbf{flags}
Object containing memory-layout information.

It only contains \texttt{c\_contiguous, f\_contiguous, and owndata} attributes. All of these are read-only. Accessing by indexes is also supported.

\textbf{See also:}
\texttt{numpy.ndarray.flags}

\textbf{flat}
\textbf{imag}
\textbf{itemsize}
Size of each element in bytes.

\textbf{See also:}
\texttt{numpy.ndarray.itemsize}

\textbf{nbytes}
Total size of all elements in bytes.

It does not count skips between elements.

\textbf{See also:}
\texttt{numpy.ndarray.nbytes}

\textbf{ndim}
Number of dimensions.

\text{a.ndim} is equivalent to \texttt{len(a.shape)}.

\textbf{See also:}
\texttt{numpy.ndarray.ndim}

\textbf{real}
\textbf{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
```

### 4.1.2 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.

```python

cupy.get_array_module

```

Returns the array module for arguments.

```python

cupy

```

or

```python

numpy

```

is returned based on the types of the arguments.

**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

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

```python

cupyx.scipy.get_array_module

```

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.
4.1.3 Conversion to/from NumPy arrays

cupy.ndarray and numpy.ndarray are not implicitly convertible to each other. That means, NumPy functions cannot take cupy.ndarray 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.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cupy.array</td>
<td>Creates an array on the current device.</td>
</tr>
<tr>
<td>cupy.asarray</td>
<td>Converts an object to array.</td>
</tr>
<tr>
<td>cupy.asnumpy</td>
<td>Returns an array on the host memory from an arbitrary source array.</td>
</tr>
</tbody>
</table>

**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`

4.2 Universal Functions (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`. 
4.2.1 Ufunc class

```python
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**

```python
__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.

**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 ‘->’.

4.2.2 Available ufuncs

Math operations

```python
cupy.add
```

Adds two arrays elementwise.
### Table 5 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cupy.subtract</code></td>
<td>Subtracts arguments elementwise.</td>
</tr>
<tr>
<td><code>cupy.multiply</code></td>
<td>Multiplies two arrays elementwise.</td>
</tr>
<tr>
<td><code>cupy.divide</code></td>
<td>Elementwise true division (i.e.</td>
</tr>
<tr>
<td><code>cupy.logaddexp</code></td>
<td>Computes ( \log(\exp(x_1) + \exp(x_2)) ) elementwise.</td>
</tr>
<tr>
<td><code>cupy.logaddexp2</code></td>
<td>Computes ( \log_2(\exp_2(x_1) + \exp_2(x_2)) ) elementwise.</td>
</tr>
<tr>
<td><code>cupy.true_divide</code></td>
<td>Elementwise true division (i.e.</td>
</tr>
<tr>
<td><code>cupy.floor_divide</code></td>
<td>Elementwise floor division (i.e.</td>
</tr>
<tr>
<td><code>cupy.negative</code></td>
<td>Takes numerical negative elementwise.</td>
</tr>
<tr>
<td><code>cupy.power</code></td>
<td>Computes ( x_1 ** x_2 ) elementwise.</td>
</tr>
<tr>
<td><code>cupy.remainder</code></td>
<td>Computes the remainder of Python division elementwise.</td>
</tr>
<tr>
<td><code>cupy.mod</code></td>
<td>Computes the remainder of Python division elementwise.</td>
</tr>
<tr>
<td><code>cupy.fmod</code></td>
<td>Computes the remainder of C division elementwise.</td>
</tr>
<tr>
<td><code>cupy.absolute</code></td>
<td>Elementwise absolute value function.</td>
</tr>
<tr>
<td><code>cupy.sign</code></td>
<td>Elementwise sign function.</td>
</tr>
<tr>
<td><code>cupy.exp</code></td>
<td>Elementwise exponential function.</td>
</tr>
<tr>
<td><code>cupy.exp2</code></td>
<td>Elementwise exponentiation with base 2.</td>
</tr>
<tr>
<td><code>cupy.log</code></td>
<td>Elementwise natural logarithm function.</td>
</tr>
<tr>
<td><code>cupy.log2</code></td>
<td>Elementwise binary logarithm function.</td>
</tr>
<tr>
<td><code>cupy.log10</code></td>
<td>Elementwise common logarithm function.</td>
</tr>
<tr>
<td><code>cupy.expm1</code></td>
<td>Computes ( \exp(x) - 1 ) elementwise.</td>
</tr>
<tr>
<td><code>cupy.log1p</code></td>
<td>Computes ( \log(1 + x) ) elementwise.</td>
</tr>
<tr>
<td><code>cupy.sqrt</code></td>
<td>Elementwise square root function.</td>
</tr>
<tr>
<td><code>cupy.square</code></td>
<td>Elementwise square function.</td>
</tr>
<tr>
<td><code>cupy.reciprocal</code></td>
<td>Computes ( 1 / x ) elementwise.</td>
</tr>
<tr>
<td><code>cupy.gcd</code></td>
<td>Computes ( \gcd(x_1, x_2) ) elementwise.</td>
</tr>
<tr>
<td><code>cupy.lcm</code></td>
<td>Computes ( \text{lcm}(x_1, x_2) ) elementwise.</td>
</tr>
</tbody>
</table>

**cupy.add**

\[ \text{cupy.add} = \text{ufunc}'\text{cupy_add}' \]

Adds two arrays elementwise.

See also:

`numpy.add`

**cupy.subtract**

\[ \text{cupy.subtract} = \text{ufunc}'\text{cupy_subtract}' \]

Subtracts arguments elementwise.

See also:

`numpy.subtract`
cupy.multiply

\[
\text{cupy.multiply} = \text{<ufunc 'cupy_multiply'>}
\]
Multiplies two arrays elementwise.

See also:

numpy.multiply

cupy.divide

\[
\text{cupy.divide} = \text{<ufunc 'cupy_true_divide'>}
\]
Elementwise true division (i.e. division as floating values).

See also:

numpy.true_divide

cupy.logaddexp

\[
\text{cupy.logaddexp} = \text{<ufunc 'cupy_logaddexp'>}
\]
Computes \(\log(\exp(x_1) + \exp(x_2))\) elementwise.

See also:

numpy.logaddexp

cupy.logaddexp2

\[
\text{cupy.logaddexp2} = \text{<ufunc 'cupy_logaddexp2'>}
\]
Computes \(\log_2(\exp_2(x_1) + \exp_2(x_2))\) elementwise.

See also:

numpy.logaddexp2

cupy.true_divide

\[
\text{cupy.true_divide} = \text{<ufunc 'cupy_true_divide'>}
\]
Elementwise true division (i.e. division as floating values).

See also:

numpy.true_divide

cupy.floor_divide

\[
\text{cupy.floor_divide} = \text{<ufunc 'cupy_floor_divide'>}
\]
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 $x_1 ^ {x_2}$ 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 'cupyrint'>
   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.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.reciprocal

cupy.reciprocal = <ufunc 'cupy_reciprocal'>
   Computes \( 1 / x \) elementwise.
   See also:
   numpy.reciprocal
cupy.gcd

```python
S = <ufunc 'cupy_gcd'>
```
Computes \( \gcd \) of \( x_1 \) and \( x_2 \) elementwise.

**See also:**

numpy.gcd

cupy.lcm

```python
S = <ufunc 'cupy_lcm'>
```
Computes \( \text{lcm} \) of \( x_1 \) and \( x_2 \) elementwise.

**See also:**

numpy.lcm

### Trigonometric functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cupy.sin</code></td>
<td>Elementwise sine function.</td>
</tr>
<tr>
<td><code>cupy.cos</code></td>
<td>Elementwise cosine function.</td>
</tr>
<tr>
<td><code>cupy.tan</code></td>
<td>Elementwise tangent function.</td>
</tr>
<tr>
<td><code>cupy.arcsin</code></td>
<td>Elementwise inverse-sine function (a.k.a. <code>asin</code>).</td>
</tr>
<tr>
<td><code>cupy.arccos</code></td>
<td>Elementwise inverse-cosine function (a.k.a. <code>acos</code>).</td>
</tr>
<tr>
<td><code>cupy.arctan</code></td>
<td>Elementwise inverse-tangent function (a.k.a. <code>atan</code>).</td>
</tr>
<tr>
<td><code>cupy.arctan2</code></td>
<td>Elementwise inverse-tangent of the ratio of two arrays.</td>
</tr>
<tr>
<td><code>cupy.hypot</code></td>
<td>Computes the hypoteneous of orthogonal vectors of given length.</td>
</tr>
<tr>
<td><code>cupy.sinh</code></td>
<td>Elementwise hyperbolic sine function.</td>
</tr>
<tr>
<td><code>cupy.cosh</code></td>
<td>Elementwise hyperbolic cosine function.</td>
</tr>
<tr>
<td><code>cupy.tanh</code></td>
<td>Elementwise hyperbolic tangent function.</td>
</tr>
<tr>
<td><code>cupy.arcsinh</code></td>
<td>Elementwise inverse of hyperbolic sine function.</td>
</tr>
<tr>
<td><code>cupy.arccosh</code></td>
<td>Elementwise inverse of hyperbolic cosine function.</td>
</tr>
<tr>
<td><code>cupy.arctanh</code></td>
<td>Elementwise inverse of hyperbolic tangent function.</td>
</tr>
<tr>
<td><code>cupy.deg2rad</code></td>
<td>Converts angles from degrees to radians elementwise.</td>
</tr>
<tr>
<td><code>cupy.rad2deg</code></td>
<td>Converts angles from radians to degrees elementwise.</td>
</tr>
</tbody>
</table>

cupy.sin

```python
S = <ufunc 'cupy_sin'>
```
Elementwise sine function.

**See also:**

numpy.sin

cupy.cos

```python
S = <ufunc 'cupy_cos'>
```
Elementwise cosine function.

**See also:**

numpy.cos
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 hypotenuse 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:

cupy.sinh

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

See also:

cupy.cosh

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

See also:

cupy.tanh

cupy.arcsinh
cupy.arcsinh = <ufunc 'cupy_arcsinh'>
Elementwise inverse of hyperbolic sine function.

See also:

cupy.arcsinh

cupy.arccosh
cupy.arccosh = <ufunc 'cupy_arccosh'>
Elementwise inverse of hyperbolic cosine function.

See also:

cupy.arccosh
CuPy Documentation, Release 8.2.0

**cupy.arctanh**

```python
cupy.arctanh = <ufunc 'cupy_arctanh'>
```

Elementwise inverse of hyperbolic tangent function.

See also:

numpy.arctanh

**cupy.deg2rad**

```python
cupy.deg2rad = <ufunc 'cupy_deg2rad'>
```

Converts angles from degrees to radians elementwise.

See also:

numpy.deg2rad, numpy.radians

**cupy.rad2deg**

```python
cupy.rad2deg = <ufunc 'cupy_rad2deg'>
```

Converts angles from radians to degrees elementwise.

See also:

numpy.rad2deg, numpy.degrees

**Bit-twiddling functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cupy.bitwise_and</td>
<td>Computes the bitwise AND of two arrays elementwise.</td>
</tr>
<tr>
<td>cupy.bitwise_or</td>
<td>Computes the bitwise OR of two arrays elementwise.</td>
</tr>
<tr>
<td>cupy.bitwise_xor</td>
<td>Computes the bitwise XOR of two arrays elementwise.</td>
</tr>
<tr>
<td>cupy.invert</td>
<td>Computes the bitwise NOT of an array elementwise.</td>
</tr>
<tr>
<td>cupy.left_shift</td>
<td>Shifts the bits of each integer element to the left.</td>
</tr>
<tr>
<td>cupy.right_shift</td>
<td>Shifts the bits of each integer element to the right.</td>
</tr>
</tbody>
</table>

**cupy.bitwise_and**

```python
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**

```python
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
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cupy.greater</td>
<td>Tests elementwise if ( x_1 &gt; x_2 ).</td>
</tr>
<tr>
<td>cupy.greater_equal</td>
<td>Tests elementwise if ( x_1 \geq x_2 ).</td>
</tr>
<tr>
<td>cupy.less</td>
<td>Tests elementwise if ( x_1 &lt; x_2 ).</td>
</tr>
<tr>
<td>cupy.less_equal</td>
<td>Tests elementwise if ( x_1 \leq x_2 ).</td>
</tr>
<tr>
<td>cupy.not_equal</td>
<td>Tests elementwise if ( x_1 \neq x_2 ).</td>
</tr>
<tr>
<td>cupy.equal</td>
<td>Tests elementwise if ( x_1 = x_2 ).</td>
</tr>
<tr>
<td>cupy.logical_and</td>
<td>Computes the logical AND of two arrays.</td>
</tr>
<tr>
<td>cupy.logical_or</td>
<td>Computes the logical OR of two arrays.</td>
</tr>
<tr>
<td>cupy.logical_xor</td>
<td>Computes the logical XOR of two arrays.</td>
</tr>
<tr>
<td>cupy.logical_not</td>
<td>Computes the logical NOT of an array.</td>
</tr>
<tr>
<td>cupy.maximum</td>
<td>Takes the maximum of two arrays elementwise.</td>
</tr>
<tr>
<td>cupy.minimum</td>
<td>Takes the minimum of two arrays elementwise.</td>
</tr>
<tr>
<td>cupy.fmax</td>
<td>Takes the maximum of two arrays elementwise.</td>
</tr>
<tr>
<td>cupy.fmin</td>
<td>Takes the minimum of two arrays elementwise.</td>
</tr>
</tbody>
</table>

**cupy.greater**

cupy.greater = <ufunc 'cupy_greater'>
Tests elementwise if \( x_1 > x_2 \).

See also:
numpy.greater

**cupy.greater_equal**

cupy.greater_equal = <ufunc 'cupy_greater_equal'>
Tests elementwise if \( x_1 \geq x_2 \).

See also:
numpy.greater_equal

**cupy.less**

cupy.less = <ufunc 'cupy_less'>
Tests elementwise if \( x_1 < x_2 \).

See also:
numpy.less

**cupy.less_equal**

cupy.less_equal = <ufunc 'cupy_less_equal'>
Tests elementwise if \( x_1 \leq x_2 \).

See also:
numpy.less_equal
cupy.not_equal

cupy\texttt{.not} _\texttt{equal} = \texttt{<ufunc 'cupy\_not\_equal'>}
 Tests elementwise if $x_1 \neq x_2$.

See also:

\texttt{numpy.equal}

cupy.equal

cupy\texttt{.equal} = \texttt{<ufunc 'cupy\_equal'>}
 Tests elementwise if $x_1 == x_2$.

See also:

\texttt{numpy.equal}

cupy.logical_and

cupy\texttt{.logical\_and} = \texttt{<ufunc 'cupy\_logical\_and'>}
 Computes the logical AND of two arrays.

See also:

\texttt{numpy.logical\_and}

cupy.logical_or

cupy\texttt{.logical\_or} = \texttt{<ufunc 'cupy\_logical\_or'>}
 Computes the logical OR of two arrays.

See also:

\texttt{numpy.logical\_or}

cupy.logical_xor

cupy\texttt{.logical\_xor} = \texttt{<ufunc 'cupy\_logical\_xor'>}
 Computes the logical XOR of two arrays.

See also:

\texttt{numpy.logical\_xor}

cupy.logical_not

cupy\texttt{.logical\_not} = \texttt{<ufunc 'cupy\_logical\_not'>}
 Computes the logical NOT of an array.

See also:

\texttt{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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cupy.isfinite</td>
<td>Tests finiteness elementwise.</td>
</tr>
<tr>
<td>cupy.isinf</td>
<td>Tests if each element is the positive or negative infinity.</td>
</tr>
<tr>
<td>cupy.isnan</td>
<td>Tests if each element is a NaN.</td>
</tr>
<tr>
<td>cupy.signbit</td>
<td>Tests elementwise if the sign bit is set (i.e.</td>
</tr>
<tr>
<td>cupy.copysign</td>
<td>Returns the first argument with the sign bit of the second elementwise.</td>
</tr>
<tr>
<td>cupy.nextafter</td>
<td>Computes the nearest neighbor float values towards the second argument.</td>
</tr>
</tbody>
</table>
Table 9 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cupy.modf</code></td>
<td>Extracts the fractional and integral parts of an array elementwise.</td>
</tr>
<tr>
<td><code>cupy.ldexp</code></td>
<td>Computes ( x_1 \times 2^{x_2} ) elementwise.</td>
</tr>
<tr>
<td><code>cupy.frexp</code></td>
<td>Decomposes each element to mantissa and two’s exponent.</td>
</tr>
<tr>
<td><code>cupy.fmod</code></td>
<td>Computes the remainder of C division elementwise.</td>
</tr>
<tr>
<td><code>cupy.floor</code></td>
<td>Rounds each element of an array to its floor integer.</td>
</tr>
<tr>
<td><code>cupy.ceil</code></td>
<td>Rounds each element of an array to its ceiling integer.</td>
</tr>
<tr>
<td><code>cupy.trunc</code></td>
<td>Rounds each element of an array towards zero.</td>
</tr>
</tbody>
</table>

**cupy.isfinite**

```python

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**

```python

cupy.isinf = <ufunc 'cupy_isinf'>
```

Tests if each element is the positive or negative infinity.

See also:

`numpy.isinf`

**cupy.isnan**

```python

cupy.isnan = <ufunc 'cupy_isnan'>
```

Tests if each element is a NaN.

See also:

`numpy.isnan`

**cupy.signbit**

```python

cupy.signbit = <ufunc 'cupy_signbit'>
```

Tests elementwise if the sign bit is set (i.e. less than zero).

See also:

`numpy.signbit`

**cupy.copysign**

```python

cupy.copysign = <ufunc 'cupy_copysign'>
```

Returns the first argument with the sign bit of the second elementwise.

4.2. Universal Functions (ufunc)
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 \( x_1 \times 2^{x_2} \) 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

4.2.3 ufunc.at

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

4.3 Routines

The following pages describe NumPy-compatible routines. These functions cover a subset of NumPy routines.

4.3.1 Array Creation Routines

Basic creation routines

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cupy.empty</td>
<td>Returns an array without initializing the elements.</td>
</tr>
<tr>
<td>cupy.empty_like</td>
<td>Returns a new array with same shape and dtype of a given array.</td>
</tr>
<tr>
<td>cupy.eye</td>
<td>Returns a 2-D array with ones on the diagonals and zeros elsewhere.</td>
</tr>
<tr>
<td>cupy.identity</td>
<td>Returns a 2-D identity array.</td>
</tr>
<tr>
<td>cupy.ones</td>
<td>Returns a new array of given shape and dtype, filled with ones.</td>
</tr>
<tr>
<td>cupy.ones_like</td>
<td>Returns an array of ones with same shape and dtype as a given array.</td>
</tr>
<tr>
<td>cupy.zeros</td>
<td>Returns a new array of given shape and dtype, filled with zeros.</td>
</tr>
<tr>
<td>cupy.zeros_like</td>
<td>Returns an array of zeros with same shape and dtype as a given array.</td>
</tr>
</tbody>
</table>
Table 10 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cupy.full</code></td>
<td>Returns a new array of given shape and dtype, filled with a given value.</td>
</tr>
<tr>
<td><code>cupy.full_like</code></td>
<td>Returns a full array with same shape and dtype as a given array.</td>
</tr>
</tbody>
</table>

**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 the `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.ones()`

**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.eye()`
- `cupy.ones()`

**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: `cupy.ndarray`

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()`

Creation from other data

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cupy.array</td>
<td>Creates an array on the current device.</td>
</tr>
<tr>
<td>cupy.asarray</td>
<td>Converts an object to array.</td>
</tr>
<tr>
<td>cupy.asanyarray</td>
<td>Converts an object to array.</td>
</tr>
<tr>
<td>cupy.ascontiguousarray</td>
<td>Returns a C-contiguous array.</td>
</tr>
<tr>
<td>cupy.copy</td>
<td>Creates a copy of a given array on the current device.</td>
</tr>
<tr>
<td>cupy.fromfile</td>
<td>Reads an array from a file.</td>
</tr>
</tbody>
</table>

**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
cupy.arange

Returns an array with evenly spaced values within a given interval.
cupy.linspace

Returns an array with evenly-spaced values within a given interval.
cupy.logspace

Returns an array with evenly-spaced values on a log-scale.
cupy.meshgrid

Return coordinate matrices from coordinate vectors.
cupy.mgrid

Construct a multi-dimensional "meshgrid".
cupy.ogrid

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 \([\text{start}, \text{stop})\). The first three arguments are mapped like the \textit{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

```python
cupy.mgrid = <cupy._creation.ranges.nd_grid object>
```
Construct a multi-dimensional “meshgrid”.

```python
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

```python
cupy.ogrid = <cupy._creation.ranges.nd_grid object>
```
Construct a multi-dimensional “meshgrid”.

```python
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`

Matrix creation

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cupy.diag</code></td>
<td>Returns a diagonal or a diagonal array.</td>
</tr>
<tr>
<td><code>cupy.diagflat</code></td>
<td>Creates a diagonal array from the flattened input.</td>
</tr>
<tr>
<td><code>cupy.tri</code></td>
<td>Creates an array with ones at and below the given diagonal.</td>
</tr>
<tr>
<td><code>cupy.tril</code></td>
<td>Returns a lower triangle of an array.</td>
</tr>
<tr>
<td><code>cupy.triu</code></td>
<td>Returns an upper triangle of an array.</td>
</tr>
</tbody>
</table>

cupy.diag

```python
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() *

4.3.2 Array Manipulation Routines

Basic operations

<table>
<thead>
<tr>
<th>cupy.copyto</th>
<th>Copies values from one array to another with broadcasting.</th>
</tr>
</thead>
<tbody>
<tr>
<td>cupy.shape</td>
<td>Returns the shape of an array</td>
</tr>
</tbody>
</table>

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.
CuPy Documentation, Release 8.2.0

• **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

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 (str) – 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()
```

4.3. Routines
cupy.ravel

cupy.ravel \(a, \text{order}='C'\)
Returns a flattened array.

It tries to return a view if possible, otherwise returns a copy.

This function currently does not support the \text{order} = 'K' option.

Parameters
- \(a\) (cupy.ndarray) – Array to be flattened.
- \text{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()

Transpose-like operations

cupy.moveaxis

cupy.moveaxis \((a, \text{source}, \text{destination})\)
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.
- \text{source} (int or sequence of int) – Original positions of the axes to move. These must be unique.
- \text{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.rolaxis

cupy.rolaxis(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.moveaxis()

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cupy.atleast_1d</td>
<td>Converts arrays to arrays with dimensions &gt;= 1.</td>
</tr>
<tr>
<td>cupy.atleast_2d</td>
<td>Converts arrays to arrays with dimensions &gt;= 2.</td>
</tr>
<tr>
<td>cupy.atleast_3d</td>
<td>Converts arrays to arrays with dimensions &gt;= 3.</td>
</tr>
<tr>
<td>cupy.broadcast</td>
<td>Object that performs broadcasting.</td>
</tr>
<tr>
<td>cupy.broadcast_to</td>
<td>Broadcast an array to a given shape.</td>
</tr>
<tr>
<td>cupy.broadcast_arrays</td>
<td>Broadcasts given arrays.</td>
</tr>
<tr>
<td>cupy.expand_dims</td>
<td>Expands given arrays.</td>
</tr>
<tr>
<td>cupy.squeeze</td>
<td>Removes size-one axes from the shape of an array.</td>
</tr>
</tbody>
</table>

#### cupy.atleast_1d

`cupy.atleast_1d(*arys)`

Converts arrays to arrays with dimensions >= 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 >= 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 >= 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**

<table>
<thead>
<tr>
<th>ary (tuple of arrays)</th>
<th>Arrays to be converted. All arguments must be cupy.ndarray objects.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Returns</strong></td>
<td>If there are only one input, then it returns its converted version. Otherwise, it returns a list of converted arrays.</td>
</tr>
</tbody>
</table>

**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**

| ary (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**

**Attributes**

nd
shape
dsize
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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cupy.asarray</code></td>
<td>Converts an object to array.</td>
</tr>
<tr>
<td><code>cupy.asanyarray</code></td>
<td>Converts an object to array.</td>
</tr>
<tr>
<td><code>cupy.asfortranarray</code></td>
<td>Return an array laid out in Fortran order in memory.</td>
</tr>
<tr>
<td><code>cupy.ascontiguousarray</code></td>
<td>Returns a C-contiguous array.</td>
</tr>
<tr>
<td><code>cupy.require</code></td>
<td>Return an array which satisfies the requirements.</td>
</tr>
</tbody>
</table>

**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 : \( \text{ndarray} \)

See also:

numpy.require()

Joining arrays

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cupy.concatenate</td>
<td>Joins arrays along an axis.</td>
</tr>
<tr>
<td>cupy.stack</td>
<td>Stacks arrays along a new axis.</td>
</tr>
<tr>
<td>cupy.column_stack</td>
<td>Stacks 1-D and 2-D arrays as columns into a 2-D array.</td>
</tr>
<tr>
<td>cupy.dstack</td>
<td>Stacks arrays along the third axis.</td>
</tr>
<tr>
<td>cupy.hstack</td>
<td>Stacks arrays horizontally.</td>
</tr>
<tr>
<td>cupy.vstack</td>
<td>Stacks arrays vertically.</td>
</tr>
</tbody>
</table>

\[ \text{cupy.concatenate}(tup, axis=0, out=None) \]

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 : \( \text{cupy.ndarray} \)

See also:

numpy.concatenate()

\[ \text{cupy.stack}(tup, axis=0, out=None) \]

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 : \( \text{cupy.ndarray} \)
See also:

numpy.stack()

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()

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.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.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()

### Splitting arrays

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cupy.split</td>
<td>Splits an array into multiple sub arrays along a given axis.</td>
</tr>
<tr>
<td>cupy.array_split</td>
<td>Splits an array into multiple sub arrays along a given axis.</td>
</tr>
<tr>
<td>cupy.dsplit</td>
<td>Splits an array into multiple sub arrays along the third axis.</td>
</tr>
<tr>
<td>cupy.hsplit</td>
<td>Splits an array into multiple sub arrays horizontally.</td>
</tr>
<tr>
<td>cupy.vsplit</td>
<td>Splits an array into multiple sub arrays along the first axis.</td>
</tr>
</tbody>
</table>

**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.dsplit`**

`cupy.dsplit(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.dsplit()`

**`cupy.hsplit`**

`cupy.hsplit(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.hsplit()`

**`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**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cupy.tile</code></td>
<td>Construct an array by repeating <code>A</code> the number of times given by <code>reps</code>.</td>
</tr>
<tr>
<td><code>cupy.repeat</code></td>
<td>Repeat arrays along an axis.</td>
</tr>
</tbody>
</table>

**`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

<table>
<thead>
<tr>
<th>cupy.unique</th>
<th>Find the unique elements of an array.</th>
</tr>
</thead>
<tbody>
<tr>
<td>cupy.trim_zeros</td>
<td>Trim the leading and/or trailing zeros from a 1-D array or sequence.</td>
</tr>
</tbody>
</table>

`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`. 

Chapter 4. API Reference
• `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**

<table>
<thead>
<tr>
<th>cupy.flip</th>
<th>Reverse the order of elements in an array along the given axis.</th>
</tr>
</thead>
<tbody>
<tr>
<td>cupy.fliplr</td>
<td>Flip array in the left/right direction.</td>
</tr>
<tr>
<td>cupy.flipud</td>
<td>Flip array in the up/down direction.</td>
</tr>
<tr>
<td>cupy.reshape</td>
<td>Returns an array with new shape and same elements.</td>
</tr>
</tbody>
</table>

Continued on next page
CuPy Documentation, Release 8.2.0

Table 23 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cupy.roll</code></td>
<td>Roll array elements along a given axis.</td>
</tr>
<tr>
<td><code>cupy.rot90</code></td>
<td>Rotate an array by 90 degrees in the plane specified by axes.</td>
</tr>
</tbody>
</table>

**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  

\textit{ndarray}

See also:

\texttt{numpy.flipud()}

\texttt{cupy.roll}

cupy.\texttt{roll}(a, \textit{shift}, \textit{axis}=\textit{None})

Roll array elements along a given axis.

Elements that roll beyond the last position are re-introduced at the first.

\begin{itemize}
  \item \texttt{a (ndarray)} – Array to be rolled.
  \item \texttt{shift (int or tuple of int)} – The number of places by which elements are shifted. If a tuple, then \textit{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 \textit{axis} is a tuple of ints, then the same value is used for all given axes.
  \item \texttt{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.
\end{itemize}

Returns  Output array.

Return type  \textit{ndarray}

See also:

\texttt{numpy.roll()}

\texttt{cupy.rot90}

cupy.\texttt{rot90}(a, \textit{k}=1, \textit{axes}=(0, 1))

Rotate an array by 90 degrees in the plane specified by axes.

Note that \texttt{axes} argument has been introduced since NumPy v1.12. The contents of this document is the same as the original one.

\begin{itemize}
  \item \texttt{a (ndarray)} – Array of two or more dimensions.
  \item \texttt{k (int)} – Number of times the array is rotated by 90 degrees.
  \item \texttt{axes} – (tuple of ints): The array is rotated in the plane defined by the axes. Axes must be different.
\end{itemize}

Returns  Output array.

Return type  \textit{ndarray}

See also:

\texttt{numpy.rot90()}

4.3. Routines  75
4.3.3 Binary Operations

Elementwise bit operations

<table>
<thead>
<tr>
<th>cupy.bitwise_and</th>
<th>Computes the bitwise AND of two arrays elementwise.</th>
</tr>
</thead>
<tbody>
<tr>
<td>cupy.bitwise_not</td>
<td>Computes the bitwise NOT of an array elementwise.</td>
</tr>
<tr>
<td>cupy.bitwise_or</td>
<td>Computes the bitwise OR of two arrays elementwise.</td>
</tr>
<tr>
<td>cupy.bitwise_xor</td>
<td>Computes the bitwise XOR of two arrays elementwise.</td>
</tr>
<tr>
<td>cupy.invert</td>
<td>Computes the bitwise NOT of an array elementwise.</td>
</tr>
<tr>
<td>cupy.left_shift</td>
<td>Shifts the bits of each integer element to the left.</td>
</tr>
<tr>
<td>cupy.right_shift</td>
<td>Shifts the bits of each integer element to the right.</td>
</tr>
</tbody>
</table>

**cupy.bitwise_not**

cupy.bitwise_not = <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

**Bit packing**

<table>
<thead>
<tr>
<th>cupy.packbits</th>
<th>Packs the elements of a binary-valued array into bits in a uint8 array.</th>
</tr>
</thead>
<tbody>
<tr>
<td>cupy.unpackbits</td>
<td>Unpacks elements of a uint8 array into a binary-valued output array.</td>
</tr>
</tbody>
</table>

**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

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()

4.3.4 Data Type Routines

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()`

**Creating data types**

`cupy.dtype(alias of numpy.dtype)`

`cupy.format_parser(alias of numpy.format_parser)`

**Data type information**

`cupy.finfo(alias of numpy.finfo)`

`cupy.iinfo(alias of numpy.iinfo)`

`cupy.MachAr(alias of numpy.MachAr)`

**Data type testing**

`cupy.issctype(alias of numpy.issctype())`

`cupy.issubdtype(alias of numpy.issubdtype())`

`cupy.issubctype(alias of numpy.issubdtype())`

`cupy.issubclass_(alias of numpy.issubclass_)`

`cupy.find_common_type(alias of numpy.find_common_type())`

**Miscellaneous**

`cupy.typename(alias of numpy.typename())`

`cupy.sctype2char(alias of numpy.sctype2char())`

`cupy.mintypecode(alias of numpy.mintypecode())`
4.3.5 FFT Functions

Standard FFTs

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cupy.fft.fft</code></td>
<td>Compute the one-dimensional FFT.</td>
</tr>
<tr>
<td><code>cupy.fft.ifft</code></td>
<td>Compute the one-dimensional inverse FFT.</td>
</tr>
<tr>
<td><code>cupy.fft.fft2</code></td>
<td>Compute the two-dimensional FFT.</td>
</tr>
<tr>
<td><code>cupy.fft.ifft2</code></td>
<td>Compute the two-dimensional inverse FFT.</td>
</tr>
<tr>
<td><code>cupy.fft.fftn</code></td>
<td>Compute the N-dimensional FFT.</td>
</tr>
<tr>
<td><code>cupy.fft.ifftn</code></td>
<td>Compute the N-dimensional inverse FFT.</td>
</tr>
</tbody>
</table>

**`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 (None or "ortho")`: Keyword to specify the normalization mode.

**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 (None or "ortho")`: Keyword to specify the normalization mode.

**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=None, 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** ([None or "ortho"]) – Keyword to specify the normalization mode.

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=None, 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** ([None or "ortho"]) – Keyword to specify the normalization mode.

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=None, 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** *(None or "ortho")* – Keyword to specify the normalization mode.

**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` *(None or "ortho")* – Keyword to specify the normalization mode.

**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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cupy.fft.rfft</td>
<td>Compute the one-dimensional FFT for real input.</td>
</tr>
<tr>
<td>cupy.fft.irfft</td>
<td>Compute the one-dimensional inverse FFT for real input.</td>
</tr>
<tr>
<td>cupy.fft.rfft2</td>
<td>Compute the two-dimensional FFT for real input.</td>
</tr>
<tr>
<td>cupy.fft.irfft2</td>
<td>Compute the two-dimensional inverse FFT for real input.</td>
</tr>
<tr>
<td>cupy.fft.rfftn</td>
<td>Compute the N-dimensional FFT for real input.</td>
</tr>
<tr>
<td>cupy.fft.irfftn</td>
<td>Compute the N-dimensional inverse FFT for real input.</td>
</tr>
</tbody>
</table>

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** (None or "ortho") – Keyword to specify the normalization mode.

**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** (None or "ortho") – Keyword to specify the normalization mode.

**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** (None or "ortho") – Keyword to specify the normalization mode.

**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 (None or "ortho")` – Keyword to specify the normalization mode.

**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 (None or "ortho")` – Keyword to specify the normalization mode.

**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 (None or "ortho")` – Keyword to specify the normalization mode.

**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()
```
• **`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 (None or "ortho")`** – Keyword to specify the normalization mode.

**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 $2 \times (m-1)$ where $m$ is the length of the final transformed axis of the input.

**Return type** *cupy.ndarray*

See also: *numpy.fft.irfftn()*

**Hermitian FFTs**

<table>
<thead>
<tr>
<th><code>cupy.fft.hfft</code></th>
<th>Compute the FFT of a signal that has Hermitian symmetry.</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cupy.fft.ihfft</code></td>
<td>Compute the FFT of a signal that has Hermitian symmetry.</td>
</tr>
</tbody>
</table>

**`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 (None or "ortho")`** – Keyword to specify the normalization mode.

**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 $2 \times (m-1)$ 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**
• \texttt{a (cupy.ndarray)} – Array to be transform.

• \texttt{n (None or int)} – Number of points along transformation axis in the input to use. If \texttt{n} is not given, the length of the input along the axis specified by \texttt{axis} is used.

• \texttt{axis (int)} – Axis over which to compute the FFT.

• \texttt{norm (None or ”ortho”)} – Keyword to specify the normalization mode.

\textbf{Returns} The transformed array which shape is specified by \texttt{n} and type will convert to complex if the input is other. The length of the transformed axis is \(n/2+1\).

\textbf{Return type} \texttt{cupy.ndarray}

\textbf{See also:} \texttt{numpy.fft.ihfft()}

\textbf{Helper routines}

\begin{tabular}{l|l}
\texttt{cupy.fft.fftfreq} & Return the FFT sample frequencies. \\
\texttt{cupy.fft.rfftfreq} & Return the FFT sample frequencies for real input. \\
\texttt{cupy.fft.fftfreq} & Shift the zero-frequency component to the center of the spectrum. \\
\texttt{cupy.fft.iftshift} & The inverse of \texttt{fftshift()}. \\
\texttt{cupy.fft.config.set_cufft_gpus} & Set the GPUs to be used in multi-GPU FFT. \\
\texttt{cupy.fft.config.get_plan_cache} & Get the per-thread, per-device plan cache, or create one if not found. \\
\texttt{cupy.fft.config.show_plan_cache_info} & Show all of the plan caches’ info on this thread. \\
\end{tabular}

\textbf{cupy.fft.fftfreq}

\texttt{cupy.fft.fftfreq(n, d=1.0)}

Return the FFT sample frequencies.

\textbf{Parameters}

• \texttt{n (int)} – Window length.

• \texttt{d (scalar)} – Sample spacing.

\textbf{Returns} Array of length \texttt{n} containing the sample frequencies.

\textbf{Return type} \texttt{cupy.ndarray}

\textbf{See also:} \texttt{numpy.fft.fftfreq()}

\textbf{cupy.fft.rfftfreq}

\texttt{cupy.fft.rfftfreq(n, d=1.0)}

Return the FFT sample frequencies for real input.

\textbf{Parameters}

• \texttt{n (int)} – Window length.

• \texttt{d (scalar)} – Sample spacing.
CuPy Documentation, Release 8.2.0

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**

`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.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 has the direct transforms unscaled and the inverse transforms are scaled by \(1/n\). If the keyword argument `norm` is "ortho", both transforms will be 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.
4.3.6 Functional programming

cupy.piecewise

Evaluate a piecewise-defined function.

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()

4.3.7 Indexing Routines

cupy.c_
cupy.r_
cupy.nonzero
Return the indices of the elements that are non-zero.
cupy.where
Return elements, either from x or y, depending on condition.
cupy.indices
Returns an array representing the indices of a grid.
cupy.ix_
Construct an open mesh from multiple sequences.
cupy.ravel_multi_index
Converts a tuple of index arrays into an array of flat indices, applying boundary modes to the multi-index.
cupy.unravel_index
Converts array of flat indices into a tuple of coordinate arrays.
cupy.take
Takes elements of an array at specified indices along an axis.
cupy.take_along_axis
Take values from the input array by matching 1d index and data slices.
cupy.choose
Returns selected slices of an array along given axis.
cupy.compress
Returns a diagonal or a diagonal array.
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cupy.diag_indices</code></td>
<td>Return the indices to access the main diagonal of an array.</td>
</tr>
<tr>
<td><code>cupy.diag_indices_from</code></td>
<td>Return the indices to access the main diagonal of an n-dimensional array.</td>
</tr>
<tr>
<td><code>cupy.diagonal</code></td>
<td>Returns specified diagonals.</td>
</tr>
<tr>
<td><code>cupy.extract</code></td>
<td>Return the elements of an array that satisfy some condition.</td>
</tr>
<tr>
<td><code>cupy.select</code></td>
<td>Return an array drawn from elements in choicelist, depending on conditions.</td>
</tr>
<tr>
<td><code>cupy.lib.stride_tricks.as_strided</code></td>
<td>Create a view into the array with the given shape and strides.</td>
</tr>
<tr>
<td><code>cupy.place</code></td>
<td>Change elements of an array based on conditional and strides.</td>
</tr>
<tr>
<td><code>cupy.put</code></td>
<td>Replaces specified elements of an array with given values.</td>
</tr>
<tr>
<td><code>cupy.putmask</code></td>
<td>Changes elements of an array inplace, based on a conditional mask and input values.</td>
</tr>
<tr>
<td><code>cupy.fill_diagonal</code></td>
<td>Fills the main diagonal of the given array of any dimensionality.</td>
</tr>
<tr>
<td><code>cupy.flatiter</code></td>
<td>Flat iterator object to iterate over arrays.</td>
</tr>
</tbody>
</table>

```python

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

```python
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**

```python
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**

```python
>>> 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

```python
g >>> a = cupy.arange(10).reshape(2, 5)
g >>> a
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
g >>> ixgrid = cupy.ix_([0,1], [2,4])
g >>> 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

```python
>>> 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:

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

```python
>>> 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.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.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:

```python
>>> 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:

```python
>>> 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:

```python
>>> 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()`

`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.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()

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.

---

**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`.  

---

4.3. Routines
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

```python
>>> 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**

```python
generate code here
```

If `values` is smaller than `a` it is repeated:

```python
generate code here
```

**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**

```python
generate code here
```

**See also:**

numpy.fill_diagonal()

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.

Attributes

base

A reference to the array that is iterated over.

4.3.8 Input and Output

NumPy binary files (NPY, NPZ)

<table>
<thead>
<tr>
<th>cupy.load</th>
<th>Loads arrays or pickled objects from .npy, .npz or pickled file.</th>
</tr>
</thead>
<tbody>
<tr>
<td>cupy.save</td>
<td>Saves an array to a binary file in .npy format.</td>
</tr>
<tr>
<td>cupy.savez</td>
<td>Saves one or more arrays into a file in uncompressed .npz format.</td>
</tr>
<tr>
<td>cupy.savez_compressed</td>
<td>Saves one or more arrays into a file in compressed .npz format.</td>
</tr>
</tbody>
</table>
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 npy 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, **kwds)

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.
- `**kwds` – Arrays with explicit keys.

*See also:*

`numpy.savez()`

**cupy.savez_compressed**

`cupy.savez_compressed(file, *args, **kwds)`

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**

| `cupy.array_repr` | Returns the string representation of an array. |
| `cupy.array_str`  | 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

```python
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**

- `cupy.binary_repr`
  Return the binary representation of the input number as a string.

- `cupy.base_repr`
  Return a string representation of a number in the given base system.

---

**cupy.base_repr**

```python
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()`

---

### 4.3.9 Linear Algebra

**Matrix and vector products**

- `cupy.cross`
  Returns the cross product of two vectors.

- `cupy.dot`
  Returns a dot product of two arrays.

- `cupy.vdot`
  Returns the dot product of two vectors.

- `cupy.inner`
  Returns the inner product of two arrays.

- `cupy.outer`
  Returns the outer product of two vectors.

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

- `cupy.tensordot`
  Returns the tensor dot product of two arrays along specified axes.

- `cupy.einsum`
  Evaluates the Einstein summation convention on the operands.

- `cupy.linalg.matrix_power`
  Raise a square matrix to the (integer) power $n$.

Continued on next page
Table 37 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cupy.kron</code></td>
<td>Returns the kronecker product of two arrays.</td>
</tr>
<tr>
<td><code>cupyx.scipy.linalg.kron</code></td>
<td>Kronecker product.</td>
</tr>
</tbody>
</table>

**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 \( \mathbb{R}^3 \) 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()

**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 Documentation, Release 8.2.0

**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**


**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**


**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**


**Returns** 2-D array of the outer product of `a` and `b`.

**Return type** *cupy.ndarray*

See also:

`numpy.outer()`

4.3. Routines
cupy.matmul

cupy.matmul(ndarray a, ndarray b, ndarray out=None) → ndarray

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**

- a (cupy.ndarray) – The left argument.
- b (cupy.ndarray) – The right argument.
- out (cupy.ndarray) – Output array.

**Returns**

Output array.

**Return type** cupy.ndarray

**See also:**

numpy.matmul()

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`

**Note:** `M` must be of dtype `float32` or `float64`.

**See also:** `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:

```python
numpy.kron()
```

cupyx.scipy.linalg.kron

cupyx.scipy.linalg.kron(a, b)

Kronecker product.

The result is the block matrix:

\[
\begin{array}{cccc}
a[0,0]\times b & a[0,1]\times b & \ldots & a[0,-1]\times b \\
a[1,0]\times b & a[1,1]\times b & \ldots & a[1,-1]\times b \\
\ldots & \ldots & \ldots & \ldots \\
a[-1,0]\times b & a[-1,1]\times b & \ldots & a[-1,-1]\times b \\
\end{array}
\]

Parameters

- a (cupy.ndarray) – Input array
- b (cupy.ndarray) – Input array

Returns Kronecker product of a and b.

Return type `cupy.ndarray`

See also:

```python
scipy.linalg.kron()
```

Decompositions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cupy.linalg.cholesky</td>
<td>Cholesky decomposition.</td>
</tr>
<tr>
<td>cupy.linalg.qr</td>
<td>QR decomposition.</td>
</tr>
<tr>
<td>cupy.linalg.svd</td>
<td>Singular Value Decomposition.</td>
</tr>
</tbody>
</table>

cupy.linalg.cholesky

cupy.linalg.cholesky(a)

Cholesky decomposition.

Decompose a given two-dimensional square matrix into \(L \times L^T\), where \(L\) is a lower-triangular matrix and \(L^T\) 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:

```python
numpy.linalg.cholesky()
```
cupy.linalg.qr

```python
cupy.linalg.qr(a, mode='reduced')
```

QR decomposition.

Decompose a given two-dimensional matrix into $Q \cdot R$, where $Q$ is an orthonormal and $R$ is an upper-triangular matrix.

**Parameters**

- `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

```python
cupy.linalg.svd(a, full_matrices=True, compute_uv=True)
```

Singular Value Decomposition.

Factorizes the matrix $A$ as $u \cdot \text{np.diag}(s) \cdot v$, 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 respectively $(M, K)$ and $(K, N)$, where $K = \min(M, N)$.
- `compute_uv` (*bool*) – If False, it only returns singular values.

**Returns**

A tuple of $(u, s, v)$ such that $A = u \cdot \text{np.diag}(s) \cdot v$.

**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()`.

See also:
`numpy.linalg.svd()`

Matrix eigenvalues

<table>
<thead>
<tr>
<th><code>cupy.linalg.eigh</code></th>
<th>Eigenvalues and eigenvectors of a symmetric matrix.</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cupy.linalg.eigvalsh</code></td>
<td>Calculates eigenvalues of a symmetric matrix.</td>
</tr>
</tbody>
</table>

`cupy.linalg.eigh`

cupy.linalg.eigh(a, UPLO='L')

Eigenvalues and eigenvectors of a given symmetric matrix.

This method calculates eigenvalues and eigenvectors of a given symmetric matrix.

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')

Calculates eigenvalues of a given symmetric matrix.

This method calculates eigenvalues a given symmetric matrix. Note that `cupy.linalg.eigh()` calculates both eigenvalues and eigenvectors.

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 etc.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cupy.linalg.det</td>
<td>Returns the determinant of an array.</td>
</tr>
<tr>
<td>cupy.linalg.norm</td>
<td>Returns one of matrix norms specified by ord parameter.</td>
</tr>
<tr>
<td>cupy.linalg.matrix_rank</td>
<td>Return matrix rank of array using SVD method</td>
</tr>
<tr>
<td>cupy.linalg.slogdet</td>
<td>Returns sign and logarithm of the determinant of an array.</td>
</tr>
<tr>
<td>cupy.trace</td>
<td>Returns the sum along the diagonals of an array.</td>
</tr>
</tbody>
</table>

**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.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 cumputed over axis.
- **keepdims (bool)** – If this is set True, the axes which are normed over are left.

Returns:
- cupy.ndarray

4.3. Routines 111
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()`

**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.slogdet()`

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 `(..., 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:** 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 linear equations

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cupy.linalg.solve</code></td>
<td>Solves a linear matrix equation.</td>
</tr>
<tr>
<td><code>cupy.linalg.tensorsolve</code></td>
<td>Solves tensor equations denoted by <code>ax = b</code>.</td>
</tr>
<tr>
<td><code>cupy.linalg.lstsq</code></td>
<td>Return the least-squares solution to a linear matrix equation.</td>
</tr>
<tr>
<td><code>cupy.linalg.inv</code></td>
<td>Computes the inverse of a matrix.</td>
</tr>
<tr>
<td><code>cupy.linalg.pinv</code></td>
<td>Compute the Moore-Penrose pseudoinverse of a matrix.</td>
</tr>
<tr>
<td><code>cupy.linalg.tensorinv</code></td>
<td>Computes the inverse of a tensor.</td>
</tr>
<tr>
<td><code>cupyx.scipy.linalg.lu_factor</code></td>
<td>LU decomposition.</td>
</tr>
<tr>
<td><code>cupyx.scipy.linalg.lu_solve</code></td>
<td>Solve an equation system, <code>a * x = b</code>, given the LU factorization of <code>a</code>.</td>
</tr>
<tr>
<td><code>cupyx.scipy.linalg.solve_triangular</code></td>
<td>Solve the equation <code>a x = b</code> for <code>x</code>, assuming <code>a</code> is a triangular matrix.</td>
</tr>
</tbody>
</table>

**cupy.linalg.solve**

```python
cupy.linalg.solve(a, b)
```

Solves a linear matrix equation.

It computes the exact solution of `x` in `ax = b`, where `a` is a square and full rank matrix.

Parameters

- **a (cupy.ndarray)** – The matrix with dimension `(..., M, M)`.
- **b (cupy.ndarray)** – The matrix with dimension `(..., M)` or `(..., M, K)`.

Returns

The matrix with dimension `(..., M)` or `(..., M, 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()`.

4.3. Routines

113
cupy.linalg.tensorsolve

cupy.linalg.tensorsolve(a, b, axes=None)

Solves tensor equations denoted by \( ax = b \).

Suppose that \( b \) is equivalent to \( \text{cupy.tensordot}(a, x) \). This function computes tensor \( x \) from \( a \) and \( b \).

**Parameters**

- \( a \) (cupy.ndarray) – The tensor with \( \text{len}(\text{shape}) \geq 1 \)
- \( b \) (cupy.ndarray) – The tensor with \( \text{len}(\text{shape}) \geq 1 \)
- \( \text{axes} \) (tuple of ints) – Axes in \( a \) to reorder to the right before inversion.

**Returns**
The tensor with shape \( Q \) such that \( b.\text{shape} + Q == a.\text{shape} \).

**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=1e-15)

Return the least-squares solution to a linear matrix equation.

Solves the equation \( a x = b \) by computing a vector \( x \) that minimizes the Euclidean 2-norm \( \| b - a x \|_2 \). The equation may be under-, well-, or over-determined (i.e., the number of linearly independent rows of \( a \) can be less than, equal to, or greater than its number of linearly independent columns). If \( a \) is square and of full rank, then \( x \) (but for round-off error) is the “exact” solution of the equation.

**Parameters**

- \( a \) (cupy.ndarray) – “Coefficient” matrix with dimension \((M, N)\)
- \( b \) (cupy.ndarray) – “Dependent variable” values with dimension \((M, ) \) or \((M, K)\)
- \( rcond \) (float) – Cutoff parameter for small singular values. For stability it computes the largest singular value denoted by \( s \), and sets all singular values smaller than \( s \) to zero.

**Returns**
A tuple of \((x, \text{residuals}, \text{rank}, s)\). Note \( x \) is the least-squares solution with shape \((N, )\) or \((N, K)\) depending if \( b \) was two-dimensional. The sums of \( \text{residuals} \) is the squared Euclidean 2-norm for each column in \( b - a x \). The \( \text{residuals} \) is an empty array if the rank of \( a \) is \(< N \) or \( M \leq N \), but if \( b \) is 1-dimensional, this is a \((1, )\) shape array. Otherwise the shape is \((K, )\). The \( \text{rank} \) 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 (M, N)
- rcond (float) – Cutoff parameter for small singular values. For stability it computes the largest singular value denoted by s, and sets all singular values smaller than s to zero.

Returns

The pseudoinverse of a with dimension (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 \( a^{-1} \) from tensor \( a \) such that \( \text{tensordot}(a^{-1}, a, \text{ind}) = I \), where \( I \) denotes the identity tensor.

Parameters

- **a** (cupy.ndarray) – The tensor such that \( \text{prod}(a.shape[:\text{ind}]) = \text{prod}(a.shape[\text{ind}:]) \).
- **ind** (int) – The positive number used in axes option of tensordot.

Returns The inverse of a tensor whose shape is equivalent to \( a.shape[\text{ind}:] + a.shape[:\text{ind}] \).

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()

cupy.scipy.linalg.lu_factor

cupy.scipy.linalg.lu_factor(a, overwrite_a=False, check_finite=True)
LU decomposition.

Decompose a given two-dimensional square matrix into \( P \cdot L \cdot U \), where \( P \) is a permutation matrix, \( L \) lower-triangular with unit diagonal elements, and \( U \) upper-triangular matrix. Note that in the current implementation \( a \) must be a real matrix, and only numpy.float32 and numpy.float64 are supported.

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()

**Note:** Current implementation returns result different from SciPy when the matrix singular. SciPy returns an array containing 0. while the current implementation returns an array containing nan.

```python
>>> import numpy as np
>>> import scipy.linalg

scipy.linalg.lu_factor(np.array([[0, 1], [0, 0]], dtype=np.float32))
(array([[0., 1.],
        [0., 0.]], dtype=float32), array([0, 1], dtype=int32))
```

```python
>>> import cupy as cp
>>> import cupyx.scipy.linalg

cupyx.scipy.linalg.lu_factor(cp.array([[0, 1], [0, 0]], dtype=cp.float32))
(array([[ 0., 1.],
        [nan, nan]], dtype=float32), array([0, 1], dtype=int32))
```

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, \( a \times x = b \), 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:

<table>
<thead>
<tr>
<th>trans</th>
<th>system</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(a \times x = b)</td>
</tr>
<tr>
<td>1</td>
<td>(a^T \times x = b)</td>
</tr>
<tr>
<td>2</td>
<td>(a^H \times x = b)</td>
</tr>
</tbody>
</table>

- **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()
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'` – `a x = 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()`

### Special Matrices

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cupy.tri</code></td>
<td>Creates an array with ones at and below the given diagonal.</td>
</tr>
<tr>
<td><code>cupy.tril</code></td>
<td>Returns a lower triangle of an array.</td>
</tr>
<tr>
<td><code>cupy.triu</code></td>
<td>Returns an upper triangle of an array.</td>
</tr>
<tr>
<td><code>cupyx.scipy.linalg.tri</code></td>
<td>Construct <code>(N, M)</code> matrix filled with ones at and below the <code>k</code>-th diagonal.</td>
</tr>
<tr>
<td><code>cupyx.scipy.linalg.tril</code></td>
<td>Make a copy of a matrix with elements above the <code>k</code>-th diagonal zeroed.</td>
</tr>
<tr>
<td><code>cupyx.scipy.linalg.triu</code></td>
<td>Make a copy of a matrix with elements below the <code>k</code>-th diagonal zeroed.</td>
</tr>
<tr>
<td><code>cupyx.scipy.linalg.toeplitz</code></td>
<td>Construct a Toeplitz matrix.</td>
</tr>
<tr>
<td><code>cupyx.scipy.linalg.circulant</code></td>
<td>Construct a circulant matrix.</td>
</tr>
<tr>
<td><code>cupyx.scipy.linalg.hankel</code></td>
<td>Construct a Hankel matrix.</td>
</tr>
<tr>
<td><code>cupyx.scipy.linalg.hadamard</code></td>
<td>Construct an Hadamard matrix.</td>
</tr>
<tr>
<td><code>cupyx.scipy.linalg.leslie</code></td>
<td>Create a Leslie matrix.</td>
</tr>
<tr>
<td><code>cupyx.scipy.linalg.block_diag</code></td>
<td>Create a block diagonal matrix from provided arrays.</td>
</tr>
</tbody>
</table>

*Continued on next page*
### Table 42 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cupyx.scipy.linalg.companion</code></td>
<td>Create a companion matrix.</td>
</tr>
<tr>
<td><code>cupyx.scipy.linalg.helmert</code></td>
<td>Create an Helmert matrix of order ( n ).</td>
</tr>
<tr>
<td><code>cupyx.scipy.linalg.hilbert</code></td>
<td>Create a Hilbert matrix of order ( n ).</td>
</tr>
<tr>
<td><code>cupyx.scipy.linalg.dft</code></td>
<td>Discrete Fourier transform matrix.</td>
</tr>
<tr>
<td><code>cupyx.scipy.linalg.fiedler</code></td>
<td>Returns a symmetric Fiedler matrix.</td>
</tr>
<tr>
<td><code>cupyx.scipy.linalg.fiedler_companion</code></td>
<td>Returns a Fiedler companion matrix.</td>
</tr>
<tr>
<td><code>cupyx.scipy.linalg.convolution_matrix</code></td>
<td>Construct a convolution matrix.</td>
</tr>
</tbody>
</table>

**cupyx.scipy.linalg.tri**

The function `cupyx.scipy.linalg.tri(N, M=None, k=0, dtype=None)` constructs an \((N, M)\) 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()`

**cupyx.scipy.linalg.tril**

The function `cupyx.scipy.linalg.tril(m, k=0)` makes 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()`

4.3. Routines 119
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()

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:
cupy.scipy.linalg.circulant()
See also:
cupy.scipy.linalg.hankel()
See also:
cupy.scipy.linalg.solve_toeplitz()
See also:
cupy.scipy.linalg.fiedler()
See also:
scipy.linalg.toeplitz()
cupyx.scipy.linalg.circulant

```python

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.hankel

```python

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.hadamard

```
cupy.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.leslie

```
cupy.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.block_diag

```
cupy.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**, **B**, **C**, .. (A,) -- 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()

cupy.scipy.linalg.companion

cupy.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:

cupy.scipy.linalg.fiedler_companion()

See also:

scipy.linalg.companion()

cupy.scipy.linalg.helmert

cupy.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.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, ‘sqrtn’, or ‘n’. If \(scale\) is ‘sqrtn’, 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()`
- `cupyx.scipy.linalg.toeplitz()`
- `scipy.linalg.fiedler()`

**cupyx.scipy.linalg.fiedler_companion**

cupy.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()`
- `scipy.linalg.fiedler_companion()`

**cupyx.scipy.linalg.convolution_matrix**

cupy.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)`.

4.3. Routines
Returns

The convolution matrix whose row count depends on

\[
\text{mode:} \quad 'full' \; m + n - 1 \quad 'same' \; \max(m, n) \quad 'valid' \; \max(m, n) - \min(m, n) + 1
\]

Return type **cupy.ndarray**

See also:

* cupyx.scipy.linalg.toeplitz()
* scipy.linalg.convolution_matrix()

### 4.3.10 Logic Functions

**Truth value testing**

<table>
<thead>
<tr>
<th><strong>cupy.all</strong></th>
<th>Tests whether all array elements along a given axis evaluate to True.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>cupy.any</strong></td>
<td>Tests whether any array elements along a given axis evaluate to True.</td>
</tr>
<tr>
<td><strong>cupy.in1d</strong></td>
<td>Tests whether each element of a 1-D array is also present in a second array.</td>
</tr>
<tr>
<td><strong>cupy.isin</strong></td>
<td>Calculates element in test_elements, broadcasting over element only.</td>
</tr>
</tbody>
</table>

**cupy.all**

\[ \text{cupy.all}(a, axis=None, out=None, keepdims=False) \]

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**

\[ \text{cupy.any}(a, axis=None, out=None, keepdims=False) \]

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()`

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`.

4.3. Routines
Return type  \textit{cupy.ndarray}, bool

\textbf{Infinities and NaNs}

\begin{tabular}{ll}
\textit{cupy.isfinite} & Tests finiteness elementwise. \\
\textit{cupy.isinf} & Tests if each element is the positive or negative infinity. \\
\textit{cupy.isnan} & Tests if each element is a NaN. \\
\end{tabular}

\textbf{Array type testing}

\begin{tabular}{ll}
\textit{cupy.iscomplex} & Returns a bool array, where True if input element is complex. \\
\textit{cupy.iscomplexobj} & Check for a complex type or an array of complex numbers. \\
\textit{cupy.isfortran} & Returns True if the array is Fortran contiguous but \textit{not} C contiguous. \\
\textit{cupy.isreal} & Returns a bool array, where True if input element is real. \\
\textit{cupy.isrealobj} & Return True if \( x \) is a not complex type or an array of complex numbers. \\
\textit{cupy.isscalar} & Returns True if the type of num is a scalar type. \\
\end{tabular}

\textbf{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 \( x \) (\textit{cupy.ndarray}) – Input array.

Returns  Boolean array of the same shape as \( x \).

Return type  \textit{cupy.ndarray}

See also:  \textit{isreal()}, \textit{iscomplexobj()}

\textbf{Examples}

```python
>>> cupy.iscomplex(cupy.array([1+1j, 1+0j, 4.5, 3, 2, 2j]))
array([ True, False, False, False, False,  True])
```

\textbf{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, \textit{iscomplexobj} evaluates to True.

Parameters \( x \) (\textit{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

```python
>>> 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).

```python
>>> 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.

```python
>>> a = cupy.array([[1, 2, 3], [4, 5, 6]], order='C')
>>> a
array([[1, 2, 3],)
       [4, 5, 6]])
```

(continues on next page)
>>> cupy.isfortran(a)
False
>>> b = a.T
>>> b
array([[1, 4],
       [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**

```python
>>> 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

```python
>>> 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:

cupy.isscalar

Logic operations

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cupy.logical_and</td>
<td>Computes the logical AND of two arrays.</td>
</tr>
<tr>
<td>cupy.logical_or</td>
<td>Computes the logical OR of two arrays.</td>
</tr>
<tr>
<td>cupy.logical_not</td>
<td>Computes the logical NOT of an array.</td>
</tr>
<tr>
<td>cupy.logical_xor</td>
<td>Computes the logical XOR of two arrays.</td>
</tr>
</tbody>
</table>

Comparison

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cupy.allclose</td>
<td>Returns True if two arrays are element-wise equal within a tolerance.</td>
</tr>
<tr>
<td>cupy.array_equal</td>
<td>Returns True if two arrays are element-wise exactly equal.</td>
</tr>
<tr>
<td>cupy.isclose</td>
<td>Returns a boolean array where two arrays are equal within a tolerance.</td>
</tr>
<tr>
<td>cupy.greater</td>
<td>Tests elementwise if ( x_1 &gt; x_2 ).</td>
</tr>
<tr>
<td>cupy.greater_equal</td>
<td>Tests elementwise if ( x_1 \geq x_2 ).</td>
</tr>
<tr>
<td>cupy.less</td>
<td>Tests elementwise if ( x_1 &lt; x_2 ).</td>
</tr>
<tr>
<td>cupy.less_equal</td>
<td>Tests elementwise if ( x_1 \leq x_2 ).</td>
</tr>
<tr>
<td>cupy.equal</td>
<td>Tests elementwise if ( x_1 = x_2 ).</td>
</tr>
<tr>
<td>cupy.not_equal</td>
<td>Tests elementwise if ( x_1 \neq x_2 ).</td>
</tr>
</tbody>
</table>

cupy.allclose
cupy.allclose(a, b, atol=1e-05, rtol=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.

\[
|a - b| \leq \text{atol} + \text{rtol}|b|
\]

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 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.array_equal

cuPy.array_equal(a1, a2)

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.

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()

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()

4.3.11 Mathematical Functions

Trigonometric functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cupy.sin</td>
<td>Elementwise sine function.</td>
</tr>
<tr>
<td>cupy.cos</td>
<td>Elementwise cosine function.</td>
</tr>
<tr>
<td>cupy.tan</td>
<td>Elementwise tangent function.</td>
</tr>
<tr>
<td>cupy.arcsin</td>
<td>Elementwise inverse-sine function (a.k.a.</td>
</tr>
<tr>
<td>cupy.arccos</td>
<td>Elementwise inverse-cosine function (a.k.a.</td>
</tr>
<tr>
<td>cupy.arctan</td>
<td>Elementwise inverse-tangent function (a.k.a.</td>
</tr>
<tr>
<td>cupy.hypot</td>
<td>Computes the hypoteneous of orthogonal vectors of given length.</td>
</tr>
<tr>
<td>cupy.arctan2</td>
<td>Elementwise inverse-tangent of the ratio of two arrays.</td>
</tr>
<tr>
<td>cupy.degrees</td>
<td>Converts angles from radians to degrees elementwise.</td>
</tr>
<tr>
<td>cupy.radians</td>
<td>Converts angles from degrees to radians elementwise.</td>
</tr>
<tr>
<td>cupy.unwrap</td>
<td>Unwrap by changing deltas between values to 2*pi complement.</td>
</tr>
<tr>
<td>cupy.deg2rad</td>
<td>Converts angles from degrees to radians elementwise.</td>
</tr>
<tr>
<td>cupy.rad2deg</td>
<td>Converts angles from radians to degrees elementwise.</td>
</tr>
</tbody>
</table>

**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.unwrap**

```
cupy.unwrap(p, discont=3.141592653589793, axis=-1)
```

Unwrap by changing deltas between values to 2*pi complement.

Parameters

- **p (cupy.ndarray)** – Input array.
- **discont (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**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cupy.sinh</code></td>
<td>Elementwise hyperbolic sine function.</td>
</tr>
<tr>
<td><code>cupy.cosh</code></td>
<td>Elementwise hyperbolic cosine function.</td>
</tr>
<tr>
<td><code>cupy.tanh</code></td>
<td>Elementwise hyperbolic tangent function.</td>
</tr>
<tr>
<td><code>cupy.arcsinh</code></td>
<td>Elementwise inverse of hyperbolic sine function.</td>
</tr>
<tr>
<td><code>cupy.arccosh</code></td>
<td>Elementwise inverse of hyperbolic cosine function.</td>
</tr>
<tr>
<td><code>cupy.arctanh</code></td>
<td>Elementwise inverse of hyperbolic tangent function.</td>
</tr>
</tbody>
</table>

**Rounding**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cupy.around</code></td>
<td>Rounds to the given number of decimals.</td>
</tr>
<tr>
<td><code>cupy.round_</code></td>
<td>Rounds each element of an array to the nearest integer.</td>
</tr>
<tr>
<td><code>cupy.rint</code></td>
<td>If given value x is positive, it return floor(x).</td>
</tr>
<tr>
<td><code>cupy.fix</code></td>
<td>Rounds each element of an array to its floor integer.</td>
</tr>
<tr>
<td><code>cupy.ceil</code></td>
<td>Rounds each element of an array to its ceiling integer.</td>
</tr>
<tr>
<td><code>cupy.trunc</code></td>
<td>Rounds each element of an array towards zero.</td>
</tr>
</tbody>
</table>

**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 Documentation, Release 8.2.0

CuPy Documentation, Release 8.2.0

**cupy.fix**

```python
import cupy

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cupy.prod</code></td>
<td>Returns the product of an array along given axes.</td>
</tr>
<tr>
<td><code>cupy.sum</code></td>
<td>Returns the sum of an array along given axes.</td>
</tr>
<tr>
<td><code>cupy.cumprod</code></td>
<td>Returns the cumulative product of an array along a given axis.</td>
</tr>
<tr>
<td><code>cupy.cumsum</code></td>
<td>Returns the cumulative sum of an array along a given axis.</td>
</tr>
<tr>
<td><code>cupy.nansum</code></td>
<td>Returns the sum of an array along given axes treating Not a Numbers (NaNs) as zero.</td>
</tr>
<tr>
<td><code>cupy.nanprod</code></td>
<td>Returns the product of an array along given axes treating Not a Numbers (NaNs) as zero.</td>
</tr>
<tr>
<td><code>cupy.diff</code></td>
<td>Calculate the n-th discrete difference along the given axis.</td>
</tr>
</tbody>
</table>

**cupy.prod**

```python
import cupy

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**

```python
import cupy

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.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.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.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.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.
CuPy Documentation, Release 8.2.0

- `append(int, float, cupy.ndarray) – Value to append to a.

  Returns: The result array.

  Return type: `cupy.ndarray`

  See also:

  `numpy.diff()`

Exponents and logarithms

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cupy.exp</code></td>
<td>Elementwise exponential function.</td>
</tr>
<tr>
<td><code>cupy.expm1</code></td>
<td>Computes ( \exp(x) - 1 ) elementwise.</td>
</tr>
<tr>
<td><code>cupy.exp2</code></td>
<td>Elementwise exponentiation with base 2.</td>
</tr>
<tr>
<td><code>cupy.log</code></td>
<td>Elementwise natural logarithm function.</td>
</tr>
<tr>
<td><code>cupy.log10</code></td>
<td>Elementwise common logarithm function.</td>
</tr>
<tr>
<td><code>cupy.log2</code></td>
<td>Elementwise binary logarithm function.</td>
</tr>
<tr>
<td><code>cupy.logaddexp</code></td>
<td>Computes ( \log(\exp(x1) + \exp(x2)) ) elementwise.</td>
</tr>
<tr>
<td><code>cupy.logaddexp2</code></td>
<td>Computes ( \log2(\exp2(x1) + \exp2(x2)) ) elementwise.</td>
</tr>
</tbody>
</table>

Other special functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cupy.i0</code></td>
<td>Modified Bessel function of the first kind, order 0.</td>
</tr>
<tr>
<td><code>cupy.sinc</code></td>
<td>Elementwise sinc function.</td>
</tr>
</tbody>
</table>

`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

`cupy.signbit`

Tests elementwise if the sign bit is set (i.e.
Table 54 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cupy.copysign</code></td>
<td>Returns the first argument with the sign bit of the second elementwise.</td>
</tr>
<tr>
<td><code>cupy.frexp</code></td>
<td>Decomposes each element to mantissa and two’s exponent.</td>
</tr>
<tr>
<td><code>cupy.ldexp</code></td>
<td>Computes $x_1 * 2 ^ {x_2}$ elementwise.</td>
</tr>
<tr>
<td><code>cupy.nextafter</code></td>
<td>Computes the nearest neighbor float values towards the second argument.</td>
</tr>
</tbody>
</table>

Arithmetic operations

- `cupy.add` Adds two arrays elementwise.
- `cupy.reciprocal` Computes $1 / x$ elementwise.
- `cupy.negative` Takes numerical negative elementwise.
- `cupy.multiply` Multiplies two arrays elementwise.
- `cupy.divide` Elementwise true division (i.e., $x_1 / x_2$).
- `cupy.power` Computes $x_1 ^ {x_2}$ elementwise.
- `cupy.subtract` Subtracts arguments elementwise.
- `cupy.true_divide` Elementwise true division (i.e., $x_1 / x_2$).
- `cupy.floor_divide` Elementwise floor division (i.e., $x_1 \div x_2$).
- `cupy.fmod` Computes the remainder of C division elementwise.
- `cupy.mod` Computes the remainder of Python division elementwise.
- `cupy.modf` Extracts the fractional and integral parts of an array elementwise.
- `cupy.remainder` Computes the remainder of Python division elementwise.
- `cupy.divmod` Computes the remainder of Python division elementwise.

`cupy.divmod`

`cupy.divmod = <ufunc 'cupy_divmod'>`

Handling complex numbers

- `cupy.angle` Returns the angle of the complex argument.
- `cupy.real` Returns the real part of the elements of the array.
- `cupy.imag` Returns the imaginary part of the elements of the array.
- `cupy.conj` 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 Documentation, Release 8.2.0

**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()

**cupy.conj**

cupy.conj = <ufunc 'cupy_conjugate'>

Returns the complex conjugate, element-wise.

See also:

numpy.conjugate

**Miscellaneous**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cupy.convolve</td>
<td>Returns the discrete, linear convolution of two one-dimensional sequences.</td>
</tr>
<tr>
<td>cupy.clip</td>
<td>Clips the values of an array to a given interval.</td>
</tr>
<tr>
<td>cupy.sqrt</td>
<td>Elementwise square root function.</td>
</tr>
<tr>
<td>cupy.cbrt</td>
<td>Elementwise cube root function.</td>
</tr>
<tr>
<td>cupy.square</td>
<td>Elementwise square function.</td>
</tr>
<tr>
<td>cupy.abs</td>
<td>Elementwise absolute value function.</td>
</tr>
<tr>
<td>cupy.sign</td>
<td>Elementwise sign function.</td>
</tr>
<tr>
<td>cupy.maximum</td>
<td>Takes the maximum of two arrays elementwise.</td>
</tr>
<tr>
<td>cupy.minimum</td>
<td>Takes the minimum of two arrays elementwise.</td>
</tr>
<tr>
<td>cupy.fmax</td>
<td>Takes the maximum of two arrays elementwise.</td>
</tr>
<tr>
<td>cupy.fmin</td>
<td>Takes the minimum of two arrays elementwise.</td>
</tr>
<tr>
<td>cupy.nan_to_num</td>
<td>Elementwise nan_to_num function.</td>
</tr>
<tr>
<td>cupy.bartlett</td>
<td>Returns the Bartlett window.</td>
</tr>
<tr>
<td>cupy.blackman</td>
<td>Returns the Blackman window.</td>
</tr>
<tr>
<td>cupy.hamming</td>
<td>Returns the Hamming window.</td>
</tr>
<tr>
<td>cupy.hanning</td>
<td>Returns the Hanning window.</td>
</tr>
<tr>
<td>cupy.kaiser</td>
<td>Return the Kaiser window.</td>
</tr>
</tbody>
</table>

**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_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.

Returns Clipped array.

Return type *cupy.ndarray*

See also:

- `numpy.clip()`

**cupy.cbrt**

cupy.cbrt = <ufunc 'cupy_cbrt'>

Elementwise cube root function.

See also:

- `numpy.cbrt`

**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 Documentation, Release 8.2.0

**cupy.bartlett**

cupy.bartlett(M)

Returns the Bartlett window.

The Bartlett window is defined as

\[
    w(n) = \frac{2}{M-1} \left( M - 1 - \left| n - \frac{M-1}{2} \right| \right)
\]

**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.

\[ w(n) = I_0 \left( \sqrt{1 - \frac{4n^2}{(M-1)^2}} \right) / I_0(\beta) \]

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()`

### 4.3.12 Padding

`cupy.pad`

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**

```python
def example_function(vector, iaxis_pad_width, iaxis, kwargs):
    # Implementation of the padding function

>>> 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])
```

```python
>>> cupy.pad(a, (2, 3), 'edge')
array([1, 1, 1, ..., 5, 5, 5])
```

```python
>>> cupy.pad(a, (2, 3), 'linear_ramp', end_values=(5, -4))
array([ 5, 3, 1, 2, 3, 4, 5, 2, -1, -4])
```

```python
>>> cupy.pad(a, (2, ), 'maximum')
array([5, 5, 1, 2, 3, 4, 5, 5, 5])
```

```python
>>> cupy.pad(a, (2, ), 'mean')
array([3, 3, 1, 2, 3, 4, 5, 3, 3])
```

```python
>>> 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],
       [3, 3, 4, 3, 3, 3],
       [3, 3, 4, 3, 3, 3],
       [3, 3, 4, 3, 3, 3],
       [3, 3, 4, 3, 3, 3],
       [3, 3, 4, 3, 3, 3],
       [3, 3, 4, 3, 3, 3]])
```

(continues on next page)
CuPy Documentation, Release 8.2.0

(continued from previous page)

```python
>>> 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, 6, 7])

>>> cupy.pad(a, (2, 3), 'wrap')
array([4, 5, 1, 2, 3, 4, 5, 1, 2, 3])
```

```python
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]])
```

```python
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]])
```

4.3.13 Polynomials

Polynomial Package

Polynomial Module

- `cupy.polynomial.polynomial.polyvander` Computes the Vandermonde matrix of given degree.
- `cupy.polynomial.polynomial.polycompanion` Computes the companion matrix of `c`.
cupy.polynomial.polynomial.polyvander

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

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**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cupy.polynomial.polyutils.as_series</td>
<td>Returns argument as a list of 1-d arrays.</td>
</tr>
<tr>
<td>cupy.polynomial.polyutils.trimseq</td>
<td>Removes small polynomial series coefficients.</td>
</tr>
<tr>
<td>cupy.polynomial.polyutils.trimcoef</td>
<td>Removes small trailing coefficients from a polynomial.</td>
</tr>
</tbody>
</table>

cupy.polynomial.polyutils.as_series

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

Basics

<table>
<thead>
<tr>
<th>cupy.poly1d</th>
<th>A one-dimensional polynomial class.</th>
</tr>
</thead>
<tbody>
<tr>
<td>cupy.polyval</td>
<td>Evaluates a polynomial at specific values.</td>
</tr>
<tr>
<td>cupy.roots</td>
<td>Computes the roots of a polynomial with given coefficients.</td>
</tr>
</tbody>
</table>

cupy.poly1d

class cupy.poly1d(c_or_r, r=False, variable=None)
A one-dimensional polynomial class.

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

```python
__call__()
Call self as a function.

__getitem__()
Return self[key].

__setitem__()
Set self[key] to value.

__len__()
Return len(self).

__iter__()
Implement iter(self).

def deriv(self, m=1)
def 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

def integ(self, m=1, k=0)
def 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.

### Attributes

c
ccoef
coefficients
coeffs

order

4.3. Routines
```python
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.

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()

Arithmetic

<table>
<thead>
<tr>
<th>cupy.polyadd</th>
<th>Computes the sum of two polynomials.</th>
</tr>
</thead>
<tbody>
<tr>
<td>cupy.polysub</td>
<td>Computes the difference of two polynomials.</td>
</tr>
<tr>
<td>cupy.polymul</td>
<td>Computes the product of two polynomials.</td>
</tr>
</tbody>
</table>
```
cupy.polyadd

```python
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()
Differences between 
\texttt{cupy.random} and \texttt{numpy.random}:

- CuPy provides Legacy Random Generation API (see also: NumPy 1.16 Reference). The new random generator API (\texttt{numpy.random.Generator} class) introduced in NumPy 1.17 has not been implemented yet.

- Most functions under \texttt{cupy.random} support the \texttt{dtype} option, which do not exist in the corresponding NumPy APIs. This option enables generation of float32 values directly without any space overhead.

- CuPy does not guarantee that the same number generator is used across major versions. This means that numbers generated by \texttt{cupy.random} by new major version may not be the same as the previous one, even if the same seed and distribution are used.

### Simple random data

<table>
<thead>
<tr>
<th>\texttt{cupy.random.rand}</th>
<th>Returns an array of uniform random values over the interval $[0, 1)$.</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{cupy.random.randn}</td>
<td>Returns an array of standard normal random values.</td>
</tr>
<tr>
<td>\texttt{cupy.random.randint}</td>
<td>Returns a scalar or an array of integer values over $[\text{low}, \text{high})$.</td>
</tr>
<tr>
<td>\texttt{cupy.random.random_integers}</td>
<td>Return a scalar or an array of integer values over $[\text{low}, \text{high}]$.</td>
</tr>
<tr>
<td>\texttt{cupy.random.random_sample}</td>
<td>Returns an array of random values over the interval $[0, 1)$.</td>
</tr>
<tr>
<td>\texttt{cupy.random.random}</td>
<td>Returns an array of random values over the interval $[0, 1)$.</td>
</tr>
<tr>
<td>\texttt{cupy.random.ranf}</td>
<td>Returns an array of random values over the interval $[0, 1)$.</td>
</tr>
<tr>
<td>\texttt{cupy.random.sample}</td>
<td>Returns an array of random values over the interval $[0, 1)$.</td>
</tr>
<tr>
<td>\texttt{cupy.random.choice}</td>
<td>Returns an array of random values from a given 1-D array.</td>
</tr>
<tr>
<td>\texttt{cupy.random.bytes}</td>
<td>Returns random bytes.</td>
</tr>
</tbody>
</table>

\texttt{cupy.random.rand}\

\texttt{cupy.random.rand(*size, **kwargs)}

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**

- \texttt{size} (ints) – The shape of the array.

- \texttt{dtype} – Data type specifier. Only \texttt{numpy.float32} and \texttt{numpy.float64} types are allowed. The default is \texttt{numpy.float64}.

**Returns** A random array.

**Return type** \texttt{cupy.ndarray}

**See also:**

\texttt{numpy.random.rand}
Example

```python
>>> cupy.random.rand(3, 2)
array([[0.86476479, 0.05633727],
       [0.27283185, 0.38255354],
       [0.16592278, 0.75150313]])
# random

>>> cupy.random.rand(3, 2, dtype=cupy.float32)
array([[0.9672306 , 0.9590486 ],
       [0.6851264 , 0.70457625],
       [0.22382522, 0.36055237]], dtype=float32)
# random
```

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

```python
>>> cupy.random.randn(3, 2)
array([[0.41193321, 1.59579542],
       [0.47904589, 0.18566376],
       [0.59748424, 2.32602829]])
# random

>>> cupy.random.randn(3, 2, dtype=cupy.float32)
array([[ 0.1373886 , 2.403238 ],
       [ 0.84020025, 1.5089266 ],
       [-1.2268474 , -0.48219103]], 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 \([low, high)\).

4.3. Routines
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 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.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.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.ranf

`cupy.random.ranf(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.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`
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.bytes

cupy.random.bytes(length)

Returns random bytes.

See also:

- `numpy.random.bytes`

---

Permutations

- **cupy.random.shuffle**
  Shuffles an array.
- **cupy.random.permutation**
  Returns a permuted range or a permutation of an 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.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

Distributions

cupy.random.beta
Beta distribution.
cupy.random.binomial
Binomial distribution.
cupy.random.chisquare
Chi-square distribution.
cupy.random.dirichlet
Dirichlet distribution.
cupy.random.exponential
Exponential distribution.
cupy.random.f
F distribution.
cupy.random.gamma
Gamma distribution.
cupy.random.geometric
Geometric distribution.
cupy.random.gumbel
Returns an array of samples drawn from a Gumbel distribution.
cupy.random.hypergeometric
hypergeometric distribution.
cupy.random.laplace
Laplace distribution.
cupy.random.logistic
Logistic distribution.
cupy.random.lognormal
Returns an array of samples drawn from a log normal distribution.
cupy.random.logseries
Log series distribution.
cupy.random.multinomial
Returns an array from multinomial distribution.
cupy.random.multivariate_normal
Multivariate normal distribution.
cupy.random.negative_binomial
Negative binomial distribution.
cupy.random.noncentral_chisquare
Noncentral chi-square distribution.
cupy.random.noncentral_f
Noncentral F distribution.
cupy.random.normal
Returns an array of normally distributed samples.
cupy.random.pareto
Pareto II or Lomax distribution.
cupy.random.poisson
Poisson distribution.
cupy.random.power
Power distribution.
cupy.random.rayleigh
Rayleigh distribution.
cupy.random.standard_cauchy
Standard cauchy distribution.
cupy.random.standard_exponential
Standard exponential distribution.
cupy.random.standard_gamma
Standard gamma distribution.
cupy.random.standard_normal
Returns an array of samples drawn from the standard normal distribution.
CuPy Documentation, Release 8.2.0

Table 65 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cupy.random.standard_t</code></td>
<td>Standard Student’s t distribution.</td>
</tr>
<tr>
<td><code>cupy.random.triangular</code></td>
<td>Triangular distribution.</td>
</tr>
<tr>
<td><code>cupy.random.uniform</code></td>
<td>Returns an array of uniformly-distributed samples over an interval.</td>
</tr>
<tr>
<td><code>cupy.random.vonmises</code></td>
<td>von Mises distribution.</td>
</tr>
<tr>
<td><code>cupy.random.wald</code></td>
<td>Wald distribution.</td>
</tr>
<tr>
<td><code>cupy.random.weibull</code></td>
<td>weibull distribution.</td>
</tr>
<tr>
<td><code>cupy.random.zipf</code></td>
<td>Zipf distribution.</td>
</tr>
</tbody>
</table>

`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

\[ f(x) = \frac{x^{a-1}(1-x)^{b-1}}{B(a, b)} \]

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  \texttt{cupy.ndarray}

See also:
\texttt{numpy.random.binomial}

\texttt{cupy.random.chisquare}

cupy.random.chisquare(\texttt{df}, \texttt{size=\texttt{None}}, \texttt{dtype=\texttt{<class \textquotesingle float\textquotesingle>}})

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

\begin{itemize}
\item \texttt{df} (\texttt{int} or array_like of \texttt{ints}) – Degree of freedom \( k \).
\item \texttt{size} (\texttt{int} or tuple of \texttt{ints}) – The shape of the array. If \texttt{None}, a zero-dimensional array is generated.
\item \texttt{dtype} – Data type specifier. Only \texttt{numpy.float32} and \texttt{numpy.float64} types are allowed.
\end{itemize}

Returns  Samples drawn from the chi-square distribution.

Return type  \texttt{cupy.ndarray}

See also:
\texttt{numpy.random.chisquare}

\texttt{cupy.random.dirichlet}

cupy.random.dirichlet(\texttt{alpha}, \texttt{size=\texttt{None}}, \texttt{dtype=\texttt{<class \textquotesingle float\textquotesingle>}})

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

\begin{itemize}
\item \texttt{alpha} (array) – Parameters of the dirichlet distribution \( \alpha \).
\item \texttt{size} (\texttt{int} or tuple of \texttt{ints}) – The shape of the array. If \texttt{None}, a zero-dimensional array is generated.
\item \texttt{dtype} – Data type specifier. Only \texttt{numpy.float32} and \texttt{numpy.float64} types are allowed.
\end{itemize}

Returns  Samples drawn from the dirichlet distribution.

Return type  \texttt{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=1.0, 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=1.0, size=None, dtype=<class 'int'>)

Geometric distribution. Returns an array of samples drawn from the geometric distribution. Its probability mass function is defined as

\[ f(x) = p(1 - p)^{k-1}. \]

**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:

- cupy.random.RandomState.geometric() 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 $\text{loc}$ and scale $\text{scale}$. 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 $\text{loc}$ and $\eta$ is $\text{scale}$.

**Parameters**

- $\text{loc}$ ($\text{float}$) – The location of the mode $\mu$.
- $\text{scale}$ ($\text{float}$) – The scale parameter $\eta$.
- $\text{size}$ ($\text{int or tuple of ints}$) – The shape of the array. If None, a zero-dimensional array is generated.
- $\text{dtype}$ – 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 ($\text{ngood}$, $\text{nbad}$, $\text{nsample}$, $\text{size}$=None, $\text{dtype}$=class 'int'>)

Returns an array of samples drawn from the hypergeometric distribution. Its probability mass function is defined as

$$f(x) = \binom{n}{x} \binom{N-n}{m-x} \binom{N}{n}^{-1}.$$

**Parameters**

- $\text{ngood}$ ($\text{int or array_like of ints}$) – Parameter of the hypergeometric distribution $n$.
- $\text{nbad}$ ($\text{int or array_like of ints}$) – Parameter of the hypergeometric distribution $m$.
- $\text{nsample}$ ($\text{int or array_like of ints}$) – Parameter of the hypergeometric distribution $N$.
- $\text{size}$ ($\text{int or tuple of ints}$) – The shape of the array. If None, a zero-dimensional array is generated.
- $\text{dtype}$ – 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 \( p = \text{len}(\text{pvals}) \). 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 \( \text{sum}(\text{pvals}) < 1 \) case.

**See also:**

* numpy.random.multinomial

**cupy.random.multivariate_normal**

```python
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

\[
f(x) = \frac{1}{(2\pi)^{n/2} |\Sigma|} \exp\left(-\frac{1}{2} (x - \mu)^T \Sigma^{-1} (x - \mu)\right).
\]

**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 \( A \cdot A^T = \text{cov} \). 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  cupy.ndarray

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

\[
f(x) = \frac{1}{2} e^{-\frac{x + \lambda}{2}} \left( \frac{x}{\lambda} \right)^{k/4 - 1/2} I_{k/2 - 1}(\sqrt{\lambda x}),
\]

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.


**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.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

\[
f(x) = \frac{\lambda^x e^{-\lambda}}{k!}.
\]

**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.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.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

\[ f(x) = \frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\sqrt{\nu\pi} \Gamma\left(\frac{\nu}{2}\right)} \left(1 + \frac{x^2}{\nu}\right)^{-\left(\frac{\nu+1}{2}\right)}. \]
Parameters

• \textbf{df} (float or array_like of floats) – Degree of freedom \( \nu \).
• \textbf{size} (int or tuple of ints) – The shape of the array. If None, a zero-dimensional array is generated.
• \textbf{dtype} – Data type specifier. Only \texttt{numpy.float32} and \texttt{numpy.float64} types are allowed.

Returns Samples drawn from the standard Student’s t distribution.

Return type \texttt{cupy.ndarray}

See also: \texttt{numpy.random.standard_t}

cupy.random.triangular

cupy.random.triangular\(\text{(left}, \text{mode}, \text{right}, \text{size=None}, \text{dtype=<class} \text{'}float\text{'}))\)

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

• \textbf{left} (float) – Lower limit \( l \).
• \textbf{mode} (float) – The value where the peak of the distribution occurs. \( m \).
• \textbf{right} (float) – Higher Limit \( r \).
• \textbf{size} (int or tuple of ints) – The shape of the array. If None, a zero-dimensional array is generated.
• \textbf{dtype} – Data type specifier. Only \texttt{numpy.float32} and \texttt{numpy.float64} types are allowed.

Returns Samples drawn from the triangular distribution.

Return type \texttt{cupy.ndarray}

See also: \texttt{cupy.random.RandomState.triangular()} \texttt{numpy.random.triangular}

cupy.random.uniform

cupy.random.uniform\(\text{(low=0.0, high=1.0, size=None, dtype=<class} \text{'}float\text{'}))\)

Returns an array of uniformly-distributed samples over an interval.

Samples are drawn from a uniform distribution over the half-open interval \([low, high)\).

Parameters

• \textbf{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

\[ f(x) = \sqrt{\frac{\lambda}{2\pi x^3}} e^{-\frac{\lambda(x-\mu)^2}{2x^2}} \]

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:**

`cupy.random.RandomState.wald()`  
`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

Random generator

<table>
<thead>
<tr>
<th>cupy.random.RandomState</th>
<th>Portable container of a pseudo-random number generator.</th>
</tr>
</thead>
<tbody>
<tr>
<td>cupy.random.seed</td>
<td>Resets the state of the random number generator with a seed.</td>
</tr>
<tr>
<td>cupy.random.get_random_state</td>
<td>Gets the state of the random number generator for the current device.</td>
</tr>
<tr>
<td>cupy.random.set_random_state</td>
<td>Sets the state of the random number generator for the current device.</td>
</tr>
</tbody>
</table>

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_XORWOW
  - cupy.cuda.curand.CURAND_RNG_MRG32K3A
  - cupy.cuda.curand.CURAND_RNG_MTGP32
  - cupy.cuda.curand.CURAND_RNG_MT19937
  - cupy.cuda.curand.CURAND_RNG_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 document, 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 Documentation, Release 8.2.0

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.

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

See also:

cupy.random.negative_binomial() for full documentation, numpy.random.

`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.

`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.

`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_kernel() for full documentation, numpy.random.RandomState.

`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='l')
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.

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.

Warning: 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.

Warning: This function may synchronize the device.

See also:
cupy.random.zipf() for full documentation, numpy.random.RandomState.zipf

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.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.

Note: CuPy does not provide cupy.random.get_state nor cupy.random.set_state at this time. Use cupy.random.get_random_state() and cupy.random.set_random_state() instead. Note that these functions use cupy.random.RandomState instance to represent the internal state, which cannot be serialized.

4.3.15 Sorting, Searching, and Counting

Sorting

cupy.sort
Returns a sorted copy of an array with a stable sorting algorithm.

cupy.lexsort
Perform an indirect sort using an array of keys.

cupy.argsort
Returns the indices that would sort an array with a stable sorting.

cupy.msort
Returns a copy of an array sorted along the first axis.

cupy.sort_complex
Sort a complex array using the real part first, then the imaginary part.

cupy.partition
Returns a partitioned copy of an array.

cupy.argpartition
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)`

Performs 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 the `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 the `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.argpartition**

`cupy.argpartition(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.argpartition` fully sorts the given array as `cupy.argsort` does. It also does not support `kind` and `order` parameters that `numpy.argpartition` supports.

See also:
- `numpy.argpartition()`

See also:
- `cupy.ndarray.sort()`

### Searching

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cupy.argmax</code></td>
<td>Returns the indices of the maximum along an axis.</td>
</tr>
<tr>
<td><code>cupy.nanargmax</code></td>
<td>Return the indices of the maximum values in the specified axis ignoring NaNs.</td>
</tr>
<tr>
<td><code>cupy.argmin</code></td>
<td>Returns the indices of the minimum along an axis.</td>
</tr>
<tr>
<td><code>cupy.nanargmin</code></td>
<td>Return the indices of the minimum values in the specified axis ignoring NaNs.</td>
</tr>
<tr>
<td><code>cupy.nonzero</code></td>
<td>Return the indices of the elements that are non-zero.</td>
</tr>
<tr>
<td><code>cupy.flatnonzero</code></td>
<td>Return indices that are non-zero in the flattened version of <code>a</code>.</td>
</tr>
<tr>
<td><code>cupy.where</code></td>
<td>Return elements, either from <code>x</code> or <code>y</code>, depending on condition.</td>
</tr>
<tr>
<td><code>cupy.argwhere</code></td>
<td>Return the indices of the elements that are non-zero.</td>
</tr>
<tr>
<td><code>cupy.searchsorted</code></td>
<td>Finds indices where elements should be inserted to maintain order.</td>
</tr>
</tbody>
</table>

#### `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.argmin

cupy.argmin(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.argmin()
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.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.argwhere

`cupy.argwhere(a)`

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

Returns a (N, ndim) dimantional 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.argwhere()

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 \( a \).
- \( \text{side} \) – {‘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 \( a \).
- \( \text{sorter} \) – 1-D array_like Optional array of integer indices that sort array \( a \) into ascending order. They are typically the result of \( \text{argsort()} \).

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()

Counting
cupy.count_nonzero

cupy.count_nonzero(a, axis=None)

Counts the number of non-zero values in the array.

Note: \( \text{numpy.count_nonzero()} \) returns \( \text{int} \) value when \( \text{axis}=\text{None} \), but \( \text{cupy.count_nonzero()} \) returns zero-dimensional array to reduce CPU-GPU synchronization.

Parameters

- \( a (cupy.ndarray) \) – The array for which to count non-zeros.
- \( \text{axis} (\text{int or tuple, optional}) \) – 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

### 4.3.16 Statistical Functions

**Order statistics**

<table>
<thead>
<tr>
<th>cupy.amin</th>
<th>Returns the minimum of an array or the minimum along an axis.</th>
</tr>
</thead>
<tbody>
<tr>
<td>cupy.amax</td>
<td>Returns the maximum of an array or the maximum along an axis.</td>
</tr>
<tr>
<td>cupy.nanmin</td>
<td>Returns the minimum of an array along an axis ignoring NaN.</td>
</tr>
<tr>
<td>cupy.nanmax</td>
<td>Returns the maximum of an array along an axis ignoring NaN.</td>
</tr>
<tr>
<td>cupy.percentile</td>
<td>Computes the q-th percentile of the data along the specified axis.</td>
</tr>
</tbody>
</table>

**cupy.amin**

```python
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

```python
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**

```python
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 Documentation, Release 8.2.0**

**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.
- `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.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.
- `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()`
Means and variances

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cupy.average</code></td>
<td>Returns the weighted average along an axis.</td>
</tr>
<tr>
<td><code>cupy.mean</code></td>
<td>Returns the arithmetic mean along an axis.</td>
</tr>
<tr>
<td><code>cupy.var</code></td>
<td>Returns the variance along an axis.</td>
</tr>
<tr>
<td><code>cupy.std</code></td>
<td>Returns the standard deviation along an axis.</td>
</tr>
<tr>
<td><code>cupy.nanmean</code></td>
<td>Returns the arithmetic mean along an axis ignoring NaN</td>
</tr>
<tr>
<td><code>cupy.nanvar</code></td>
<td>Returns the variance along an axis ignoring NaN values.</td>
</tr>
<tr>
<td><code>cupy.nanstd</code></td>
<td>Returns the standard deviation along an axis ignoring NaN</td>
</tr>
</tbody>
</table>

**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)* - 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.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.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.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)` – 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.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()`

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()`

---

**Histograms**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cupy.histogram</code></td>
<td>Computes the histogram of a set of data.</td>
</tr>
<tr>
<td><code>cupy.bincount</code></td>
<td>Count number of occurrences of each value in array of non-negative ints.</td>
</tr>
</tbody>
</table>

**`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.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 \( \max(cupy.\text{max}(x) + 1, \text{minlength}) \).

Return type cupy.ndarray

Warning: This function may synchronize the device.

See also:

numpy.bincount()

Correlations

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.cov
cupy.cov\(a, y=\text{None}, \text{rowvar}=\text{True}, \text{bias}=\text{False}, ddof=\text{None}\)
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()

cupy.correlate
cupy.correlate\(a, v, \text{mode}='\text{valid}'\)
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()

4.3.17 CuPy-specific Functions
CuPy-specific functions are placed under cupyx namespace.

<table>
<thead>
<tr>
<th>cupyx.rsqrt</th>
<th>Returns the reciprocal square root.</th>
</tr>
</thead>
<tbody>
<tr>
<td>cupyx.scatter_add</td>
<td>Adds given values to specified elements of an array.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 74 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cupyx.scatter_max</code></td>
<td>Stores a maximum value of elements specified by indices to an array.</td>
</tr>
<tr>
<td><code>cupyx.scatter_min</code></td>
<td>Stores a minimum value of elements specified by indices to an array.</td>
</tr>
</tbody>
</table>

**cupyx.rsqrt**

`cupyx.rsqrt = <ufunc 'cupy_rsqt'>`

Returns the reciprocal square root.

**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**

```python
>>> 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.

Note: As of v4, this function is moved from `cupy` package to `cupyx` package. `cupy.scatter_add` is still available for backward compatibility.

See also:

`numpy.ufunc.at()`.

**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**

```python
>>> 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.

**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

```python
>>> 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.

4.3.18 CUB/cuTENSOR backend for reduction 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 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. We recommend users to perform some benchmarks to determine whether CUB/cuTENSOR offers better performance or not.

4.4 SciPy-compatible Routines

The following pages describe SciPy-compatible routines. These functions cover a subset of SciPy routines.

4.4.1 Discrete Fourier transforms (scipy.fft)

Fast Fourier Transforms

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cupyx.scipy.fft.fft</td>
<td>Compute the one-dimensional FFT.</td>
</tr>
<tr>
<td>cupyx.scipy.fft.ifft</td>
<td>Compute the one-dimensional inverse FFT.</td>
</tr>
<tr>
<td>cupyx.scipy.fft.fft2</td>
<td>Compute the two-dimensional FFT.</td>
</tr>
<tr>
<td>cupyx.scipy.fft.ifft2</td>
<td>Compute the two-dimensional inverse FFT.</td>
</tr>
<tr>
<td>cupyx.scipy.fft.fftn</td>
<td>Compute the N-dimensional FFT.</td>
</tr>
<tr>
<td>cupyx.scipy.fft.ifftn</td>
<td>Compute the N-dimensional inverse FFT.</td>
</tr>
</tbody>
</table>
### CuPy Documentation, Release 8.2.0

Table 75 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cupyx.scipy.fft.rfft</code></td>
<td>Compute the one-dimensional FFT for real input.</td>
</tr>
<tr>
<td><code>cupyx.scipy.fft.irfft</code></td>
<td>Compute the one-dimensional inverse FFT for real input.</td>
</tr>
<tr>
<td><code>cupyx.scipy.fft.rfft2</code></td>
<td>Compute the two-dimensional FFT for real input.</td>
</tr>
<tr>
<td><code>cupyx.scipy.fft.irfft2</code></td>
<td>Compute the two-dimensional inverse FFT for real input.</td>
</tr>
<tr>
<td><code>cupyx.scipy.fft.rfftn</code></td>
<td>Compute the N-dimensional FFT for real input.</td>
</tr>
<tr>
<td><code>cupyx.scipy.fft.irfftn</code></td>
<td>Compute the N-dimensional inverse FFT for real input.</td>
</tr>
<tr>
<td><code>cupyx.scipy.fft.hfft</code></td>
<td>Compute the FFT of a signal that has Hermitian symmetry.</td>
</tr>
<tr>
<td><code>cupyx.scipy.fft.ihfft</code></td>
<td>Compute the FFT of a signal that has Hermitian symmetry.</td>
</tr>
</tbody>
</table>

#### `cupyx.scipy.fft.fft`

```python
cupy.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 (None or 'ortho')` – Normalization mode.
- `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:

  ```python
  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.ifft`

```python
cupy.scipy.fft.ifft(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** (None or 'ortho') – Normalization mode.
• **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:

```python
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.ifft()`

**cupyx.scipy.fft.fft2**

cupy.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.
• **$norm$** (None or 'ortho') – Normalization mode.
• **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:

```python
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**

cupy.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 (None or 'ortho')` – Normalization mode.
- `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:

```python
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.idftn()`

cupy.scipy.fft.fftn

cupy.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 (None or 'ortho')` – Normalization mode.
- `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:

```python
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.fftn()`
cupyx.scipy.fft.ifftn

**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` (None or 'ortho') – Normalization mode.
- `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:

```python
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` (None or 'ortho') – Normalization mode.
- `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:

```python
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**

cupy.scipy.fft.\texttt{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` (*None or 'ortho'*) – Normalization mode.
- `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:

\[
\text{plan} = \text{cupyx.scipy.fftpack.get_fft_plan}(x, n, axis, \text{value_type}='C2R')
\]

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**

cupy.scipy.fft.\texttt{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` (*None or "ortho")* – Keyword to specify the normalization mode.
- `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:

```python
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()`

cupy.scipy.fft.irfft2

cupy.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** (`None` or "ortho") – Keyword to specify the normalization mode.
- **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:

```python
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()`

cupy.scipy.fft.rfftn

cupy.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 (None or "ortho")` – Keyword to specify the normalization mode.
• `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:

```python
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()

cupy.scipy.fft.irfftn
cupy.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 (None or "ortho")` – Keyword to specify the normalization mode.
• `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:

```python
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.irfftn()`

cupy.scipy.fft.hfft

cupy.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 (None or "ortho")` – Keyword to specify the normalization mode.
- `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: `cupy.ndarray`

`scipy.fft.hfft()`

cupy.scipy.fft.ihfft

cupy.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 (None or "ortho")` – Keyword to specify the normalization mode.
- `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: `cupy.ndarray`

`scipy.fft.ihfft()`
Helper functions for FFT

`cupyx.scipy.fft.next_fast_len`  
Find the next fast size to `fft`.

**`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 `FFT Functions`. 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.

5. The boolean switch `cupy.fft.config.use_multi_gpus` also affects the FFT functions in this module, see `FFT Functions`. Moreover, this switch is honored when planning manually using `get_fft_plan()`.

**Note:** `scipy.fft` requires SciPy version 1.4.0 or newer.

**Note:** To use `scipy.fft.set_backend()` together with an explicit `plan` argument requires SciPy version 1.5.0 or newer.
4.4.2 Legacy Discrete Fourier transforms (scipy.fftpack)

Note: As of SciPy version 1.4.0, scipy.fft is recommended over scipy.fftpack. Consider using cupyx.scipy.fft instead.

Fast Fourier Transforms

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cupyx.scipy.fftpack.fft</td>
<td>Compute the one-dimensional FFT.</td>
</tr>
<tr>
<td>cupyx.scipy.fftpack.ifft</td>
<td>Compute the one-dimensional inverse FFT.</td>
</tr>
<tr>
<td>cupyx.scipy.fftpack.fft2</td>
<td>Compute the two-dimensional FFT.</td>
</tr>
<tr>
<td>cupyx.scipy.fftpack.ifft2</td>
<td>Compute the two-dimensional inverse FFT.</td>
</tr>
<tr>
<td>cupyx.scipy.fftpack.fftn</td>
<td>Compute the N-dimensional FFT.</td>
</tr>
<tr>
<td>cupyx.scipy.fftpack.ifftn</td>
<td>Compute the N-dimensional inverse FFT.</td>
</tr>
<tr>
<td>cupyx.scipy.fftpack.rfft</td>
<td>Compute the one-dimensional FFT for real input.</td>
</tr>
<tr>
<td>cupyx.scipy.fftpack.irfft</td>
<td>Compute the one-dimensional inverse FFT for real input.</td>
</tr>
<tr>
<td>cupyx.scipy.fftpack.get_fft_plan</td>
<td>Generate a CUDA FFT plan for transforming up to three axes.</td>
</tr>
</tbody>
</table>

**cupyx.scipy.fftpack.fft**

cupy.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:

```python
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:

```python
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:

```python
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` is True, or use no cuFFT plan if it is set to False.

### 4.4. SciPy-compatible Routines
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.ifft2
cupyx.scipy.fftpack.ifft2(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.ifft2()

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.
CuPy Documentation, Release 8.2.0

- **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:

  ```python
  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**

*cupy.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:

  ```python
  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.

**cumpy.scipy.fftpack.rfft**

cumpy.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:

\[
\begin{align*}
[y(0), \text{Re}(y(1)), \text{Im}(y(1)), \ldots, \text{Re}(y(n/2))] & \quad \# \text{if } n \text{ is even} \\
[y(0), \text{Re}(y(1)), \text{Im}(y(1)), \ldots, \text{Re}(y(n/2)), \text{Im}(y(n/2))] & \quad \# \text{if } n \text{ is odd}
\end{align*}
\]

**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:

\[
\text{plan} = \text{cupyx.scipy.fftpack.get_fft_plan}(\text{x, axes, value_type='R2C'})
\]

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.

**cumpy.scipy.fftpack.irfft**

cumpy.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 `cupyx.scipy.fft.irfft`.

---

**cupyx.scipy.fftpack.get_fft_plan**

cupy.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:

```python
x = cupy.random.random(16).reshape(4, 4).astype(cupy.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.

**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 `FFT Functions`. 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 overwitten 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 `FFT Functions`. Moreover, this switch is honored when planning manually using `get_fft_plan()`.

### 4.4.3 Linear Algebra

**Basics**

- `cupyx.scipy.linalg.solve_triangular`: Solve the equation `a x = b` for `x`, assuming `a` is a triangular matrix.

**Decompositions**

- `cupyx.scipy.linalg.lu_factor`: LU decomposition.
- `cupyx.scipy.linalg.lu_solve`: Solve an equation system, `a * x = b`, given the LU factorization of `a`.

**Special Matrices**

- `cupyx.scipy.linalg.block_diag`: Create a block diagonal matrix from provided arrays.
- `cupyx.scipy.linalg.circulant`: Construct a circulant matrix.
- `cupyx.scipy.linalg.companion`: Create a companion matrix.
- `cupyx.scipy.linalg.convolution_matrix`: Construct a convolution matrix.
- `cupyx.scipy.linalg.dft`: Discrete Fourier transform matrix.

Continued on next page
Table 80 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cupyx.scipy.linalg.fiedler</code></td>
<td>Returns a symmetric Fiedler matrix</td>
</tr>
<tr>
<td><code>cupyx.scipy.linalg.fiedler_companion</code></td>
<td>Returns a Fiedler companion matrix</td>
</tr>
<tr>
<td><code>cupyx.scipy.linalg.hadamard</code></td>
<td>Construct an Hadamard matrix.</td>
</tr>
<tr>
<td><code>cupyx.scipy.linalg.hankel</code></td>
<td>Construct a Hankel matrix.</td>
</tr>
<tr>
<td><code>cupyx.scipy.linalg.helmert</code></td>
<td>Create an Helmert matrix of order n.</td>
</tr>
<tr>
<td><code>cupyx.scipy.linalg.hilbert</code></td>
<td>Create a Hilbert matrix of order n.</td>
</tr>
<tr>
<td><code>cupyx.scipy.linalg.kron</code></td>
<td>Kronecker product.</td>
</tr>
<tr>
<td><code>cupyx.scipy.linalg.leslie</code></td>
<td>Create a Leslie matrix.</td>
</tr>
<tr>
<td><code>cupyx.scipy.linalg.toeplitz</code></td>
<td>Construct a Toeplitz matrix.</td>
</tr>
<tr>
<td><code>cupyx.scipy.linalg.tri</code></td>
<td>Construct ((N, M)) matrix filled with ones at and below the (k)-th diagonal.</td>
</tr>
<tr>
<td><code>cupyx.scipy.linalg.tril</code></td>
<td>Make a copy of a matrix with elements above the (k)-th diagonal zeroed.</td>
</tr>
<tr>
<td><code>cupyx.scipy.linalg.triu</code></td>
<td>Make a copy of a matrix with elements below the (k)-th diagonal zeroed.</td>
</tr>
</tbody>
</table>

4.4.4 Multi-dimensional image processing

CuPy provides multi-dimensional image processing functions. It supports a subset of \texttt{scipy.ndimage} interface.

**Filters**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cupyx.scipy.ndimage.convolve</code></td>
<td>Multi-dimensional convolution.</td>
</tr>
<tr>
<td><code>cupyx.scipy.ndimage.convolve1d</code></td>
<td>One-dimensional convolution.</td>
</tr>
<tr>
<td><code>cupyx.scipy.ndimage.correlate</code></td>
<td>Multi-dimensional correlate.</td>
</tr>
<tr>
<td><code>cupyx.scipy.ndimage.correlate1d</code></td>
<td>One-dimensional correlate.</td>
</tr>
<tr>
<td><code>cupyx.scipy.ndimage.gaussian_filter</code></td>
<td>Multi-dimensional Gaussian filter.</td>
</tr>
<tr>
<td><code>cupyx.scipy.ndimage.gaussian_filter1d</code></td>
<td>One-dimensional Gaussian filter along the given axis.</td>
</tr>
<tr>
<td><code>cupyx.scipy.ndimage.gaussian_gradient_magnitude</code></td>
<td>Multi-dimensional gradient magnitude using Gaussian derivatives.</td>
</tr>
<tr>
<td><code>cupyx.scipy.ndimage.gaussian_laplace</code></td>
<td>Multi-dimensional Laplace filter using Gaussian second derivatives.</td>
</tr>
<tr>
<td><code>cupyx.scipy.ndimage.generic_filter</code></td>
<td>Compute a multi-dimensional filter using the provided raw kernel or reduction kernel.</td>
</tr>
<tr>
<td><code>cupyx.scipy.ndimage.generic_filter1d</code></td>
<td>Compute a 1D filter along the given axis using the provided raw kernel.</td>
</tr>
<tr>
<td><code>cupyx.scipy.ndimage.generic_gradient_magnitude</code></td>
<td>Multi-dimensional gradient magnitude filter using a provided derivative function.</td>
</tr>
<tr>
<td><code>cupyx.scipy.ndimage.generic_laplace</code></td>
<td>Multi-dimensional Laplace filter using a provided second derivative function.</td>
</tr>
<tr>
<td><code>cupyx.scipy.ndimage.laplace</code></td>
<td>Multi-dimensional Laplace filter based on approximate second derivatives.</td>
</tr>
<tr>
<td><code>cupyx.scipy.ndimage.maximum_filter</code></td>
<td>Multi-dimensional maximum filter.</td>
</tr>
<tr>
<td><code>cupyx.scipy.ndimage.maximum_filter1d</code></td>
<td>Compute the maximum filter along a single axis.</td>
</tr>
<tr>
<td><code>cupyx.scipy.ndimage.median_filter</code></td>
<td>Multi-dimensional median filter.</td>
</tr>
<tr>
<td><code>cupyx.scipy.ndimage.minimum_filter</code></td>
<td>Multi-dimensional minimum filter.</td>
</tr>
<tr>
<td><code>cupyx.scipy.ndimage.minimum_filter1d</code></td>
<td>Compute the minimum filter along a single axis.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 81 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cupyx.scipy.ndimage.percentile_filter</code></td>
<td>Multi-dimensional percentile filter.</td>
</tr>
<tr>
<td><code>cupyx.scipy.ndimage.prewitt</code></td>
<td>Compute a Prewitt filter along the given axis.</td>
</tr>
<tr>
<td><code>cupyx.scipy.ndimage.rank_filter</code></td>
<td>Multi-dimensional rank filter.</td>
</tr>
<tr>
<td><code>cupyx.scipy.ndimage.sobel</code></td>
<td>Compute a Sobel filter along the given axis.</td>
</tr>
<tr>
<td><code>cupyx.scipy.ndimage.uniform_filter</code></td>
<td>Multi-dimensional uniform filter.</td>
</tr>
<tr>
<td><code>cupyx.scipy.ndimage.uniform_filter1d</code></td>
<td>One-dimensional uniform filter along the given axis.</td>
</tr>
</tbody>
</table>

**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 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**


- **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.
cuPy Documentation, Release 8.2.0

**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**

cupy.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.

---

cupy.scipy.ndimage.gaussian_filter1d

cupy.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 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 the 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 the 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.

cupy.scipy.ndimage.generic_filter

cupy.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 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:

```cpp
__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**

- **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 the 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/blockId/threadId`:

```c
__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:

```python
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 `derivative2`.

Returns

The result of the filtering.

Return type  *cupy.ndarray*

See also:  *cupy.ndarray*  

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.
CuPy Documentation, Release 8.2.0

- **derivative2** *(callable)* – Function or other callable with the following signature that is called once per axis:

```python
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 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:**  `cupy.ndarray`

```python
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 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:**  `cupy.ndarray`

```python
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 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 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

- **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.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

- **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*
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:  
cupy.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".
The documentation of CuPy is shown in the image. It explains the `rank_filter` function, which is a multi-dimensional rank filter. The function takes several parameters:

- `input`: The input array.
- `rank`: The rank of the element to get. Can be negative to count from the largest value, e.g., `-1` indicates the largest value.
- `size` or `footprint`: One of these must be provided. `size` is ignored if `footprint` is given. Otherwise, `footprint = cupy.ones(size)` with `size` automatically made to match the number of dimensions in `input`.
- `footprint`: A boolean array which specifies which of the elements within this shape will get passed to the filter function.
- `output`: The array in which to place the output. Default is same dtype as the input.
- `mode`: The array borders are handled according to the given mode ("reflect", "constant", "nearest", "mirror", "wrap"). Default is "reflect".
- `cval`: Value to fill past edges of input if mode is "constant". Default is 0.0.
- `origin`: 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`.

The function returns the result of the filtering, which is a `cupy.ndarray`. It also notes that 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.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: *cupy.ndarray*

*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**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cupyx.scipy.ndimage.fourier_gaussian</td>
<td>Multidimensional Gaussian shift filter.</td>
</tr>
<tr>
<td>cupyx.scipy.ndimage.fourier_shift</td>
<td>Multidimensional Fourier shift filter.</td>
</tr>
<tr>
<td>cupyx.scipy.ndimage.fourier_uniform</td>
<td>Multidimensional uniform shift filter.</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cupyx.scipy.ndimage.affine_transform</td>
<td>Apply an affine transformation.</td>
</tr>
<tr>
<td>cupyx.scipy.ndimage.map_coordinates</td>
<td>Map the input array to new coordinates by interpolation.</td>
</tr>
<tr>
<td>cupyx.scipy.ndimage.rotate</td>
<td>Rotate an array.</td>
</tr>
<tr>
<td>cupyx.scipy.ndimage.shift</td>
<td>Shift an array.</td>
</tr>
<tr>
<td>cupyx.scipy.ndimage.zoom</td>
<td>Zoom an array.</td>
</tr>
</tbody>
</table>

**cupyx.scipy.ndimage.affine_transform**

cupyx.scipy.ndimage.affine_transform(input, matrix, offset=0.0, output_shape=None, output=None, order=None, 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, \ldots, 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. If it is not given, order 1 is used. It is different from *scipy.ndimage* and can change in the future. Currently it supports only order 0 and 1.

• **mode** *(str)* – Points outside the boundaries of the input are filled according to the given mode ('constant', 'nearest', 'mirror' or 'opencv'). Default is 'constant'.

• **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=None*, *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. If it is not given, order 1 is used. It is different from *scipy.ndimage* and can change in the future. Currently it supports only order 0 and 1.

• **mode** *(str)* – Points outside the boundaries of the input are filled according to the given mode ('constant', 'nearest', 'mirror' or 'opencv'). Default is 'constant'.

• **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()`

```python
cupyx.scipy.ndimage.rotate
```

`cupyx.scipy.ndimage.rotate(input, angle, axes=(1, 0), reshape=True, output=None, order=None, 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. If it is not given, order 1 is used. It is different from `scipy.ndimage` and can change in the future. Currently it supports only order 0 and 1.
- **mode** *(str)* – Points outside the boundaries of the input are filled according to the given mode ("constant", "nearest", "mirror" or "opencv"). Default is "constant".
- **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

```
cupy.scipy.ndimage.shift(input, shift, output=None, order=None, 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**

- **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. If it is not given, order 1 is used. It is different from `scipy.ndimage` and can change in the future. Currently it supports only order 0 and 1.
- **mode** (*str*) – Points outside the boundaries of the input are filled according to the given mode ("constant", "nearest", "mirror" or "opencv"). Default is "constant".
- **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()`

cupy.scipy.ndimage.zoom

```
cupy.scipy.ndimage.zoom(input, zoom, output=None, order=None, mode='constant', cval=0.0, prefilter=True)
```
Zoom an array.

The array is zoomed using spline interpolation of the requested order.

**Parameters**

- **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. If it is not given, order 1 is used. It is different from `scipy.ndimage` and can change in the future. Currently it supports only order 0 and 1.
• **mode** (*str*) – Points outside the boundaries of the input are filled according to the given mode ('constant', 'nearest', 'mirror' or 'opencv'). Default is 'constant'.

• **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 zoomed input.

**Return type** *cupy.ndarray* or None

See also:

`scipy.ndimage.zoom()`

### Measurements

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cupyx.scipy.ndimage.label</code></td>
<td>Labels features in an array.</td>
</tr>
<tr>
<td><code>cupyx.scipy.ndimage.mean</code></td>
<td>Calculates the mean of the values of an n-D image array, optionally</td>
</tr>
<tr>
<td><code>cupyx.scipy.ndimage.standard_deviation</code></td>
<td>Calculates the standard deviation of the values of an n-D image array, optionally at specified sub-regions.</td>
</tr>
<tr>
<td><code>cupyx.scipy.ndimage.sum</code></td>
<td>Calculates the sum of the values of an n-D image array, optionally</td>
</tr>
<tr>
<td><code>cupyx.scipy.ndimage.variance</code></td>
<td>Calculates the variance of the values of an n-D image array, optionally at specified sub-regions.</td>
</tr>
</tbody>
</table>

### `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.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.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
cupyx.scipy.ndimage.sum(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()

cupy.scipy.ndimage.variance

cupy.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** variance *(cupy.ndarray)*

See also:
scipy.ndimage.variance()

**Morphology**

cupy.scipy.ndimage.grey_closing

cupy.scipy.ndimage.grey_closing *(input, size=None, footprint=None, structure=None, out=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. The `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: *cupy.ndarray*

```python
scipy.ndimage.grey_opening()
```

### OpenCV mode

`cupyx.scipy.ndimage` supports additional mode, opencv. If it is given, the function performs like cv2.warpAffine or cv2.resize. Example:

```python
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
```

## 4.4.5 Sparse matrices

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**

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cupyx.scipy.sparse.coo_matrix</code></td>
<td>COOrdinate format sparse matrix.</td>
</tr>
<tr>
<td><code>cupyx.scipy.sparse.csc_matrix</code></td>
<td>Compressed Sparse Column matrix.</td>
</tr>
<tr>
<td><code>cupyx.scipy.sparse.csr_matrix</code></td>
<td>Compressed Sparse Row matrix.</td>
</tr>
<tr>
<td><code>cupyx.scipy.sparse.dia_matrix</code></td>
<td>Sparse matrix with DIAgonal storage.</td>
</tr>
<tr>
<td><code>cupyx.scipy.sparse.spmatrix</code></td>
<td>Base class of all sparse matrixes.</td>
</tr>
</tbody>
</table>

**cupyx.scipy.sparse.coo_matrix**

```python
class cupyx.scipy.sparse.coo_matrix(arg1, shape=None, dtype=None, copy=False)
```

COOrdinate format sparse matrix.

Now it has only one initializer format below:

- `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-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 data are always used.

**See also:**

`scipy.sparse.coo_matrix`

**Methods**

- `__len__()`
- `__iter__()`
- `arcsin()`  
  Elementwise arcsin.

---

4.4. SciPy-compatible Routines 245
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 entries, which does not include explicit zero entries. Instead \texttt{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 \(i+k\)
- \(a[i,]\) Default \((a[i,] - 0\) (the main diagonal).

**Returns** The \(k\)-th diagonal.

**Return type** \texttt{cupy.ndarray}

dot \((other)\)
Ordinary dot product.

eliminate_zeros()
Removes zero entries in place.

expm1()
Elementwise \texttt{expm1}.

floor()
Elementwise \texttt{floor}.

get \((stream=None)\)
Returns a copy of the array on host memory.

**Parameters**

- \texttt{stream} \((\text{cupy.cuda.Stream})\) – 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** \texttt{scipy.sparse.coo_matrix}

getH()

get_shape()
Returns the shape of the matrix.

**Returns** Shape of the matrix.

**Return type** \texttt{tuple}

getformat()

getmaxprint()

getnnz \((axis=None)\)
Returns the number of stored values, including explicit zeros.

log1p()
Elementwise \texttt{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)*

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)*

**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], 'f')
>>> row = cupy.array([0, 1, 2, 2, 2], 'i')
>>> col = cupy.array([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, 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
cupy.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
cupy.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
cupy.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
cupy.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'
dim
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.

Now it has only part of initializer formats:

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, indices, indptr)) All data, indices and indptr are one-dimensional cupy.ndarray.

Parameters
    • arg1 – Arguments for the initializer.
CuPy Documentation, Release 8.2.0

- **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 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.narray 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 entries, 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
  - i+k

Default (a[i] – 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.

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 ()

generate_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`

```python
getformat()
```

```python
getmaxprint()
```

```python
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

```python
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`

```python
log1p()
```

Elementwise log1p.

```python
max(axis=None, out=None, *, explicit=False)
```

Returns the maximum of the matrix or maximum along an axis.

Parameters

- **axis** (int or 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

```python
maximum(other)
```

```python
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)`.

4.4. SciPy-compatible Routines 255
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)
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.

4.4. SciPy-compatible Routines
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 COOOrdinate 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 LLinked 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`

```python
trunc()
Elementwise trunc.
```

```python
__eq__(other)
Return self==value.
```

```python
__ne__(other)
Return self!=value.
```

```python
__lt__(other)
Return self<value.
```

```python
__le__(other)
Return self<=value.
```

```python
__gt__(other)
Return self>value.
```

```python
__ge__(other)
Return self>=value.
```

```python
__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.

4.4. SciPy-compatible Routines
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.

ndim
nnz
shape
size

cupy.scipy.sparse.csr_matrix

class cupyx.scipy.sparse.csr_matrix(arg1, shape=None, dtype=None, copy=False)
Compressed Sparse Row matrix.

Now it has only part of initializer formats:

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, 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.narray 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.narray 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 entries, 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

• i+k] Default (a[i] = 0 (the main diagonal).

**Returns** The k-th diagonal.
Return type \( \texttt{cupy.ndarray} \)

dot \( \text{(other)} \)
Ordinary dot product

eliminate_zeros ()
Removes zero entries in place.

expm1 ()
Elementwise expm1.

floor ()
Elementwise floor.

get \( \text{(stream=None)} \)
Returns a copy of the array on host memory.

Parameters

- \text{stream} \ (\texttt{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 \( \texttt{scipy.sparse.csr_matrix} \)

getH ()

get_shape ()
Returns the shape of the matrix.

Returns
Shape of the matrix.

Return type \( \texttt{tuple} \)

getcol \( \text{(i)} \)
Returns a copy of column \( i \) of the matrix, as a \((m \times 1)\) CSR matrix (column vector).

Parameters

- \text{i} \ (\texttt{integer}) – Column

Returns
Sparse matrix with single column

Return type \( \texttt{cupyx.scipy.sparse.csr_matrix} \)

getformat ()

getmaxprint ()

getnnz \( \text{(axis=None)} \)
Returns the number of stored values, including explicit zeros.

Parameters

- \text{axis} \ – Not supported yet.

Returns
The number of stored values.

Return type \( \texttt{int} \)

getrow \( \text{(i)} \)
Returns a copy of row \( i \) of the matrix, as a \((1 \times n)\) CSR matrix (row vector).

Parameters

- \text{i} \ (\texttt{integer}) – Row

Returns
Sparse matrix with single row

Return type \( \texttt{cupyx.scipy.sparse.csr_matrix} \)

log1p ()
Elementwise log1p.

4.4. SciPy-compatible Routines
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.
- mean: Compute the arithmetic mean along the specified axis.

**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

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)

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.

4.4. SciPy-compatible Routines
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.
tocoocopy=False
Converts the matrix to COOdinate format.

- **Parameters**
  - *copy* (bool) – If False, it shares data arrays as much as possible.

- **Returns**
  - Converted matrix.

- **Return type**
  - cupyx.scipy.sparse.coo_matrix
tocsccopy=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
tocsrcopy=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
todenseorder=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.
tollcopy=False
Convert this matrix to LLinked List format.

transposeaxes=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**
cupy.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**
cupy.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 entries, 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 `i+k`.
  - Default *(a[i,]) – 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.

---

4.4. SciPy-compatible Routines
**reshape** \((shape, order='C')\)
Gives a new shape to a sparse matrix without changing its data.

**rint()**
Elementwise rint.

**set_shape** \((shape)\)

**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 \(\{\text{None}, 0, 1, -2, -1\}\).
- **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** \((\text{bool})\) – 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**  
cupy.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
  
  * i+k  
  
  **Default** *(a[i, ] = 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)*  
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**
CuPy Documentation, Release 8.2.0

- \{\{-2, -1, 0, 1, \textbf{None}\}\} \text{(axis)} – optional Axis along which the mean is computed. The default is to compute the mean of all elements in the matrix (i.e., \texttt{axis = None}).
- \texttt{dtype} \text{(dtype)} – optional Type to use in computing the mean. For integer inputs, the default is \texttt{float64}; for floating point inputs, it is the same as the input \	exttt{dtype}.
- \texttt{out} \text{(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.

\textbf{Returns}  
Output array of means

\textbf{Return type} m \text{(cupy.ndarray)}

\textbf{See also:}
- \texttt{scipy.sparse.spmatrix.mean()}
- \texttt{minimum} \text{(other)}
- \texttt{multiply} \text{(other)}Point-wise multiplication by another matrix
- \texttt{power} \text{(n, dtype=\texttt{None})}
- \texttt{reshape} \text{(shape, order=’C’)}  
Gives a new shape to a sparse matrix without changing its data.
- \texttt{set\_shape} \text{(shape)}
- \texttt{sum} \text{(axis=\texttt{None}, dtype=\texttt{None}, out=\texttt{None})}Summs the matrix elements over a given axis.

\textbf{Parameters}
- \texttt{axis} \text{(int or \texttt{None})} – Axis along which the sum is computed. If it is \texttt{None}, it computes the sum of all the elements. Select from \{\texttt{None, 0, 1, -2, -1}\}.
- \texttt{dtype} – The type of returned matrix. If it is not specified, type of the array is used.
- \texttt{out} \text{(cupy.ndarray)} – Output matrix.

\textbf{Returns}  
Summed array.

\textbf{Return type} cupy.ndarray

\textbf{See also:}
- \texttt{scipy.sparse.spmatrix.sum()}
- \texttt{toarray} \text{(order=\texttt{None}, out=\texttt{None})}Return a dense ndarray representation of this matrix.
- \texttt{tobsr} \text{(blocksize=\texttt{None}, copy=\texttt{False})}Convert this matrix to Block Sparse Row format.
- \texttt{tocoo} \text{(copy=\texttt{False})}Convert this matrix to COOrdinate format.
- \texttt{tocsc} \text{(copy=\texttt{False})}Convert this matrix to Compressed Sparse Column format.
- \texttt{tocsr} \text{(copy=\texttt{False})}Convert this matrix to Compressed Sparse Row format.
- \texttt{todense} \text{(order=\texttt{None}, out=\texttt{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
CuPy Documentation, Release 8.2.0

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cupyx.scipy.sparse.bmat</code></td>
<td>Builds a sparse matrix from sparse sub-blocks</td>
</tr>
<tr>
<td><code>cupyx.scipy.sparse.diags</code></td>
<td>Construct a sparse matrix from diagonals.</td>
</tr>
<tr>
<td><code>cupyx.scipy.sparse.eye</code></td>
<td>Creates a sparse matrix with ones on diagonal.</td>
</tr>
<tr>
<td><code>cupyx.scipy.sparse.hstack</code></td>
<td>Stacks sparse matrices horizontally (column wise)</td>
</tr>
<tr>
<td><code>cupyx.scipy.sparse.identity</code></td>
<td>Creates an identity matrix in sparse format.</td>
</tr>
<tr>
<td><code>cupyx.scipy.sparse.kron</code></td>
<td>Kronecker product of sparse matrices A and B.</td>
</tr>
<tr>
<td><code>cupyx.scipy.sparse.spdiags</code></td>
<td>Creates a sparse matrix from diagonals.</td>
</tr>
<tr>
<td><code>cupyx.scipy.sparse.rand</code></td>
<td>Generates a random sparse matrix.</td>
</tr>
<tr>
<td><code>cupyx.scipy.sparse.random</code></td>
<td>Generates a random sparse matrix.</td>
</tr>
<tr>
<td><code>cupyx.scipy.sparse.vstack</code></td>
<td>Stacks sparse matrices vertically (row wise)</td>
</tr>
</tbody>
</table>

### `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**

```python
>>> 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.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:

```python
cupy.diag(diagonals[0], offsets[0])
+ ...
+ cupy.diag(diagonals[k], offsets[k])
```

Repeated diagonal offsets are disallowed.

**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:

scipysparse.eye()
**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**

```python
>>> 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.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()`
CuPy Documentation, Release 8.2.0

**cupyx.scipy.sparse.kron**

cupy.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.spdiags**

cupy.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.rand**

cupy.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:

```
sct.scipy.rand()
```

See also:

```
cupy.scipy.sparse.random()
```

### 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:

```
sct.scipy.rand()
```

### cupyx.scipy.sparse.vstack

**cupyx.scipy.sparse.vstack**

`cupyx.scipy.sparse.vstack(blocks, format=None, dtype=None)`

Stacks sparse matrices vertically (row wise)

**Parameters**
CuPy Documentation, Release 8.2.0

- **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

```python
>>> 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.]]
```

Identifying sparse matrices

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cupyx.scipy.sparse.issparse</td>
<td>Checks if a given matrix is a sparse matrix.</td>
</tr>
<tr>
<td>cupyx.scipy.sparse.isspmatrix</td>
<td>Checks if a given matrix is a sparse matrix.</td>
</tr>
<tr>
<td>cupyx.scipy.sparse.isspmatrix_csc</td>
<td>Checks if a given matrix is of CSC format.</td>
</tr>
<tr>
<td>cupyx.scipy.sparse.isspmatrix_csr</td>
<td>Checks if a given matrix is of CSR format.</td>
</tr>
<tr>
<td>cupyx.scipy.sparse.isspmatrix_coo</td>
<td>Checks if a given matrix is of COO format.</td>
</tr>
<tr>
<td>cupyx.scipy.sparse.isspmatrix_dia</td>
<td>Checks if a given matrix is of DIA format.</td>
</tr>
</tbody>
</table>

`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 \texttt{cupyx.scipy.sparse.spmatrix} that is a base class of all sparse matrix classes.

Return type  \( \text{bool} \)

cupy.scipy.sparse.isspmatrix_csc

cupy.scipy.sparse.\texttt{isspmatrix\_csc}(x)
Checks if a given matrix is of CSC format.

Returns  Returns if \( x \) is \texttt{cupyx.scipy.sparse.csc\_matrix}.

Return type  \( \text{bool} \)

cupy.scipy.sparse.isspmatrix_csr

cupy.scipy.sparse.\texttt{isspmatrix\_csr}(x)
Checks if a given matrix is of CSR format.

Returns  Returns if \( x \) is \texttt{cupyx.scipy.sparse.csr\_matrix}.

Return type  \( \text{bool} \)

cupy.scipy.sparse.isspmatrix_coo

cupy.scipy.sparse.\texttt{isspmatrix\_coo}(x)
Checks if a given matrix is of COO format.

Returns  Returns if \( x \) is \texttt{cupyx.scipy.sparse.coo\_matrix}.

Return type  \( \text{bool} \)

cupy.scipy.sparse.isspmatrix_dia

cupy.scipy.sparse.\texttt{isspmatrix\_dia}(x)
Checks if a given matrix is of DIA format.

Returns  Returns if \( x \) is \texttt{cupyx.scipy.sparse.dia\_matrix}.

Return type  \( \text{bool} \)

Linear Algebra

cupy.scipy.sparse.linalg.\texttt{lsqr}
Solves linear system with QR decomposition.

cupy.scipy.sparse.linalg.\texttt{norm}
Norm of a \texttt{cupy.scipy.spmatrix}

cupyx.scipy.sparse.linalg.\texttt{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 \( Q \times R \).

**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 \( \text{None} \) because the implementation of cuSOLVER is different from the one of SciPy. You can easily calculate the fourth element by \( \text{norm}(b - Ax) \) and the ninth element by \( \text{norm}(x) \).

**Return type** tuple

See also:

scipy.sparse.linalg.lsqr()

cupy.scipy.sparse.linalg.norm

cupy.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 \( \text{ord} \) parameter.

**Parameters**

- \( x \) (sparse matrix) – Input sparse matrix.
- \( \text{ord} \) (non-zero int, inf, -inf, 'fro', optional) – Order of the norm (see table under Notes). inf means numpy’s \( \text{inf} \) object.
- \( \text{axis} \) – (int, 2-tuple of ints, None, optional): If \( \text{axis} \) is an integer, it specifies the axis of \( x \) along which to compute the vector norms. If \( \text{axis} \) is a 2-tuple, it specifies the axes that hold 2-D matrices, and the matrix norms of these matrices are computed. If \( \text{axis} \) 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()

### 4.4.6 Special Functions

**Bessel Functions**

<table>
<thead>
<tr>
<th>cupyx.scipy.special.j0</th>
<th>Bessel function of the first kind of order 0.</th>
</tr>
</thead>
<tbody>
<tr>
<td>cupyx.scipy.special.j1</td>
<td>Bessel function of the first kind of order 1.</td>
</tr>
<tr>
<td>cupyx.scipy.special.y0</td>
<td>Bessel function of the second kind of order 0.</td>
</tr>
<tr>
<td>cupyx.scipy.special.y1</td>
<td>Bessel function of the second kind of order 1.</td>
</tr>
<tr>
<td>cupyx.scipy.special.10</td>
<td>Modified Bessel function of order 0.</td>
</tr>
<tr>
<td>cupyx.scipy.special.11</td>
<td>Modified Bessel function of order 1.</td>
</tr>
</tbody>
</table>

### 4.4. SciPy-compatible Routines
cupyx.scipy.special.j0

cupyx.scipy.special.j0 = <ufunc 'cupyx_scipy_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_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_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_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_i0'>
Modified Bessel function of order 0.

See also:
scipy.special.i0()

cupyx.scipy.special.i1

cupyx.scipy.special.i1 = <ufunc 'cupyx_scipy_i1'>
Modified Bessel function of order 1.

See also:
scipy.special.i1()
Information Theory Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>See also</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cupyx.scipy.special.entr</code></td>
<td>Elementwise function for computing entropy.</td>
<td><code>scipy.special.entr()</code></td>
</tr>
<tr>
<td><code>cupyx.scipy.special.huber</code></td>
<td>Elementwise function for computing the Huber loss.</td>
<td><code>scipy.special.huber()</code></td>
</tr>
<tr>
<td><code>cupyx.scipy.special.kl_div</code></td>
<td>Elementwise function for computing Kullback-Leibler divergence.</td>
<td><code>scipy.special.kl_div()</code></td>
</tr>
<tr>
<td><code>cupyx.scipy.special.pseudo_huber</code></td>
<td>Elementwise function for computing the Pseudo-Huber loss.</td>
<td><code>scipy.special.pseudo_huber()</code></td>
</tr>
<tr>
<td><code>cupyx.scipy.special.rel_entr</code></td>
<td>Elementwise function for computing relative entropy.</td>
<td><code>scipy.special.rel_entr()</code></td>
</tr>
</tbody>
</table>

**cupyx.scipy.special.entr**

cupyx.scipy.special.entr = <ufunc 'cupyx_scipy_entr'>
Elementwise function for computing entropy.

See also:
scipy.special.entr()

**cupyx.scipy.special.huber**

cupyx.scipy.special.huber = <ufunc 'cupyx_scipy_huber'>
Elementwise function for computing the Huber loss.

See also:
scipy.special.huber()

**cupyx.scipy.special.kl_div**

cupyx.scipy.special.kl_div = <ufunc 'cupyx_scipy_kl_div'>
Elementwise function for computing Kullback-Leibler divergence.

See also:
scipy.special.kl_div()

**cupyx.scipy.special.pseudo_huber**

cupyx.scipy.special.pseudo_huber = <ufunc 'cupyx_scipy_pseudo_huber'>
Elementwise function for computing the Pseudo-Huber loss.

See also:
scipy.special.pseudo_huber()

**cupyx.scipy.special.rel_entr**

cupyx.scipy.special.rel_entr = <ufunc 'cupyx_scipy_rel_entr'>
Elementwise function for computing relative entropy.

See also:
scipy.special.rel_entr()
Gamma and Related Functions

<table>
<thead>
<tr>
<th>cupyx.scipy.special.gamma</th>
<th>Gamma function.</th>
</tr>
</thead>
<tbody>
<tr>
<td>cupyx.scipy.special.gammaln</td>
<td>Logarithm of the absolute value of the Gamma function.</td>
</tr>
<tr>
<td>cupyx.scipy.special.polygamma</td>
<td>Polygamma function n.</td>
</tr>
<tr>
<td>cupyx.scipy.special.digamma</td>
<td>The digamma function.</td>
</tr>
</tbody>
</table>

**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_gammaln'>`

Logarithm of the absolute value of the Gamma function.

**Parameters**
- `x (cupy.ndarray)` – Values on the real line at which to compute

**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_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

---

**Raw Statistical Functions**

<table>
<thead>
<tr>
<th><code>cupyx.scipy.special.ndtr</code></th>
<th>Cumulative distribution function of normal distribution.</th>
</tr>
</thead>
</table>

**cupyx.scipy.special.ndtr**

`cupyx.scipy.special.ndtr = <ufunc 'cupyx_scipy_ndtr'>`

Cumulative distribution function of normal distribution.

**See also:**

scipy.special.ndtr()

---

**Error Function**

<table>
<thead>
<tr>
<th><code>cupyx.scipy.special.erf</code></th>
<th>Error function.</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cupyx.scipy.special.erfc</code></td>
<td>Complementary error function.</td>
</tr>
<tr>
<td><code>cupyx.scipy.special.erfcx</code></td>
<td>Scaled complementary error function.</td>
</tr>
<tr>
<td><code>cupyx.scipy.special.erfinv</code></td>
<td>Inverse function of error function.</td>
</tr>
<tr>
<td><code>cupyx.scipy.special.erfcinv</code></td>
<td>Inverse function of complementary error function.</td>
</tr>
</tbody>
</table>

**cupyx.scipy.special.erf**

`cupyx.scipy.special.erf = <ufunc 'cupyx_scipy_erf'>`

Error function.

**See also:**

scipy.special.erf()

---

**cupyx.scipy.special.erfc**

`cupyx.scipy.special.erfc = <ufunc 'cupyx_scipy_erfc'>`

Complementary error function.

**See also:**

scipy.special.erfc()
**cupyx.scipy.special.erfcx**

cupyx.scipy.special.erfcx = <ufunc 'cupyx_scipy_erfcx'>

Scaled complementary error function.

See also:
scipy.special.erfcx()

**cupyx.scipy.special.erfinv**

cupyx.scipy.special.erfinv = <ufunc 'cupyx_scipy_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_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**

cupyx.scipy.special.zeta

Hierwitz zeta function.

**cupyx.scipy.special.zeta**

cupyx.scipy.special.zeta = <ufunc 'cupyx_scipy_zeta'>

Hierwitz 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
4.4.7 Signal processing

Convolution

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cupyx.scipy.signal.choose_conv_method</code></td>
<td>Find the fastest convolution/correlation method.</td>
</tr>
<tr>
<td><code>cupyx.scipy.signal.convolve</code></td>
<td>Convolve two N-dimensional arrays.</td>
</tr>
<tr>
<td><code>cupyx.scipy.signal.convolve2d</code></td>
<td>Convolve two 2-dimensional arrays.</td>
</tr>
<tr>
<td><code>cupyx.scipy.signal.correlate</code></td>
<td>Cross-correlate two N-dimensional arrays.</td>
</tr>
<tr>
<td><code>cupyx.scipy.signal.correlate2d</code></td>
<td>Cross-correlate two 2-dimensional arrays.</td>
</tr>
</tbody>
</table>

`cupyx.scipy.signal.choose_conv_method`

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 fft1.

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()`

`cupyx.scipy.signal.convolve`

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 of FFT based on an estimate of which is faster for the arguments (default).

Returns the result of convolution.

Return type cupy.ndarray

See also:
cupy.scipy.signal.choose_conv_method()

See also:
cupy.scipy.signal.correlation()

See also:
cupy.scipy.signal.fftconvolve()

See also:
cupy.scipy.signal.oaconvolve()

See also:
cupy.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.

cupy.scipy.signal.convolve2d

cupy.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()

cupy.scipy.signal.correlate

cupy.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 \texttt{in1}, centered with respect to the 'full' output

- \texttt{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 \texttt{fftconvolve}.
  - 'auto': Automatically choose direct of FFT based on an estimate of which is faster for the arguments (default).

  \textbf{Returns} the result of correlation.

  \textbf{Return type} \texttt{cupy.ndarray}

\textbf{See also:} \texttt{cupyx.scipy.signal.choose_conv_method()}

\textbf{See also:} \texttt{cupyx.scipy.signal.convolve()}

\textbf{See also:} \texttt{cupyx.scipy.signal.fftconvolve()}

\textbf{See also:} \texttt{cupyx.scipy.signal.oaconvolve()}

\textbf{See also:} \texttt{cupyx.scipy.ndimage.correlation()}

\textbf{See also:} \texttt{scipy.signal.correlation()}

\textbf{Note:} By default, \texttt{convolve} and \texttt{correlate} use method='auto', which calls \texttt{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.

\textbf{cupyx.scipy.signal.correlate2d}

cupyx.scipy.signal.correlate2d \texttt{(in1, in2, mode='full', boundary='fill', fillvalue=0)}

Cross-correlate two 2-dimensional arrays.

Cross correlate \texttt{in1} and \texttt{in2} with output size determined by \texttt{mode}, and boundary conditions determined by \texttt{boundary} and \texttt{fillvalue}.

\textbf{Parameters}

- \texttt{in1 (cupy.ndarray)} – First input.
- \texttt{in2 (cupy.ndarray)} – Second input. Should have the same number of dimensions as \texttt{in1}.
- \texttt{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 \texttt{in1} or \texttt{in2} must be at least as large as the other in every dimension.

- 'same': output is the same size as \texttt{in1}, centered with respect to the 'full' output

- \textbf{boundary (str)} – Indicates how to handle boundaries:
  - \texttt{fill}: pad input arrays with \texttt{fillvalue} (default)
  - \texttt{wrap}: circular boundary conditions
  - \texttt{symm}: symmetrical boundary conditions

- \textbf{fillvalue (scalar)} – Value to fill pad input arrays with. Default is 0.

\textbf{Returns}

A 2-dimensional array containing a subset of the discrete linear cross-correlation of \texttt{in1} with \texttt{in2}.

\textbf{Return type} \texttt{cupy.ndarray}

\textbf{Note:} When using "same" mode with even-length inputs, the outputs of \texttt{correlate} and \texttt{correlate2d} differ: There is a 1-index offset between them.

\textbf{See also:}

- \texttt{cupyx.scipy.signal.correlate()}
- \texttt{cupyx.scipy.signal.convolve2d()}
- \texttt{cupyx.scipy.ndimage.correlate()}
- \texttt{scipy.signal.correlate2d()}

\section*{Filtering}

\begin{center}
\begin{tabular}{ll}
\texttt{cupyx.scipy.signal.order_filter} & Perform an order filter on an N-D array. \\
\texttt{cupyx.scipy.signal.medfilt} & Perform a median filter on an N-dimensional array. \\
\texttt{cupyx.scipy.signal.medfilt2d} & Median filter a 2-dimensional array. \\
\texttt{cupyx.scipy.signal.wiener} & Perform a Wiener filter on an N-dimensional array. \\
\end{tabular}
\end{center}

\textbf{cupyx.scipy.signal.order_filter}

\texttt{cupyx.scipy.signal.order_filter(a, domain, rank)}

Perform an order filter on an N-D array.

Perform an order filter on the array \texttt{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.

\textbf{Parameters}

- \texttt{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:
cupy.scipy.ndimage.rank_filter()

See also:
scipy.signal.order_filter()

cupy.scipy.signal.medfilt

cupy.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:
cupy.scipy.ndimage.median_filter()

See also:
scipy.signal.medfilt()

cupy.scipy.signal.medfilt2d

cupy.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 of 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()

### 4.5 NumPy-CuPy Generic Code Support

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cupy.get_array_module</td>
<td>Returns the array module for arguments.</td>
</tr>
<tr>
<td>cupyx.scipy.get_array_module</td>
<td>Returns the array module for arguments.</td>
</tr>
</tbody>
</table>

### 4.6 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.

### 4.6.1 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:

```python
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 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
```

4.6.2 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).

```python
# 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.

```python
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.

4.6.3 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`.

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

```python
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)
```
4.7 Low-Level CUDA Support

4.7.1 Device management

cupy.cu\da.Device

Object that represents a CUDA device.

cupy.cu\da.Device

class cupy.cu\da.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:

```python
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.cu\da.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.

**Attributes**

attributes

A dictionary of device attributes.
Returns Dictionary of attribute values with the names as keys. The string \texttt{cudaDevAttr} has been trimmed from the names. For example, the attribute corresponding to the enumerated value \texttt{cudaDevAttrMaxThreadsPerBlock} will have key \texttt{MaxThreadsPerBlock}.

Return type attributes (dict)

\begin{verbatim}
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)
\end{verbatim}

4.7.2 Memory management

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{cupy.get_default_memory_pool}</td>
<td>Returns CuPy default memory pool for GPU memory.</td>
</tr>
<tr>
<td>\texttt{cupy.get_default_pinned_memory_pool}</td>
<td>Returns CuPy default memory pool for pinned memory.</td>
</tr>
<tr>
<td>\texttt{cupy.cudarray.Memory}</td>
<td>Memory allocation on a CUDA device.</td>
</tr>
<tr>
<td>\texttt{cupy.cudarray.UnownedMemory}</td>
<td>CUDA memory that is not owned by CuPy.</td>
</tr>
<tr>
<td>\texttt{cupy.cudarray.PinnedMemory}</td>
<td>Pinned memory allocation on host.</td>
</tr>
<tr>
<td>\texttt{cupy.cudarray.MemoryPointer}</td>
<td>Pointer to a point on a device memory.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 100 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cupy.cuda.PinnedMemoryPointer</td>
<td>Pointer of a pinned memory.</td>
</tr>
<tr>
<td>cupy.cuda.alloc</td>
<td>Calls the current allocator.</td>
</tr>
<tr>
<td>cupy.cuda.alloc_pinned_memory</td>
<td>Calls the current allocator.</td>
</tr>
<tr>
<td>cupy.cuda.get_allocator</td>
<td>Returns the current allocator for GPU memory.</td>
</tr>
<tr>
<td>cupy.cuda.set_allocator</td>
<td>Sets the current allocator for GPU memory.</td>
</tr>
<tr>
<td>cupy.cuda.using_allocator</td>
<td>Sets a thread-local allocator for GPU memory inside</td>
</tr>
<tr>
<td>cupy.cuda.set_pinned_memory_allocator</td>
<td>Sets the current allocator for the pinned memory.</td>
</tr>
<tr>
<td>cupy.cuda.MemoryPool</td>
<td>Memory pool for all GPU devices on the host.</td>
</tr>
<tr>
<td>cupy.cuda.PinnedMemoryPool</td>
<td>Memory pool for pinned memory on the host.</td>
</tr>
<tr>
<td>cupy.cuda.PythonFunctionAllocator</td>
<td>Allocator with python functions to perform memory allocation.</td>
</tr>
</tbody>
</table>

**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

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

Attributes

device
device_id
   ‘int’
   Type device_id
ptr
   ‘intptr_t’
   Type ptr
size
   ‘size_t’
   Type size
cupy.cuMPa. Pinde Memory

class cupy.cuda.PinnedMemory
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

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 (ctypes. c_void_p or cupy.cuda.MemoryPointer) – Source memory pointer.
- size (int) – Size of the sequence in bytes.

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 (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.

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 (ctypes.c_void_p) – Source memory pointer.
• size (int) – Size of the sequence in bytes.

copy_from_host_async (self, mem, size_t size, stream=None)
Copies a memory sequence from the host memory asynchronously.

Parameters
• mem (ctypes.c_void_p) – Source memory pointer. It must be a 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 (ctypes.c_void_p) – Target memory pointer.
• size (int) – Size of the sequence in bytes.

copy_to_host_async (self, mem, size_t size, stream=None)
Copies a memory sequence to the host memory asynchronously.

Parameters
• mem (ctypes.c_void_p) – Target memory pointer. It must be a 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.

`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.

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

Attributes

mem
ptr
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.

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
CuPy Documentation, Release 8.2.0

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:

```python
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:

```python
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

```python
set_limit(fraction=0.5)
```
or

```python
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.
CuPy Documentation, Release 8.2.0

**Returns** The total number of bytes used.

**Return type** int

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

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.
CuPy Documentation, Release 8.2.0

Methods

\texttt{malloc}(self, size_t size) \rightarrow \texttt{MemoryPointer}

4.7.3 Memory hook

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cupy.cuda.MemoryHook</code></td>
<td>Base class of hooks for Memory allocations.</td>
</tr>
<tr>
<td><code>cupy.cuda.memory_hooks.DebugPrintHook</code></td>
<td>Memory hook that prints debug information.</td>
</tr>
<tr>
<td><code>cupy.cuda.memory_hooks.LineProfileHook</code></td>
<td>Code line CuPy memory profiler.</td>
</tr>
</tbody>
</table>

\texttt{cupy.cuda.MemoryHook}

class \texttt{cupy.cuda.MemoryHook}  
Base class of hooks for Memory allocations.

\texttt{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 \texttt{MemoryHook} are required to implement six methods: \texttt{alloc_preprocess()}, \texttt{alloc_postprocess()}, \texttt{malloc_preprocess()}, \texttt{malloc_postprocess()}, \texttt{free_preprocess()}, and \texttt{free_postprocess()}, By default, these methods do nothing.

Specifically, \texttt{alloc_preprocess()} (resp. \texttt{alloc_postprocess()}) of all memory hooks registered are called before (resp. after) memory is allocated from GPU device.

Likewise, \texttt{malloc_preprocess()} (resp. \texttt{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 \texttt{alloc_preprocess()} and \texttt{alloc_postprocess()} are not invoked if a cached free chunk is found:

```python
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, \texttt{free_preprocess()} (resp. \texttt{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:

```python
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 \texttt{with} statement. Memory hooks are registered to all method calls within \texttt{with} statement and are unregistered at the end of \texttt{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, *_)*

Attributes

`alloc_postprocess`  
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`  
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`  
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`  
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`  
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**
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

```python
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:

```python
>>> 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:

```json
["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**

```python
__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.

`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.

`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`.

**Attributes**

`alloc_preprocess`
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_preprocess`
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_preprocess**

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

```python
ame = 'DebugPrintHook'
```

cupy.cuda.memory_hooks.LineProfileHook

```python
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:

```python
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_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

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.

Attributes

alloc_postprocess
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.

free_postprocess
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
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
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.

name = 'LineProfileHook'

4.7.4 Streams and events

<table>
<thead>
<tr>
<th>cupy.cuda.Stream</th>
<th>CUDA stream.</th>
</tr>
</thead>
<tbody>
<tr>
<td>cupy.cuda.ExternalStream</td>
<td>CUDA stream.</td>
</tr>
<tr>
<td>cupy.cuda.get_current_stream</td>
<td>Gets current CUDA stream.</td>
</tr>
<tr>
<td>cupy.cuda.Event</td>
<td>CUDA event, a synchronization point of CUDA streams.</td>
</tr>
<tr>
<td>cupy.cuda.get_elapsed_time</td>
<td>Gets the elapsed time between two events.</td>
</tr>
</tbody>
</table>

cupy.cuda.Stream

class cupy.cuda.Stream
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.

Parameters
- `null` (`bool`) – If True, the stream is a null stream (i.e. the default stream that synchronizes with all streams). Otherwise, a plain new stream is created. Note that you can also use `Stream.null` singleton object instead of creating new null stream object.
- `non_blocking` (`bool`) – If True, the stream does not synchronize with the NULL stream.

Variables `ptr` (`intptr_t`) – Raw stream handle. It can be passed to the CUDA Runtime API via ctypes.

Methods

__enter__ (self)
__exit__ (self, *args)

Attributes

add_callback
Add 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.

**done**

True if all work on this stream has been done.

**null** = `<cupy.cuda.stream.Stream object>`

**record**

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**

Waits for the stream completing all queued work.

**use**

Makes this stream current.

If you want to switch a stream temporarily, use the `with` statement.

**wait_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.

**cupy.cuda.ExternalStream**

**class** `cupy.cuda.ExternalStream`

`cupy.cuda.ExternalStream` 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. It can be passed to the CUDA Runtime API via ctypes.

**Methods**

- `__enter__(self)`
- `__exit__(self, *args)`
Attributes

\texttt{add\_callback}

Adds a callback that is called when all queued work is done.

\textbf{Parameters}

- \texttt{callback} (function) – Callback function. It must take three arguments (Stream object, int error status, and user data object), and returns nothing.
- \texttt{arg} (object) – Argument to the callback.

\texttt{done}

True if all work on this stream has been done.

\texttt{null = None}

\texttt{record}

Records an event on the stream.

\textbf{Parameters} \texttt{event} (\texttt{None} or \texttt{cupy.cuda.Event}) – CUDA event. If \texttt{None}, then a new plain event is created and used.

\textbf{Returns} The recorded event.

\textbf{Return type} \texttt{cupy.cuda.Event}

\textbf{See also:}

\texttt{cupy.cuda.Event.record()}

\texttt{synchronize}

Waits for the stream completing all queued work.

\texttt{use}

Makes this stream current.

If you want to switch a stream temporarily, use the \texttt{with} statement.

\texttt{wait\_event}

Makes the stream wait for an event.

The future work on this stream will be done after the event.

\textbf{Parameters} \texttt{event} (\texttt{cupy.cuda.Event}) – CUDA event.

\texttt{cupy.cuda.get\_current\_stream}

\texttt{cupy.cuda.get\_current\_stream()}

Gets current CUDA stream.

\textbf{Returns} The current CUDA stream.

\textbf{Return type} \texttt{cupy.cuda.Stream}

\texttt{cupy.cuda.Event}

\textbf{class} \texttt{cupy.cuda.Event}

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 stream handle. It can be passed to the CUDA Runtime API via ctypes.

Methods

Attributes

- **done** True if the event is done.
- **record** 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**

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.

**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`

### 4.7.5 Texture and surface memory

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cupy.cuda.texture.ChannelFormatDescriptor</code></td>
<td>A class that holds the channel format description.</td>
</tr>
<tr>
<td><code>cupy.cuda.texture.CUDAArray</code></td>
<td>Allocate a CUDA array (<code>cudaArray_t</code>) that can be used as texture memory.</td>
</tr>
<tr>
<td><code>cupy.cuda.texture.ResourceDescriptor</code></td>
<td>A class that holds the resource description.</td>
</tr>
<tr>
<td><code>cupy.cuda.texture.TextureDescriptor</code></td>
<td>A class that holds the texture description.</td>
</tr>
<tr>
<td><code>cupy.cuda.texture.TextureObject</code></td>
<td>A class that holds a texture object.</td>
</tr>
</tbody>
</table>
cuPy Documentation, Release 8.2.0

Table 103 – continued from previous page

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cupy.cuda.texture.SurfaceObject</code></td>
<td>A class that holds a surface object.</td>
</tr>
<tr>
<td><code>cupy.cuda.texture.TextureReference</code></td>
<td>A class that holds a texture reference.</td>
</tr>
</tbody>
</table>

**cupy.cuda.texture.ChannelFormatDescriptor**

```python
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**

```python
get_channel_format(self)
```

Returns a dict containing the input.

**Attributes**

- `ptr`

**cupy.cuda.texture.CUDAarray**

```python
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) –
- `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`.

Attributes

- `depth`
- `desc`
- `flags`
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.
Attributes

arr
chDesc
cuArr
ptr

cupy.cuda.texture.TextureDescriptor

class cupy.cudatexture.TextureDescriptor(addressModes=None, int filterMode=0, int readMode=0, sRGB=None, borderColor=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.

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 an argument when launching RawKernel.

Parameters

- TexDesc (TextureDescriptor) – an instance of the texture descriptor.

See also:
cudaCreateTextureObject()

Methods

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


See also:
cudaCreateSurfaceObject()

Methods

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().

• **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

### Attributes

*ResDesc*

*TexDesc*

*texref*

### 4.7.6 Profiler

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cupy.cuda.profile</code></td>
<td>Enable CUDA profiling during with statement.</td>
</tr>
<tr>
<td><code>cupy.cuda.profiler.initialize</code></td>
<td>Initialize the CUDA profiler.</td>
</tr>
<tr>
<td><code>cupy.cuda.profiler.start</code></td>
<td>Enable profiling.</td>
</tr>
<tr>
<td><code>cupy.cuda.profiler.stop</code></td>
<td>Disable profiling.</td>
</tr>
<tr>
<td><code>cupy.cuda.nvtx.Mark</code></td>
<td>Marks an instantaneous event (marker) in the application.</td>
</tr>
<tr>
<td><code>cupy.cuda.nvtx.MarkC</code></td>
<td>Marks an instantaneous event (marker) in the application.</td>
</tr>
<tr>
<td><code>cupy.cuda.nvtx.RangePush</code></td>
<td>Starts a nested range.</td>
</tr>
<tr>
<td><code>cupy.cuda.nvtx.RangePushC</code></td>
<td>Starts a nested range.</td>
</tr>
<tr>
<td><code>cupy.cuda.nvtx.RangePop</code></td>
<td>Ends a nested range.</td>
</tr>
</tbody>
</table>

**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.

```python
>>> with cupy.cuda.profile():
...     # do something you want to measure
...     pass
```
**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 Documentation, Release 8.2.0

```python
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.

4.7.7 NCCL

```python
cupy.cuda.nccl.NcclCommunicator

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 buf, 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:

```python
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 recvcount, int datatype, int op, intptr_t stream)

send (self, intptr_t sendbuf, size_t count, int datatype, int peer, intptr_t stream)

size (self)

cupy.cu.inna.cl.get_build_version

cupy.cuda.nccl.get_build_version()

cupy.cu.inna.cl.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.cu.inna.cl.get_unique_id

cupy.cuda.nccl.get_unique_id()

cupy.cu.inna.cl.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:

```python
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.cu.inna.cl.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

### 4.7.8 Runtime API

CuPy wraps CUDA Runtime APIs to provide the native CUDA operations. Please check the Original CUDA Runtime API document to use these functions.

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>cupy.cuda.runtime.driverGetVersion</td>
</tr>
<tr>
<td>cupy.cuda.runtime.runtimeGetVersion</td>
</tr>
<tr>
<td>cupy.cuda.runtime.getDevice</td>
</tr>
<tr>
<td>cupy.cuda.runtime.deviceGetAttribute</td>
</tr>
<tr>
<td>cupy.cuda.runtime.getDeviceByPCIBusId</td>
</tr>
<tr>
<td>cupy.cuda.runtime.deviceGetPCIBusId</td>
</tr>
<tr>
<td>cupy.cuda.runtime.getDeviceCount</td>
</tr>
<tr>
<td>cupy.cuda.runtime.setDevice</td>
</tr>
<tr>
<td>cupy.cuda.runtime.deviceSynchronize</td>
</tr>
<tr>
<td>cupy.cuda.runtime.deviceCanAccessPeer</td>
</tr>
<tr>
<td>cupy.cuda.runtime.deviceEnablePeerAccess</td>
</tr>
<tr>
<td>cupy.cuda.runtime.deviceGetLimit</td>
</tr>
<tr>
<td>cupy.cuda.runtime.deviceSetLimit</td>
</tr>
<tr>
<td>cupy.cuda.runtime.malloc</td>
</tr>
<tr>
<td>cupy.cuda.runtime.mallocManaged</td>
</tr>
<tr>
<td>cupy.cuda.runtime.malloc3DArray</td>
</tr>
<tr>
<td>cupy.cuda.runtime.mallocArray</td>
</tr>
<tr>
<td>cupy.cuda.runtime.hostAlloc</td>
</tr>
<tr>
<td>cupy.cuda.runtime.hostRegister</td>
</tr>
<tr>
<td>cupy.cuda.runtime.hostUnregister</td>
</tr>
<tr>
<td>cupy.cuda.runtime.free</td>
</tr>
<tr>
<td>cupy.cuda.runtime.freeHost</td>
</tr>
<tr>
<td>cupy.cuda.runtime.freeArray</td>
</tr>
<tr>
<td>cupy.cuda.runtime.memGetInfo</td>
</tr>
<tr>
<td>cupy.cuda.runtime.memcpy</td>
</tr>
<tr>
<td>cupy.cuda.runtime.memcpyAsync</td>
</tr>
<tr>
<td>cupy.cuda.runtime.memcpyPeer</td>
</tr>
<tr>
<td>cupy.cuda.runtime.memcpyPeerAsync</td>
</tr>
<tr>
<td>cupy.cuda.runtime.memcpy2D</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Function Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cupy.cuda.runtime.memcpy2DAsync</code></td>
</tr>
<tr>
<td><code>cupy.cuda.runtime.memcpy2DFromArray</code></td>
</tr>
<tr>
<td><code>cupy.cuda.runtime.memcpy2DFromArrayAsync</code></td>
</tr>
<tr>
<td><code>cupy.cuda.runtime.memcpy2DToArray</code></td>
</tr>
<tr>
<td><code>cupy.cuda.runtime.memcpy2DToArrayAsync</code></td>
</tr>
<tr>
<td><code>cupy.cuda.runtime.memcpy3D</code></td>
</tr>
<tr>
<td><code>cupy.cuda.runtime.memcopy3DAsync</code></td>
</tr>
<tr>
<td><code>cupy.cuda.runtime.memset</code></td>
</tr>
<tr>
<td><code>cupy.cuda.runtime.memsetAsync</code></td>
</tr>
<tr>
<td><code>cupy.cuda.runtime.memPrefetchAsync</code></td>
</tr>
<tr>
<td><code>cupy.cuda.runtime.memAdvise</code></td>
</tr>
<tr>
<td><code>cupy.cuda.runtime.pointerGetAttributes</code></td>
</tr>
<tr>
<td><code>cupy.cuda.runtime.streamCreate</code></td>
</tr>
<tr>
<td><code>cupy.cuda.runtime.streamCreateWithFlags</code></td>
</tr>
<tr>
<td><code>cupy.cuda.runtime.streamDestroy</code></td>
</tr>
<tr>
<td><code>cupy.cuda.runtime.streamSynchronize</code></td>
</tr>
<tr>
<td><code>cupy.cuda.runtime.streamAddCallback</code></td>
</tr>
<tr>
<td><code>cupy.cuda.runtime.streamQuery</code></td>
</tr>
<tr>
<td><code>cupy.cuda.runtime.streamWaitEvent</code></td>
</tr>
<tr>
<td><code>cupy.cuda.runtime.eventCreate</code></td>
</tr>
<tr>
<td><code>cupy.cuda.runtime.eventCreateWithFlags</code></td>
</tr>
<tr>
<td><code>cupy.cuda.runtime.eventDestroy</code></td>
</tr>
<tr>
<td><code>cupy.cuda.runtime.eventElapsedTime</code></td>
</tr>
<tr>
<td><code>cupy.cuda.runtime.eventQuery</code></td>
</tr>
<tr>
<td><code>cupy.cuda.runtime.eventRecord</code></td>
</tr>
<tr>
<td><code>cupy.cuda.runtime.eventSynchronize</code></td>
</tr>
<tr>
<td><code>cupy.cuda.runtime.ipcGetMemHandle</code></td>
</tr>
<tr>
<td><code>cupy.cuda.runtime.ipcOpenMemHandle</code></td>
</tr>
<tr>
<td><code>cupy.cuda.runtime.ipcCloseMemHandle</code></td>
</tr>
<tr>
<td><code>cupy.cuda.runtime.ipcGetEventHandle</code></td>
</tr>
<tr>
<td><code>cupy.cuda.runtime.ipcOpenEventHandle</code></td>
</tr>
</tbody>
</table>

**CuPy Documentation, Release 8.2.0**

**332 Chapter 4. API Reference**
cupy.cuda.runtime.deviceGetAttribute

cupy.cu.am.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.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

4.7. Low-Level CUDA Support
**CuPy Documentation, Release 8.2.0**

```python
cupy.cuda.runtime.mallocManaged

cupy.cuda.runtime.mallocManaged(size_t size, unsigned int flags=cudaMemAttachGlobal) → intptr_t

```

```python
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

```

```python
cupy.cuda.runtime.mallocArray

cupy.cuda.runtime.mallocArray(intptr_t descPtr, size_t width, size_t height, unsigned int flags=0) → intptr_t

```

```python
cupy.cuda.runtime.hostAlloc

cupy.cuda.runtime.hostAlloc(size_t size, unsigned int flags) → intptr_t

```

```python
cupy.cuda.runtime.hostRegister

cupy.cuda.runtime.hostRegister(intptr_t ptr, size_t size, unsigned int flags)

```

```python
cupy.cuda.runtime.hostUnregister

cupy.cuda.runtime.hostUnregister(intptr_t ptr)

```

```python
cupy.cuda.runtime.free

cupy.cuda.runtime.free(intptr_t ptr)

```

```python
cupy.cuda.runtime.freeHost

cupy.cuda.runtime.freeHost(intptr_t ptr)

```

```python
cupy.cuda.runtime.freeArray

cupy.cuda.runtime.freeArray(intptr_t ptr)

```

```python
cupy.cuda.runtime.memGetInfo

cupy.cuda.runtime.memGetInfo()

```

```python
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.cudadl.runtime.memcpy3DAsync

```python
cupy.cudadl.runtime.memcpy3DAsync(intptr_t Memcpy3DParmsPtr, intptr_t stream)
```

**cupy.cudadl.runtime.memset**

```python
cupy.cudadl.runtime.memset(intptr_t ptr, int value, size_t size)
```

**cupy.cudadl.runtime.memsetAsync**

```python
cupy.cudadl.runtime.memsetAsync(intptr_t ptr, int value, size_t size, intptr_t stream)
```

**cupy.cudadl.runtime.memPrefetchAsync**

```python
cupy.cudadl.runtime.memPrefetchAsync(intptr_t devPtr, size_t count, int dstDevice, intptr_t stream)
```

**cupy.cudadl.runtime.memAdvise**

```python
cupy.cudadl.runtime.memAdvise(intptr_t devPtr, size_t count, int advice, int device)
```

**cupy.cudadl.runtime.pointerGetAttributes**

```python
cupy.cudadl.runtime.pointerGetAttributes(intptr_t ptr) \rightarrow PointerAttributes
```

**cupy.cudadl.runtime.streamCreate**

```python
cupy.cudadl.runtime.streamCreate() \rightarrow intptr_t
```

**cupy.cudadl.runtime.streamCreateWithFlags**

```python
cupy.cudadl.runtime.streamCreateWithFlags(unsigned int flags) \rightarrow intptr_t
```

**cupy.cudadl.runtime.streamDestroy**

```python
cupy.cudadl.runtime.streamDestroy(intptr_t stream)
```

**cupy.cudadl.runtime.streamSynchronize**

```python
cupy.cudadl.runtime.streamSynchronize(intptr_t stream)
```

**cupy.cudadl.runtime.streamAddCallback**

```python
cupy.cudadl.runtime.streamAddCallback(intptr_t stream, callback, intptr_t arg, unsigned int flags=0)
```
cupy.cuda.runtime.streamQuery

cupy.cuda.runtime.streamQuery(stream)

cupy.cuda.runtime.streamWaitEvent

cupy.cuda.runtime.streamWaitEvent(stream, event, flags=0)

cupy.cuda.runtime.eventCreate

cupy.cuda.runtime.eventCreate() \rightarrow intptr_t

cupy.cuda.runtime.eventCreateWithFlags

cupy.cuda.runtime.eventCreateWithFlags(flags) \rightarrow intptr_t

cupy.cuda.runtime.eventDestroy

cupy.cuda.runtime.eventDestroy(event)

cupy.cuda.runtime.eventElapsedTime

cupy.cuda.runtime.eventElapsedTime(start, end) \rightarrow float

cupy.cuda.runtime.eventQuery

cupy.cuda.runtime.eventQuery(event)

cupy.cuda.runtime.eventRecord

cupy.cuda.runtime.eventRecord(event, stream)

cupy.cuda.runtime.eventSynchronize

cupy.cuda.runtime.eventSynchronize(event)

cupy.cuda.runtime.ipcGetMemHandle

cupy.cuda.runtime.ipcGetMemHandle(devPtr)

cupy.cuda.runtime.ipcOpenMemHandle

cupy.cuda.runtime.ipcOpenMemHandle(handle, flags=cudaIpcMemLazyEnablePeerAccess)
cupy.cuda.runtime.ipcCloseMemHandle

```python
cupy.cuda.runtime.ipcCloseMemHandle(intptr_t devPtr)
```

cupy.cuda.runtime.ipcGetEventHandle

```python
cupy.cuda.runtime.ipcGetEventHandle(intptr_t event)
```

cupy.cuda.runtime.ipcOpenEventHandle

```python
cupy.cuda.runtime.ipcOpenEventHandle(bytes handle)
```

### 4.8 Kernel binary memoization

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cupy.memoize</td>
<td>Makes a function memoizing the result for each argument and device.</td>
</tr>
<tr>
<td>cupy.clear_memo</td>
<td>Clears the memoized results for all functions decorated by memoize.</td>
</tr>
</tbody>
</table>

#### 4.8.1 cupy.memoize

```python
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.

#### 4.8.2 cupy.clear_memo

```python
cupy.clear_memo()
```

Clears the memoized results for all functions decorated by memoize.

### 4.9 Custom kernels

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cupy.ElementwiseKernel</td>
<td>User-defined elementwise kernel.</td>
</tr>
<tr>
<td>cupy.ReductionKernel</td>
<td>User-defined reduction kernel.</td>
</tr>
<tr>
<td>cupy.RawKernel</td>
<td>User-defined custom kernel.</td>
</tr>
<tr>
<td>cupy.RawModule</td>
<td>User-defined custom module.</td>
</tr>
<tr>
<td>cupy.fuse</td>
<td>Decorator that fuses a function.</td>
</tr>
</tbody>
</table>
4.9.1 cupy.ElementwiseKernel

```python
class cupy.ElementwiseKernel(in_params, out_params, operation, name=u'kernel', reduce_dims=True, preamble=u'', 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.
- `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.

Attributes

in_params  
kwars  
name  
nargs  
nin  
no_return  
nout  
operation  
out_params  
params  
preamble  
reduce_dims  
return_tuple

4.9.2 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.

• `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.

Returns Arrays are returned according to the `out_params` argument of the `__init__` method.

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`
4.9.3 cupy.RawKernel

class cupy.RawKernel (unicode code, unicode name, tuple options=(), unicode backend=u'nvrtc', bool translate_cucomplex=False, *, bool enable_cooperative_groups=False)

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 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.

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.
Attributes

attributes
Returns a dictionary containing runtime kernel attributes. This is a read-only property; to overwrite the attributes, use

```python
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

cost_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.

### 4.9.4 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)*

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`.
- **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 `cudaLaunchCooperativeKernel` 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.

**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**  
A `RawKernel` instance.

**Return type**  
`RawKernel`

**Note:** The following example shows how to retrieve one of the specialized C++ template kernels:

```python
code = r'''
template<typename T>
__global__ void func(T* in_arr) { /* do something */ }
'''
kers = ('func<int>', 'func<float>', 'func<double>')
mod = cupy.RawModule(code=code, options=('--std=c++11',),
                     name_expressions=kers)

// retrieve func<int>
kint = 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:

```python
# 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
```

(continues on next page)
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

Attributes
backend
code
enable_cooperative_groups
file_path
module
name_expressions
options

4.9.5 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

```python
>>> @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])
```
4.10 Automatic Kernel Parameters Optimizations

**cupyx.optimizing.optimize**

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**

```python
>>> 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.

4.11 Interoperability

CuPy can also be used in conjunction with other frameworks.
4.11.1 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.

```python
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; note that this is currently defined as an experimental feature of NumPy and you need to specify the environment variable (NUMPY_EXPERIMENTAL_ARRAY_FUNCTION=1) to enable it.

4.11.2 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 JITed with Numba. The following is a simple example code borrowed from numba/numba#2860:

```python
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.

```python
import numpy
import numba
import cupy

x = numpy.arange(10)  # type: numpy.ndarray
x_numba = numba.cudarray.to_device(x)  # type: numba.cudarray.deviceIdArray.
# Type: DeviceNDArray
x_cupy = cupy.asarray(x_numba)  # type: cupy.ndarray
```
### 4.11.3 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 Open MPI and MVAPICH, have extended their support beyond the v3.1 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:

```python
# 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.

### 4.11.4 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()`).

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cupy.fromDlpack</code></td>
<td>Zero-copy conversion from a DLPack tensor to a ndarray.</td>
</tr>
</tbody>
</table>

#### `cupy.fromDlpack`

- **Function**: `cupy.fromDlpack(dltensor) -> ndarray`  
- **Description**: Zero-copy conversion from a DLPack tensor to a `ndarray`.

DLPack is an 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`. 

4.11. Interoperability
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).

Example

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

Here is a simple example:

```python
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)
```

Here is an example of converting PyTorch tensor into cupy.ndarray.

```python
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.
cx = cupy.fromDlpack(dx)
# Convert it back to a PyTorch tensor.
tx2 = from_dlpack(cx.toDlpack())
```

4.12 Testing Modules

CuPy offers testing utilities to support unit testing. They are under namespace cupy.testing.
## 4.12.1 Standard Assertions

The assertions have same names as NumPy's ones. The difference from NumPy is that they can accept both `numpy.ndarray` and `cupy.ndarray`.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cupy.testing.assert_allclose</code></td>
<td>Raises an <code>AssertionError</code> if objects are not equal up to desired tolerance.</td>
</tr>
<tr>
<td><code>cupy.testing.assert_array_almost_equal</code></td>
<td>Raises an <code>AssertionError</code> if objects are not equal up to desired precision.</td>
</tr>
<tr>
<td><code>cupy.testing.assert_array_max_ulp</code></td>
<td>Compare two arrays relatively to their spacing.</td>
</tr>
<tr>
<td><code>cupy.testing.assert_array_equal</code></td>
<td>Check that all items of arrays differ in at most N Units in the Last Place.</td>
</tr>
<tr>
<td><code>cupy.testing.assert_array_less</code></td>
<td>Raises an <code>AssertionError</code> if array_like objects are not ordered by less than.</td>
</tr>
<tr>
<td><code>cupy.testing.assert_array_list_equal</code></td>
<td>Compares lists of arrays pairwise with <code>assert_array_equal</code>.</td>
</tr>
</tbody>
</table>

### `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`

`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.
CuPy Documentation, Release 8.2.0

See also:

numpy.testing.assert_array_almost_equal()

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_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()`

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()`

### 4.12.2 NumPy-CuPy Consistency Check

The following decorators are for testing consistency between CuPy’s functions and corresponding NumPy’s ones.

<table>
<thead>
<tr>
<th>Decorator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cupy.testing.numpy_cupy_allclose</code></td>
<td>Decorator that checks NumPy results and CuPy ones are close.</td>
</tr>
<tr>
<td><code>cupy.testing.numpy_cupy_array_almost_equal</code></td>
<td>Decorator that checks NumPy results and CuPy ones are almost equal.</td>
</tr>
<tr>
<td><code>cupy.testing.numpy_cupy_array_almost_equal_nulp</code></td>
<td>Decorator that checks results of NumPy and CuPy are equal w.r.t.</td>
</tr>
<tr>
<td><code>cupy.testing.numpy_cupy_array_max_ulp</code></td>
<td>Decorator that checks results of NumPy and CuPy ones are equal w.r.t.</td>
</tr>
<tr>
<td><code>cupy.testing.numpy_cupy_array_equal</code></td>
<td>Decorator that checks NumPy results and CuPy ones are equal.</td>
</tr>
<tr>
<td><code>cupy.testing.numpy_cupy_array_list_equal</code></td>
<td>Decorator that checks the resulting lists of NumPy and CuPy’s one are equal.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 112 – continued from previous page

<table>
<thead>
<tr>
<th>cupy.testing.numpy_cupy_array_less</th>
<th>Decorator that checks the CuPy result is less than NumPy result.</th>
</tr>
</thead>
</table>

**cupy.testing.numpy_cupy_allclose**

cupy.testing.numpy_cupy_allclose(\(rtol=1e-07, atol=0, err_msg='', verbose=True, name='xp', type_check=True, accept_error=False, sp_name=None, scipy_name=None, contiguous_check=True\))

Decorator that checks NumPy results and CuPy ones are close.

**Parameters**

- 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.
- 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.

```python
>>> 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

```python
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

```python
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 \texttt{cupy.testing.assert_array_almost_equal_nulp()} (except the type of array module) even if \texttt{xp} is \texttt{numpy} or \texttt{cupy}.

\textbf{See also:}

\texttt{cupy.testing.assert_array_almost_equal_nulp()}

\texttt{cupy.testing.numpy_cupy_array_max_ulp}

\texttt{cupy.testing.numpy_cupy_array_max_ulp}(*maxulp=1, \texttt{dtype}=\texttt{None}, \texttt{name}=\texttt{xp'}, \texttt{type_check}=\texttt{True}, \texttt{accept_error}=\texttt{False}, \texttt{sp_name}=\texttt{None}, \texttt{scipy_name}=\texttt{None})

Decorator that checks results of NumPy and CuPy ones are equal w.r.t. ulp.

\textbf{Parameters}

- \texttt{maxulp} (int) – The maximum number of units in the last place that elements of resulting two arrays can differ.
- \texttt{dtype} (\texttt{numpy.dtype}) – Data-type to convert the resulting two array to if given.
- \texttt{name} (str) – Argument name whose value is either \texttt{numpy} or \texttt{cupy} module.
- \texttt{type_check} (bool) – If True, consistency of dtype is also checked.
- \texttt{accept_error} (bool, \texttt{Exception} or tuple of \texttt{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 \texttt{True} all error types are acceptable. If it is \texttt{False} no error is acceptable.
- \texttt{sp_name} (str or None) – Argument name whose value is either \texttt{scipy.sparse} or \texttt{cupyx.scipy.sparse} module. If None, no argument is given for the modules.
- \texttt{scipy_name} (str or None) – Argument name whose value is either \texttt{scipy} or \texttt{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 \texttt{assert_array_max_ulp()} (except the type of array module) even if \texttt{xp} is \texttt{numpy} or \texttt{cupy}.

\textbf{See also:}

\texttt{cupy.testing.assert_array_max_ulp()}

\texttt{cupy.testing.numpy_cupy_array_equal}

\texttt{cupy.testing.numpy_cupy_array_equal}(*err_msg=", \texttt{verbose}=\texttt{True}, \texttt{name}=\texttt{xp'}, \texttt{type_check}=\texttt{True}, \texttt{accept_error}=\texttt{False}, \texttt{sp_name}=\texttt{None}, \texttt{scipy_name}=\texttt{None}, \texttt{strides_check}=\texttt{False})

Decorator that checks NumPy results and CuPy ones are equal.

\textbf{Parameters}

- \texttt{err_msg} (str) – The error message to be printed in case of failure.
- \texttt{verbose} (bool) – If True, the conflicting values are appended to the error message.
- \texttt{name} (str) – Argument name whose value is either \texttt{numpy} or \texttt{cupy} module.
- \texttt{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()

### 4.12.3 Parameterized dtype Test

The following decorators offer the standard way for parameterized test with respect to single or the combination of dtype(s).

- **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.

```python
>>> 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.

```python
>>> 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()

4.12. Testing Modules
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()

4.12.4 Parameterized order Test

The following decorators offer the standard way to parameterize tests with orders.

<table>
<thead>
<tr>
<th>cupy.testing.for_orders</th>
<th>Decorator to parameterize tests with order.</th>
</tr>
</thead>
</table>

Continued on next page
Table 114 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cupy.testing.for_CF_orders</td>
<td>Decorator that checks the fixture with orders ‘C’ and ‘F’.</td>
</tr>
</tbody>
</table>

**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()

## 4.13 Profiling

### 4.13.1 time range

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cupy.prof.TimeRangeDecorator</td>
<td>Decorator to mark function calls with range in NVIDIA profiler</td>
</tr>
<tr>
<td>cupy.prof.time_range</td>
<td>A context manager to describe the enclosed block as a nested range</td>
</tr>
</tbody>
</table>

**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.

```python
>>> 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

```python
def __call__(func):
    Call self as a function.

def __enter__():

def __exit__(exc_type, exc_value, traceback):
```

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

```python
>>> 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()`

4.13.2 Device synchronization detection

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cupyx.allow_synchronize</code></td>
<td>Allows or disallows device synchronization temporarily in the current thread.</td>
</tr>
<tr>
<td><code>cupyx.DeviceSynchronized</code></td>
<td>Raised when device synchronization is detected while disallowed.</td>
</tr>
</tbody>
</table>
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()`

### 4.14 Environment variables

Here are the environment variables CuPy uses.

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA_PATH</td>
<td>Path to the directory containing CUDA. The parent of the directory containing <code>nvcc</code> is used as default. When <code>nvcc</code> is not found, <code>/usr/local/cuda</code> is used. See <a href="#">Working with Custom CUDA Installation</a> for details.</td>
</tr>
<tr>
<td>CUPY_CACHE_DIR</td>
<td>Path to the directory to store kernel cache. <code>~/.cupy/kernel_cache</code> is used by default. See <a href="#">Overview</a> for details.</td>
</tr>
<tr>
<td>CUPY_CACHE_SAVE_CUDA_SOURCE</td>
<td>If set to 1, CUDA source file will be saved along with compiled binary in the cache directory for debug purpose. It is disabled by default. Note: source file will not be saved if the compiled binary is already stored in the cache.</td>
</tr>
<tr>
<td>CUPY_CACHE_IN_MEMORY</td>
<td>If set in <code>CUPY_CACHE_DIR</code> (and its default) and <code>CUPY_CACHE_SAVE_CUDA_SOURCE</code> will be ignored, and the cache is in memory. This env var allows reducing disk I/O, but is ignored when <code>nvcc</code> is set to be the compiler backend.</td>
</tr>
<tr>
<td>CUPY_DUMP_CUDA_SOURCE</td>
<td>If set in <code>CUPY_CACHE_DIR</code> (and its default) and <code>CUPY_CACHE_SAVE_CUDA_SOURCE</code>, if CUDA kernel compilation fails, CuPy dumps CUDA kernel code to standard error. It is disabled by default.</td>
</tr>
<tr>
<td>CUPY_CUDA_COMPILE_WITH_DEBUG</td>
<td>If set to 1, CUDA kernel will be compiled with debug information (<code>--device-debug</code> and <code>--generate-line-info</code>). It is disabled by default.</td>
</tr>
<tr>
<td>CUPY_GPU_MEMORY_LIMIT</td>
<td>The amount of memory that can be allocated for each device. The value can be specified in absolute bytes or fraction (e.g., &quot;90%&quot;) of the total memory of each GPU. See <a href="#">Memory Management</a> for details. 0 (unlimited) is used by default.</td>
</tr>
<tr>
<td>CUPY_SEED</td>
<td>Set the seed for random number generators.</td>
</tr>
<tr>
<td>CUPY_EXPERIMENTAL_SLICE_COPY</td>
<td>If set to 1, the following syntax is enabled: <code>cupy.ndarray[:]=numpy.ndarray</code>.</td>
</tr>
<tr>
<td>CUPY_ACCELERATORS</td>
<td>A comma-separated string of backend names (cub or cutensor) which indicates the acceleration backends used in CuPy operations and its priority. Default is empty string (all accelerators are disabled).</td>
</tr>
<tr>
<td>CUPY_TF32</td>
<td>If set to 1, it allows CUDA libraries to use Tensor Cores TF32 compute for 32-bit floating point compute. The default is 0 and TF32 is not used.</td>
</tr>
<tr>
<td>NVCC</td>
<td>Define the compiler to use when compiling CUDA source. Note that most CuPy kernels are built with NVRTC; this environment is only effective for RawKernels/RawModules with <code>nvcc</code> backend or when using cub as the accelerator.</td>
</tr>
</tbody>
</table>

Moreover, as in any CUDA programs, all of the CUDA environment variables listed in the [CUDA Toolkit Documentation](#) will also be honored. 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++.

### 4.14.1 For installation

These environment variables are used during installation (building CuPy from source).

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA_PATH</td>
<td>See the description above.</td>
</tr>
<tr>
<td>CUTENSOR_PATH</td>
<td>Path to the cuTENSOR root directory that contains lib and include directories. (experimental)</td>
</tr>
<tr>
<td>NVCC</td>
<td>Define the compiler to use when compiling CUDA files.</td>
</tr>
<tr>
<td>CUPY_PYTHON_350_FORCE</td>
<td>Enforce CuPy to be installed against Python 3.5.0 (not recommended).</td>
</tr>
<tr>
<td>CUPY_INSTALL_USE_HIP</td>
<td>For building the ROCm support, see Building CuPy for ROCm for further detail.</td>
</tr>
<tr>
<td>CUPY_NVCC_GENERATE_CODE</td>
<td>To build CuPy for a particular CUDA architecture. For example, CUPY_NVCC_GENERATE_CODE=&quot;arch=compute_60,code=sm_60&quot;. For specifying multiple archs, concatenate the arch=... strings with semicolons (;). When this is not set, the default is to support all architectures.</td>
</tr>
</tbody>
</table>

### 4.15 Difference between CuPy and NumPy

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

#### 4.15.1 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:

```python
>>> 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)
```

#### 4.15.2 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`.

```python
>>> np.random.randn(dtype=np.float32)
Traceback (most recent call last):
  File "<stdin>", line i, in <module>
TypeError: randn() got an unexpected keyword argument 'dtype'
>>> cupy.random.randn(dtype=np.float32) # doctest: +SKIP
array(0.10689262300729752, dtype=float32)
```
4.15.3 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.

```python
>>> 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])
```

4.15.4 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.

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

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

```python
>>> 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.])
```

4.15.5 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.

```python
>>> 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.

```python
>>> (np.array(3, dtype=np.int32) * np.array([1., 2.], dtype=np.float32)).dtype
dtype('float32')
>>> (np.array(300000, 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')
```

4.15.6 Data types

Data type of CuPy arrays cannot be non-numeric like strings and objects. See Overview for details.

4.15.7 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`).

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

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

4.15.8 Random seed arrays are hashed to scalars

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

```python
>>> 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.

4.16 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.
### 4.16.1 NumPy / CuPy APIs

#### Module-Level

<table>
<thead>
<tr>
<th>NumPy</th>
<th>CuPy</th>
</tr>
</thead>
<tbody>
<tr>
<td>numpy.abs</td>
<td>cupy.abs</td>
</tr>
<tr>
<td>numpy.absolute</td>
<td>cupy.absolute</td>
</tr>
<tr>
<td>numpy.add</td>
<td>cupy.add</td>
</tr>
<tr>
<td>numpy.add_docstring</td>
<td>-</td>
</tr>
<tr>
<td>numpy.add_newdoc</td>
<td>-</td>
</tr>
<tr>
<td>numpy.add_newdoc_ufunc</td>
<td>-</td>
</tr>
<tr>
<td>numpy.alen</td>
<td>-</td>
</tr>
<tr>
<td>numpy.all</td>
<td>cupy.all</td>
</tr>
<tr>
<td>numpy.allclose</td>
<td>cupy.allclose</td>
</tr>
<tr>
<td>numpy.alltrue</td>
<td>-</td>
</tr>
<tr>
<td>numpy.amax</td>
<td>cupy.amax</td>
</tr>
<tr>
<td>numpy.amin</td>
<td>cupy.amin</td>
</tr>
<tr>
<td>numpy.angle</td>
<td>cupy.angle</td>
</tr>
<tr>
<td>numpy.any</td>
<td>cupy.any</td>
</tr>
<tr>
<td>numpy.append</td>
<td>-</td>
</tr>
<tr>
<td>numpy.apply_along_axis</td>
<td>-</td>
</tr>
<tr>
<td>numpy.apply_over_axes</td>
<td>-</td>
</tr>
<tr>
<td>numpy.arange</td>
<td>cupy.arange</td>
</tr>
<tr>
<td>numpy.arccos</td>
<td>cupy.arccos</td>
</tr>
<tr>
<td>numpy.arccosh</td>
<td>cupy.arccosh</td>
</tr>
<tr>
<td>numpy.arcsin</td>
<td>cupy.arcsin</td>
</tr>
<tr>
<td>numpy.arcsinh</td>
<td>cupy.arcsinh</td>
</tr>
<tr>
<td>numpy.arctan</td>
<td>cupy.arctan</td>
</tr>
<tr>
<td>numpy.arctan2</td>
<td>cupy.arctan2</td>
</tr>
<tr>
<td>numpy.arctanh</td>
<td>cupy.arctanh</td>
</tr>
<tr>
<td>numpy.argmax</td>
<td>cupy.argmax</td>
</tr>
<tr>
<td>numpy.argmin</td>
<td>cupy.argmin</td>
</tr>
<tr>
<td>numpy.argpartition</td>
<td>cupy.argpartition</td>
</tr>
<tr>
<td>numpy.argsort</td>
<td>cupy.argsort</td>
</tr>
<tr>
<td>numpy.argwhere</td>
<td>cupy.argwhere</td>
</tr>
<tr>
<td>numpy.around</td>
<td>cupy.around</td>
</tr>
<tr>
<td>numpy.array</td>
<td>cupy.array</td>
</tr>
<tr>
<td>numpy.array2string</td>
<td>-</td>
</tr>
<tr>
<td>numpy.array_equal</td>
<td>cupy.array_equal</td>
</tr>
<tr>
<td>numpy.array_equiv</td>
<td>-</td>
</tr>
<tr>
<td>numpy.array_repr</td>
<td>cupy.array_repr</td>
</tr>
<tr>
<td>numpy.array_split</td>
<td>cupy.array_split</td>
</tr>
<tr>
<td>numpy.array_str</td>
<td>cupy.array_str</td>
</tr>
<tr>
<td>numpy.asanyarray</td>
<td>cupy.asanyarray</td>
</tr>
<tr>
<td>numpy.asarray</td>
<td>cupy.asarray</td>
</tr>
<tr>
<td>numpy.asarray_chkfinite</td>
<td>-</td>
</tr>
<tr>
<td>numpy.ascontiguousarray</td>
<td>cupy.ascontiguousarray</td>
</tr>
<tr>
<td>numpy.asfarray</td>
<td>-</td>
</tr>
<tr>
<td>numpy.asfortranarray</td>
<td>cupy.asfortranarray</td>
</tr>
<tr>
<td>numpy.asmatrix</td>
<td>-</td>
</tr>
</tbody>
</table>
Table 117 – continued from previous page

<table>
<thead>
<tr>
<th>NumPy</th>
<th>CuPy</th>
</tr>
</thead>
<tbody>
<tr>
<td>numpy.asscalar</td>
<td>-</td>
</tr>
<tr>
<td>numpy.atleast_1d</td>
<td>cupy.atleast_1d</td>
</tr>
<tr>
<td>numpy.atleast_2d</td>
<td>cupy.atleast_2d</td>
</tr>
<tr>
<td>numpy.atleast_3d</td>
<td>cupy.atleast_3d</td>
</tr>
<tr>
<td>numpy.average</td>
<td>cupy.average</td>
</tr>
<tr>
<td>numpy.bartlett</td>
<td>cupy.bartlett</td>
</tr>
<tr>
<td>numpy.base_repr</td>
<td>cupy.base_repr</td>
</tr>
<tr>
<td>numpy.binary_repr</td>
<td>cupy.binary_repr</td>
</tr>
<tr>
<td>numpy.bincount</td>
<td>cupy.bincount</td>
</tr>
<tr>
<td>numpy.bitwise_and</td>
<td>cupy.bitwise_and</td>
</tr>
<tr>
<td>numpy.bitwise_not</td>
<td>cupy.bitwise_not</td>
</tr>
<tr>
<td>numpy.bitwise_or</td>
<td>cupy.bitwise_or</td>
</tr>
<tr>
<td>numpy.bitwise_xor</td>
<td>cupy.bitwise_xor</td>
</tr>
<tr>
<td>numpy.blackman</td>
<td>cupy.blackman</td>
</tr>
<tr>
<td>numpy.block</td>
<td>-</td>
</tr>
<tr>
<td>numpy.bmat</td>
<td>-</td>
</tr>
<tr>
<td>numpy.broadcast_arrays</td>
<td>cupy.broadcast_arrays</td>
</tr>
<tr>
<td>numpy.broadcast_to</td>
<td>cupy.broadcast_to</td>
</tr>
<tr>
<td>numpy.busday_count</td>
<td>-</td>
</tr>
<tr>
<td>numpy.busday_offset</td>
<td>-</td>
</tr>
<tr>
<td>numpy.byte_bounds</td>
<td>-</td>
</tr>
<tr>
<td>numpy.can_cast</td>
<td>cupy.can_cast</td>
</tr>
<tr>
<td>numpy.cbrt</td>
<td>cupy.cbrt</td>
</tr>
<tr>
<td>numpy.cell</td>
<td>cupy.cell</td>
</tr>
<tr>
<td>numpy.choose</td>
<td>cupy.choose</td>
</tr>
<tr>
<td>numpy.clip</td>
<td>cupy.clip</td>
</tr>
<tr>
<td>numpy.column_stack</td>
<td>cupy.column_stack</td>
</tr>
<tr>
<td>numpy.common_type</td>
<td>cupy.common_type</td>
</tr>
<tr>
<td>numpy.compare_chararrays</td>
<td>-</td>
</tr>
<tr>
<td>numpy.compress</td>
<td>cupy.compress</td>
</tr>
<tr>
<td>numpy.concatenate</td>
<td>cupy.concatenate</td>
</tr>
<tr>
<td>numpy.conj</td>
<td>cupy.conj</td>
</tr>
<tr>
<td>numpy.conjugate</td>
<td>cupy.conjugate</td>
</tr>
<tr>
<td>numpy.convolve</td>
<td>cupy.convolve</td>
</tr>
<tr>
<td>numpy.copy</td>
<td>cupy.copy</td>
</tr>
<tr>
<td>numpy.copysign</td>
<td>cupy.copysign</td>
</tr>
<tr>
<td>numpy.copyto</td>
<td>cupy.copyto</td>
</tr>
<tr>
<td>numpy.corrcoef</td>
<td>cupy.corrcoef</td>
</tr>
<tr>
<td>numpy.correlate</td>
<td>cupy.correlate</td>
</tr>
<tr>
<td>numpy.cos</td>
<td>cupy.cos</td>
</tr>
<tr>
<td>numpy.cosh</td>
<td>cupy.cosh</td>
</tr>
<tr>
<td>numpy.count_nonzero</td>
<td>cupy.count_nonzero</td>
</tr>
<tr>
<td>numpy.cov</td>
<td>cupy.cov</td>
</tr>
<tr>
<td>numpy.cross</td>
<td>cupy.cross</td>
</tr>
<tr>
<td>numpy.cumprod</td>
<td>cupy.cumprod</td>
</tr>
<tr>
<td>numpy.cumproduct</td>
<td>-</td>
</tr>
<tr>
<td>numpy.cumsum</td>
<td>cupy.cumsum</td>
</tr>
<tr>
<td>numpy.datetime_as_string</td>
<td>-</td>
</tr>
<tr>
<td>numpy.datetime_data</td>
<td>-</td>
</tr>
</tbody>
</table>

Continued on next page
Table 117 – continued from previous page

<table>
<thead>
<tr>
<th>NumPy</th>
<th>CuPy</th>
</tr>
</thead>
<tbody>
<tr>
<td>numpy.deg2rad</td>
<td>cupy.deg2rad</td>
</tr>
<tr>
<td>numpy.degrees</td>
<td>cupy.degrees</td>
</tr>
<tr>
<td>numpy.delete</td>
<td>-</td>
</tr>
<tr>
<td>numpy.deprecate</td>
<td>-</td>
</tr>
<tr>
<td>numpy.deprecate_with_doc</td>
<td>-</td>
</tr>
<tr>
<td>numpy.diag</td>
<td>cupy.diag</td>
</tr>
<tr>
<td>numpy.diag_indices</td>
<td>cupy.diag_indices</td>
</tr>
<tr>
<td>numpy.diag_indices_from</td>
<td>cupy.diag_indices_from</td>
</tr>
<tr>
<td>numpy.diagflat</td>
<td>cupy.diagflat</td>
</tr>
<tr>
<td>numpy.diagonal</td>
<td>cupy.diagonal</td>
</tr>
<tr>
<td>numpy.diff</td>
<td>cupy.diff</td>
</tr>
<tr>
<td>numpy-digitize</td>
<td>cupy-digitize</td>
</tr>
<tr>
<td>numpy.disp</td>
<td>-</td>
</tr>
<tr>
<td>numpy.divide</td>
<td>cupy.divide</td>
</tr>
<tr>
<td>numpy.divmod</td>
<td>cupy.divmod</td>
</tr>
<tr>
<td>numpy.dot</td>
<td>cupy.dot</td>
</tr>
<tr>
<td>numpy.dsplit</td>
<td>cupy.dsplit</td>
</tr>
<tr>
<td>numpy.dstack</td>
<td>cupy.dstack</td>
</tr>
<tr>
<td>numpy.ediff1d</td>
<td>-</td>
</tr>
<tr>
<td>numpy.einsum</td>
<td>cupy.einsum</td>
</tr>
<tr>
<td>numpy.einsum_path</td>
<td>-</td>
</tr>
<tr>
<td>numpy.empty</td>
<td>cupy.empty</td>
</tr>
<tr>
<td>numpy.empty_like</td>
<td>cupy.empty_like</td>
</tr>
<tr>
<td>numpy.equal</td>
<td>cupy.equal</td>
</tr>
<tr>
<td>numpy.exp</td>
<td>cupy.exp</td>
</tr>
<tr>
<td>numpy.exp2</td>
<td>cupy.exp2</td>
</tr>
<tr>
<td>numpy.expand_dims</td>
<td>cupy.expand_dims</td>
</tr>
<tr>
<td>numpy.expm1</td>
<td>cupy.expm1</td>
</tr>
<tr>
<td>numpy.extract</td>
<td>cupy.extract</td>
</tr>
<tr>
<td>numpy.eye</td>
<td>cupy.eye</td>
</tr>
<tr>
<td>numpy.fabs</td>
<td>-</td>
</tr>
<tr>
<td>numpy.fastCopyAndTranspose</td>
<td>-</td>
</tr>
<tr>
<td>numpy.fill_diagonal</td>
<td>cupy.fill_diagonal</td>
</tr>
<tr>
<td>numpy.find_common_type</td>
<td>cupy.find_common_type (alias of numpy.find_common_type)</td>
</tr>
<tr>
<td>numpy.fix</td>
<td>cupy.fix</td>
</tr>
<tr>
<td>numpy.flatnonzero</td>
<td>cupy.flatnonzero</td>
</tr>
<tr>
<td>numpy.flip</td>
<td>cupy.flip</td>
</tr>
<tr>
<td>numpy.fflplr</td>
<td>cupy.fflplr</td>
</tr>
<tr>
<td>numpy.flipud</td>
<td>cupy.flipud</td>
</tr>
<tr>
<td>numpy.float_power</td>
<td>-</td>
</tr>
<tr>
<td>numpy.floor</td>
<td>cupy.floor</td>
</tr>
<tr>
<td>numpy.floor_divide</td>
<td>cupy.floor_divide</td>
</tr>
<tr>
<td>numpy.fmax</td>
<td>cupy.fmax</td>
</tr>
<tr>
<td>numpy.fmin</td>
<td>cupy.fmin</td>
</tr>
<tr>
<td>numpy.fmod</td>
<td>cupy.fmod</td>
</tr>
<tr>
<td>numpy.format_float_positional</td>
<td>-</td>
</tr>
<tr>
<td>numpy.format_float_scientific</td>
<td>-</td>
</tr>
<tr>
<td>numpy.frexp</td>
<td>cupy.frexp</td>
</tr>
<tr>
<td>numpy.frombuffer</td>
<td>-</td>
</tr>
</tbody>
</table>

Continued on next page
Table 117 – continued from previous page

<table>
<thead>
<tr>
<th>NumPy</th>
<th>CuPy</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>numpy.fromfile</code></td>
<td><code>cupy.fromfile</code></td>
</tr>
<tr>
<td><code>numpy.fromfunction</code></td>
<td>-</td>
</tr>
<tr>
<td><code>numpy.fromiter</code></td>
<td>-</td>
</tr>
<tr>
<td><code>numpy.frompyfunc</code></td>
<td>-</td>
</tr>
<tr>
<td><code>numpy.fromregex</code></td>
<td>-</td>
</tr>
<tr>
<td><code>numpy.fromstring</code></td>
<td>-</td>
</tr>
<tr>
<td><code>numpy.full</code></td>
<td><code>cupy.full</code></td>
</tr>
<tr>
<td><code>numpy.full_like</code></td>
<td><code>cupy.full_like</code></td>
</tr>
<tr>
<td><code>numpy.fv</code></td>
<td>-</td>
</tr>
<tr>
<td><code>numpy.gcd</code></td>
<td><code>cupy.gcd</code></td>
</tr>
<tr>
<td><code>numpy.genfromtxt</code></td>
<td>-</td>
</tr>
<tr>
<td><code>numpy.geomspace</code></td>
<td>-</td>
</tr>
<tr>
<td><code>numpy.get_array_wrap</code></td>
<td>-</td>
</tr>
<tr>
<td><code>numpy.get_include</code></td>
<td>-</td>
</tr>
<tr>
<td><code>numpy.get_printoptions</code></td>
<td>-</td>
</tr>
<tr>
<td><code>numpy.getbufsize</code></td>
<td>-</td>
</tr>
<tr>
<td><code>numpy.geterr</code></td>
<td>-</td>
</tr>
<tr>
<td><code>numpy.geterrcall</code></td>
<td>-</td>
</tr>
<tr>
<td><code>numpy.geterrobj</code></td>
<td>-</td>
</tr>
<tr>
<td><code>numpy.gradient</code></td>
<td>-</td>
</tr>
<tr>
<td><code>numpy.greater</code></td>
<td><code>cupy.greater</code></td>
</tr>
<tr>
<td><code>numpy.greater_equal</code></td>
<td><code>cupy.greater_equal</code></td>
</tr>
<tr>
<td><code>numpy.hamming</code></td>
<td><code>cupy.hamming</code></td>
</tr>
<tr>
<td><code>numpy.hanning</code></td>
<td><code>cupy.hanning</code></td>
</tr>
<tr>
<td><code>numpy.heaviside</code></td>
<td>-</td>
</tr>
<tr>
<td><code>numpy.histogram</code></td>
<td><code>cupy.histogram</code></td>
</tr>
<tr>
<td><code>numpy.histogram2d</code></td>
<td>-</td>
</tr>
<tr>
<td><code>numpy.histogram_bin_edges</code></td>
<td>-</td>
</tr>
<tr>
<td><code>numpy.histogramdd</code></td>
<td>-</td>
</tr>
<tr>
<td><code>numpy.hsplit</code></td>
<td><code>cupy.hsplit</code></td>
</tr>
<tr>
<td><code>numpy.hstack</code></td>
<td><code>cupy.hstack</code></td>
</tr>
<tr>
<td><code>numpy.hypot</code></td>
<td><code>cupy.hypot</code></td>
</tr>
<tr>
<td><code>numpy.i0</code></td>
<td><code>cupy.i0</code></td>
</tr>
<tr>
<td><code>numpy.identity</code></td>
<td><code>cupy.identity</code></td>
</tr>
<tr>
<td><code>numpy.imag</code></td>
<td><code>cupy.imag</code></td>
</tr>
<tr>
<td><code>numpy.in1d</code></td>
<td><code>cupy.in1d</code></td>
</tr>
<tr>
<td><code>numpy.indices</code></td>
<td><code>cupy.indices</code></td>
</tr>
<tr>
<td><code>numpy.info</code></td>
<td>-</td>
</tr>
<tr>
<td><code>numpy.insert</code></td>
<td>-</td>
</tr>
<tr>
<td><code>numpy.interp</code></td>
<td>-</td>
</tr>
<tr>
<td><code>numpy.intersect1d</code></td>
<td>-</td>
</tr>
<tr>
<td><code>numpy.ipmt</code></td>
<td><code>cupy.ipmt</code></td>
</tr>
<tr>
<td><code>numpy.izone</code></td>
<td>-</td>
</tr>
<tr>
<td><code>numpy.issbusday</code></td>
<td>-</td>
</tr>
<tr>
<td><code>numpy.isclose</code></td>
<td><code>cupy.isclose</code></td>
</tr>
<tr>
<td><code>numpy.iscomplex</code></td>
<td><code>cupy.iscomplex</code></td>
</tr>
<tr>
<td><code>numpy.iscomplexobj</code></td>
<td><code>cupy.iscomplexobj</code></td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>NumPy</th>
<th>CuPy</th>
</tr>
</thead>
<tbody>
<tr>
<td>numpy.isfinite</td>
<td>cupy.isfinite</td>
</tr>
<tr>
<td>numpy.isfortran</td>
<td>cupy.isfortran</td>
</tr>
<tr>
<td>numpy.isin</td>
<td>cupy.isin</td>
</tr>
<tr>
<td>numpy.isinf</td>
<td>cupy.isinf</td>
</tr>
<tr>
<td>numpy.isnan</td>
<td>cupy.isnan</td>
</tr>
<tr>
<td>numpy.isnat</td>
<td>-</td>
</tr>
<tr>
<td>numpy.isneginf</td>
<td>-</td>
</tr>
<tr>
<td>numpy.isposinf</td>
<td>-</td>
</tr>
<tr>
<td>numpy.isreal</td>
<td>cupy.isreal</td>
</tr>
<tr>
<td>numpy.isrealobj</td>
<td>cupy.isrealobj</td>
</tr>
<tr>
<td>numpy.isscalar</td>
<td>cupy.isscalar</td>
</tr>
<tr>
<td>numpy.issctype</td>
<td>cupy.issctype (alias of numpy.issctype)</td>
</tr>
<tr>
<td>numpy.issubclass_</td>
<td>cupy.issubclass_ (alias of numpy.issubclass_)</td>
</tr>
<tr>
<td>numpy.issubdtype</td>
<td>cupy.issubdtype (alias of numpy.issubdtype)</td>
</tr>
<tr>
<td>numpy.issubsctype</td>
<td>cupy.issubsctype (alias of numpy.issubsctype)</td>
</tr>
<tr>
<td>numpy.iterable</td>
<td>-</td>
</tr>
<tr>
<td>numpy.ix_</td>
<td>cupy.ix_</td>
</tr>
<tr>
<td>numpy.kaiser</td>
<td>cupy.kaiser</td>
</tr>
<tr>
<td>numpy.kron</td>
<td>cupy.kron</td>
</tr>
<tr>
<td>numpy.lcm</td>
<td>cupy.lcm</td>
</tr>
<tr>
<td>numpy.ldexp</td>
<td>cupy.ldexp</td>
</tr>
<tr>
<td>numpy.left_shift</td>
<td>cupy.left_shift</td>
</tr>
<tr>
<td>numpy.less</td>
<td>cupy.less</td>
</tr>
<tr>
<td>numpy.less_equal</td>
<td>cupy.less_equal</td>
</tr>
<tr>
<td>numpy.lexsort</td>
<td>cupy.lexsort</td>
</tr>
<tr>
<td>numpy.linspace</td>
<td>cupy.linspace</td>
</tr>
<tr>
<td>numpy.load</td>
<td>cupy.load</td>
</tr>
<tr>
<td>numpy.loads</td>
<td>-</td>
</tr>
<tr>
<td>numpy.loadtxt</td>
<td>-</td>
</tr>
<tr>
<td>numpy.log</td>
<td>cupy.log</td>
</tr>
<tr>
<td>numpy.log10</td>
<td>cupy.log10</td>
</tr>
<tr>
<td>numpy.log1p</td>
<td>cupy.log1p</td>
</tr>
<tr>
<td>numpy.log2</td>
<td>cupy.log2</td>
</tr>
<tr>
<td>numpy.logaddexp</td>
<td>cupy.logaddexp</td>
</tr>
<tr>
<td>numpy.logaddexp2</td>
<td>cupy.logaddexp2</td>
</tr>
<tr>
<td>numpy.logical_and</td>
<td>cupy.logical_and</td>
</tr>
<tr>
<td>numpy.logical_not</td>
<td>cupy.logical_not</td>
</tr>
<tr>
<td>numpy.logical_or</td>
<td>cupy.logical_or</td>
</tr>
<tr>
<td>numpy.logical_xor</td>
<td>cupy.logical_xor</td>
</tr>
<tr>
<td>numpy.logspace</td>
<td>cupy.logspace</td>
</tr>
<tr>
<td>numpy.lookfor</td>
<td>-</td>
</tr>
<tr>
<td>numpy.mafromtxt</td>
<td>-</td>
</tr>
<tr>
<td>numpy.mask_indices</td>
<td>-</td>
</tr>
<tr>
<td>numpy.mat</td>
<td>-</td>
</tr>
<tr>
<td>numpy.matmul</td>
<td>cupy.matmul</td>
</tr>
<tr>
<td>numpy.max</td>
<td>cupy.max</td>
</tr>
<tr>
<td>numpy.maximum</td>
<td>cupy.maximum</td>
</tr>
<tr>
<td>numpy.maximum_sctype</td>
<td>-</td>
</tr>
<tr>
<td>numpy.may_share_memory</td>
<td>cupy.may_share_memory</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>NumPy</th>
<th>CuPy</th>
</tr>
</thead>
<tbody>
<tr>
<td>numpy.mean</td>
<td>cupy.mean</td>
</tr>
<tr>
<td>numpy.median</td>
<td>cupy.median</td>
</tr>
<tr>
<td>numpy.meshgrid</td>
<td>cupy.meshgrid</td>
</tr>
<tr>
<td>numpy.min</td>
<td>cupy.min</td>
</tr>
<tr>
<td>numpy.min scalar type</td>
<td>cupy.min scalar type (alias of numpy.min scalar type)</td>
</tr>
<tr>
<td>numpy.minimum</td>
<td>cupy.minimum</td>
</tr>
<tr>
<td>numpy.mintypecode</td>
<td>cupy.mintypecode (alias of numpy.mintypecode)</td>
</tr>
<tr>
<td>numpy.mirr</td>
<td>-</td>
</tr>
<tr>
<td>numpy.mod</td>
<td>cupy.mod</td>
</tr>
<tr>
<td>numpy.modf</td>
<td>cupy.modf</td>
</tr>
<tr>
<td>numpy.moveaxis</td>
<td>cupy.moveaxis</td>
</tr>
<tr>
<td>numpy.msort</td>
<td>cupy.msort</td>
</tr>
<tr>
<td>numpy.multiply</td>
<td>cupy.multiply</td>
</tr>
<tr>
<td>numpy.nan_to_num</td>
<td>cupy.nan_to_num</td>
</tr>
<tr>
<td>numpy.nanargmax</td>
<td>cupy.nanargmax</td>
</tr>
<tr>
<td>numpy.nanargmin</td>
<td>cupy.nanargmin</td>
</tr>
<tr>
<td>numpy.nancumprod</td>
<td>-</td>
</tr>
<tr>
<td>numpy.nancumsum</td>
<td>-</td>
</tr>
<tr>
<td>numpy.nanmax</td>
<td>cupy.nanmax</td>
</tr>
<tr>
<td>numpy.nanmean</td>
<td>cupy.nanmean</td>
</tr>
<tr>
<td>numpy.nanmedian</td>
<td>-</td>
</tr>
<tr>
<td>numpy.nanmin</td>
<td>cupy.nanmin</td>
</tr>
<tr>
<td>numpy.nanpercentile</td>
<td>-</td>
</tr>
<tr>
<td>numpy.nanprod</td>
<td>cupy.nanprod</td>
</tr>
<tr>
<td>numpy.nanquantile</td>
<td>-</td>
</tr>
<tr>
<td>numpy.nanstd</td>
<td>cupy.nanstd</td>
</tr>
<tr>
<td>numpy.nansum</td>
<td>cupy.nansum</td>
</tr>
<tr>
<td>numpy.nanvar</td>
<td>cupy.nanvar</td>
</tr>
<tr>
<td>numpy.ndfromtxt</td>
<td>-</td>
</tr>
<tr>
<td>numpy.ndim</td>
<td>cupy.ndim</td>
</tr>
<tr>
<td>numpy.negative</td>
<td>cupy.negative</td>
</tr>
<tr>
<td>numpy.nested_iters</td>
<td>-</td>
</tr>
<tr>
<td>numpy.nextafter</td>
<td>cupy.nextafter</td>
</tr>
<tr>
<td>numpy.nonzero</td>
<td>cupy.nonzero</td>
</tr>
<tr>
<td>numpy.not_equal</td>
<td>cupy.not_equal</td>
</tr>
<tr>
<td>numpy.nper</td>
<td>-</td>
</tr>
<tr>
<td>numpy.npv</td>
<td>-</td>
</tr>
<tr>
<td>numpy.obj2sctype</td>
<td>cupy.obj2sctype (alias of numpy.obj2sctype)</td>
</tr>
<tr>
<td>numpy.ones</td>
<td>cupy.ones</td>
</tr>
<tr>
<td>numpy.ones_like</td>
<td>cupy.ones_like</td>
</tr>
<tr>
<td>numpy.outer</td>
<td>cupy.outer</td>
</tr>
<tr>
<td>numpy.packbits</td>
<td>cupy.packbits</td>
</tr>
<tr>
<td>numpy.pad</td>
<td>cupy.pad</td>
</tr>
<tr>
<td>numpy.partition</td>
<td>cupy.partition</td>
</tr>
<tr>
<td>numpy.percentile</td>
<td>cupy.percentile</td>
</tr>
<tr>
<td>numpy.piecewise</td>
<td>cupy.piecewise</td>
</tr>
<tr>
<td>numpy.place</td>
<td>cupy.place</td>
</tr>
<tr>
<td>numpy.pmt</td>
<td>-</td>
</tr>
<tr>
<td>numpy.poly</td>
<td>-</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>NumPy</th>
<th>CuPy</th>
</tr>
</thead>
<tbody>
<tr>
<td>numpy.polyadd</td>
<td>cupy.polyadd</td>
</tr>
<tr>
<td>numpy.polyder</td>
<td>-</td>
</tr>
<tr>
<td>numpy.polydiv</td>
<td>-</td>
</tr>
<tr>
<td>numpy.polyfit</td>
<td>-</td>
</tr>
<tr>
<td>numpy.polyint</td>
<td>-</td>
</tr>
<tr>
<td>numpy.polymul</td>
<td>cupy.polymul</td>
</tr>
<tr>
<td>numpy.polysub</td>
<td>cupy.polysub</td>
</tr>
<tr>
<td>numpy.polyval</td>
<td>cupy.polyval</td>
</tr>
<tr>
<td>numpy.positive</td>
<td>-</td>
</tr>
<tr>
<td>numpy.power</td>
<td>cupy.power</td>
</tr>
<tr>
<td>numpy.ppmt</td>
<td>-</td>
</tr>
<tr>
<td>numpy.printoptions</td>
<td>-</td>
</tr>
<tr>
<td>numpy.prod</td>
<td>cupy.prod</td>
</tr>
<tr>
<td>numpy.product</td>
<td>-</td>
</tr>
<tr>
<td>numpy.promote_types</td>
<td>cupy.promote_types</td>
</tr>
<tr>
<td>-</td>
<td>(alias of numpy.promote_types)</td>
</tr>
<tr>
<td>numpy.ptp</td>
<td>cupy.ptp</td>
</tr>
<tr>
<td>numpy.put</td>
<td>cupy.put</td>
</tr>
<tr>
<td>numpy.put_along_axis</td>
<td>-</td>
</tr>
<tr>
<td>numpy.putmask</td>
<td>cupy.putmask</td>
</tr>
<tr>
<td>numpy.pv</td>
<td>-</td>
</tr>
<tr>
<td>numpy.quantile</td>
<td>-</td>
</tr>
<tr>
<td>numpy.rad2deg</td>
<td>cupy.rad2deg</td>
</tr>
<tr>
<td>numpy.radians</td>
<td>cupy.radians</td>
</tr>
<tr>
<td>numpy.rate</td>
<td>-</td>
</tr>
<tr>
<td>numpy.ravel</td>
<td>cupy.ravel</td>
</tr>
<tr>
<td>numpy.ravel_multi_index</td>
<td>cupy.ravel_multi_index</td>
</tr>
<tr>
<td>numpy.real</td>
<td>cupy.real</td>
</tr>
<tr>
<td>numpy.real_if_close</td>
<td>-</td>
</tr>
<tr>
<td>numpy.recfromcsv</td>
<td>-</td>
</tr>
<tr>
<td>numpy.recfromtxt</td>
<td>-</td>
</tr>
<tr>
<td>numpy.reciprocal</td>
<td>cupy.reciprocal</td>
</tr>
<tr>
<td>numpy.remainder</td>
<td>cupy.remainder</td>
</tr>
<tr>
<td>numpy.repeat</td>
<td>cupy.repeat</td>
</tr>
<tr>
<td>numpy.require</td>
<td>cupy.require</td>
</tr>
<tr>
<td>numpy.reshape</td>
<td>cupy.reshape</td>
</tr>
<tr>
<td>numpy.resize</td>
<td>-</td>
</tr>
<tr>
<td>numpy.result_type</td>
<td>cupy.result_type</td>
</tr>
<tr>
<td>numpy.right_shift</td>
<td>cupy.right_shift</td>
</tr>
<tr>
<td>numpy.rint</td>
<td>cupy.rint</td>
</tr>
<tr>
<td>numpy.roll</td>
<td>cupy.roll</td>
</tr>
<tr>
<td>numpy.rollaxis</td>
<td>cupy.rollaxis</td>
</tr>
<tr>
<td>numpy.roots</td>
<td>cupy.roots</td>
</tr>
<tr>
<td>numpy.rot90</td>
<td>cupy.rot90</td>
</tr>
<tr>
<td>numpy.round</td>
<td>-</td>
</tr>
<tr>
<td>numpy.round_</td>
<td>cupy.round_</td>
</tr>
<tr>
<td>numpy.row_stack</td>
<td>-</td>
</tr>
<tr>
<td>numpy.safe_eval</td>
<td>-</td>
</tr>
<tr>
<td>numpy.save</td>
<td>cupy.save</td>
</tr>
<tr>
<td>numpy.savetxt</td>
<td>-</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>NumPy</th>
<th>CuPy</th>
</tr>
</thead>
<tbody>
<tr>
<td>numpy.savez</td>
<td>cupy.savez</td>
</tr>
<tr>
<td>numpy.savez_compressed</td>
<td>cupy.savez_compressed</td>
</tr>
<tr>
<td>numpy.sctype2char</td>
<td>cupy.sctype2char (alias of numpy.sctype2char)</td>
</tr>
<tr>
<td>numpy.searchsorted</td>
<td>cupy.searchsorted</td>
</tr>
<tr>
<td>numpy.select</td>
<td>cupy.select</td>
</tr>
<tr>
<td>numpy.set_numeric_ops</td>
<td>-</td>
</tr>
<tr>
<td>numpy.set_printoptions</td>
<td>-</td>
</tr>
<tr>
<td>numpy.set_string_function</td>
<td>-</td>
</tr>
<tr>
<td>numpy.setbufsize</td>
<td>-</td>
</tr>
<tr>
<td>numpy.setdiffld</td>
<td>-</td>
</tr>
<tr>
<td>numpy.seterr</td>
<td>-</td>
</tr>
<tr>
<td>numpy.seterrcall</td>
<td>-</td>
</tr>
<tr>
<td>numpy.seterrobj</td>
<td>-</td>
</tr>
<tr>
<td>numpy.setxord</td>
<td>-</td>
</tr>
<tr>
<td>numpy.shape</td>
<td>cupy.shape</td>
</tr>
<tr>
<td>numpy.shares_memory</td>
<td>cupy.shares_memory</td>
</tr>
<tr>
<td>numpy.show_config</td>
<td>cupy.show_config</td>
</tr>
<tr>
<td>numpy.sign</td>
<td>cupy.sign</td>
</tr>
<tr>
<td>numpy.signbit</td>
<td>cupy.signbit</td>
</tr>
<tr>
<td>numpy.sin</td>
<td>cupy.sin</td>
</tr>
<tr>
<td>numpy.sinc</td>
<td>cupy.sinc</td>
</tr>
<tr>
<td>numpy.sinh</td>
<td>cupy.sinh</td>
</tr>
<tr>
<td>numpy.size</td>
<td>cupy.size</td>
</tr>
<tr>
<td>numpy.sometrue</td>
<td>-</td>
</tr>
<tr>
<td>numpy.sort</td>
<td>cupy.sort</td>
</tr>
<tr>
<td>numpy.sort_complex</td>
<td>cupy.sort_complex</td>
</tr>
<tr>
<td>numpy.source</td>
<td>-</td>
</tr>
<tr>
<td>numpy.spacing</td>
<td>-</td>
</tr>
<tr>
<td>numpy.split</td>
<td>cupy.split</td>
</tr>
<tr>
<td>numpy.sqrt</td>
<td>cupy.sqrt</td>
</tr>
<tr>
<td>numpy.square</td>
<td>cupy.square</td>
</tr>
<tr>
<td>numpy.squeeze</td>
<td>cupy.squeeze</td>
</tr>
<tr>
<td>numpy.stack</td>
<td>cupy.stack</td>
</tr>
<tr>
<td>numpy.std</td>
<td>cupy.std</td>
</tr>
<tr>
<td>numpy.subtract</td>
<td>cupy.subtract</td>
</tr>
<tr>
<td>numpy.sum</td>
<td>cupy.sum</td>
</tr>
<tr>
<td>numpy.swapaxes</td>
<td>cupy.swapaxes</td>
</tr>
<tr>
<td>numpy.take</td>
<td>cupy.take</td>
</tr>
<tr>
<td>numpy.take_along_axis</td>
<td>cupy.take_along_axis</td>
</tr>
<tr>
<td>numpy.tan</td>
<td>cupy.tan</td>
</tr>
<tr>
<td>numpy.tanh</td>
<td>cupy.tanh</td>
</tr>
<tr>
<td>numpy.tensordot</td>
<td>cupy.tensordot</td>
</tr>
<tr>
<td>numpy.tile</td>
<td>cupy.tile</td>
</tr>
<tr>
<td>numpy.trace</td>
<td>cupy.trace</td>
</tr>
<tr>
<td>numpy.transpose</td>
<td>cupy.transpose</td>
</tr>
<tr>
<td>numpy.trapz</td>
<td>-</td>
</tr>
<tr>
<td>numpy.tri</td>
<td>cupy.tri</td>
</tr>
<tr>
<td>numpy.tril</td>
<td>cupy.tril</td>
</tr>
<tr>
<td>numpy.tril_indices</td>
<td>-</td>
</tr>
<tr>
<td>NumPy</td>
<td>CuPy</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td><code>numpy.tril_indices_from</code></td>
<td>-</td>
</tr>
<tr>
<td><code>numpy.trim_zeros</code></td>
<td><code>cupy.trim_zeros</code></td>
</tr>
<tr>
<td><code>numpy.triu</code></td>
<td><code>cupy.triu</code></td>
</tr>
<tr>
<td><code>numpy.triu_indices</code></td>
<td>-</td>
</tr>
<tr>
<td><code>numpy.triu_indices_from</code></td>
<td>-</td>
</tr>
<tr>
<td><code>numpy.true_divide</code></td>
<td><code>cupy.true_divide</code></td>
</tr>
<tr>
<td><code>numpy.trunc</code></td>
<td><code>cupy.trunc</code></td>
</tr>
<tr>
<td><code>numpy.typename</code></td>
<td><code>cupy.typename</code> (<strong>alias of</strong> <code>numpy.typename</code>)</td>
</tr>
<tr>
<td><code>numpy.union1d</code></td>
<td>-</td>
</tr>
<tr>
<td><code>numpy.unique</code></td>
<td><code>cupy.unique</code></td>
</tr>
<tr>
<td><code>numpy.unpackbits</code></td>
<td><code>cupy.unpackbits</code></td>
</tr>
<tr>
<td><code>numpy.unravel_index</code></td>
<td><code>cupy.unravel_index</code></td>
</tr>
<tr>
<td><code>numpy.unwrap</code></td>
<td><code>cupy.unwrap</code></td>
</tr>
<tr>
<td><code>numpy.vander</code></td>
<td>-</td>
</tr>
<tr>
<td><code>numpy.var</code></td>
<td><code>cupy.var</code></td>
</tr>
<tr>
<td><code>numpy.vdot</code></td>
<td><code>cupy.vdot</code></td>
</tr>
<tr>
<td><code>numpy.vsplit</code></td>
<td><code>cupy.vsplit</code></td>
</tr>
<tr>
<td><code>numpy.vstack</code></td>
<td><code>cupy.vstack</code></td>
</tr>
<tr>
<td><code>numpy.where</code></td>
<td><code>cupy.where</code></td>
</tr>
<tr>
<td><code>numpy.who</code></td>
<td><code>cupy.who</code></td>
</tr>
<tr>
<td><code>numpy.zeros</code></td>
<td><code>cupy.zeros</code></td>
</tr>
<tr>
<td><code>numpy.zeros_like</code></td>
<td><code>cupy.zeros_like</code></td>
</tr>
</tbody>
</table>

### Multi-Dimensional Array

<table>
<thead>
<tr>
<th>NumPy</th>
<th>CuPy</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>numpy.ndarray.all()</code></td>
<td><code>cupy.ndarray.all()</code></td>
</tr>
<tr>
<td><code>numpy.ndarray.any()</code></td>
<td><code>cupy.ndarray.any()</code></td>
</tr>
<tr>
<td><code>numpy.ndarray.argmax()</code></td>
<td><code>cupy.ndarray.argmax()</code></td>
</tr>
<tr>
<td><code>numpy.ndarray.argmin()</code></td>
<td><code>cupy.ndarray.argmin()</code></td>
</tr>
<tr>
<td><code>numpy.ndarray.argpartition()</code></td>
<td><code>cupy.ndarray.argpartition()</code></td>
</tr>
<tr>
<td><code>numpy.ndarray.argsort()</code></td>
<td><code>cupy.ndarray.argsort()</code></td>
</tr>
<tr>
<td><code>numpy.ndarray.astype()</code></td>
<td><code>cupy.ndarray.astype()</code></td>
</tr>
<tr>
<td><code>numpy.ndarray.byteswap()</code></td>
<td>-</td>
</tr>
<tr>
<td><code>numpy.ndarray.choose()</code></td>
<td><code>cupy.ndarray.choose()</code></td>
</tr>
<tr>
<td><code>numpy.ndarray.clip()</code></td>
<td><code>cupy.ndarray.clip()</code></td>
</tr>
<tr>
<td><code>numpy.ndarray.compress()</code></td>
<td><code>cupy.ndarray.compress()</code></td>
</tr>
<tr>
<td><code>numpy.ndarray.conj()</code></td>
<td><code>cupy.ndarray.conj()</code></td>
</tr>
<tr>
<td><code>numpy.ndarray.conjugate()</code></td>
<td><code>cupy.ndarray.conjugate()</code></td>
</tr>
<tr>
<td><code>numpy.ndarray.copy()</code></td>
<td><code>cupy.ndarray.copy()</code></td>
</tr>
<tr>
<td><code>numpy.ndarray.cumprod()</code></td>
<td><code>cupy.ndarray.cumprod()</code></td>
</tr>
<tr>
<td><code>numpy.ndarray.cumsum()</code></td>
<td><code>cupy.ndarray.cumsum()</code></td>
</tr>
<tr>
<td><code>numpy.ndarray.diagonal()</code></td>
<td><code>cupy.ndarray.diagonal()</code></td>
</tr>
<tr>
<td><code>numpy.ndarray.dot()</code></td>
<td><code>cupy.ndarray.dot()</code></td>
</tr>
<tr>
<td><code>numpy.ndarray.dump()</code></td>
<td><code>cupy.ndarray.dump()</code></td>
</tr>
<tr>
<td><code>numpy.ndarray.dumps()</code></td>
<td><code>cupy.ndarray.dumps()</code></td>
</tr>
<tr>
<td><code>numpy.ndarray.fill()</code></td>
<td><code>cupy.ndarray.fill()</code></td>
</tr>
<tr>
<td><code>numpy.ndarray.flatten()</code></td>
<td><code>cupy.ndarray.flatten()</code></td>
</tr>
</tbody>
</table>

Continued on next page
Table 118 – continued from previous page

<table>
<thead>
<tr>
<th>NumPy</th>
<th>CuPy</th>
</tr>
</thead>
<tbody>
<tr>
<td>numpy.ndarray.getfield()</td>
<td>-</td>
</tr>
<tr>
<td>numpy.ndarray.item()</td>
<td>cupy.ndarray.item()</td>
</tr>
<tr>
<td>numpy.ndarray.itemset()</td>
<td>-</td>
</tr>
<tr>
<td>numpy.ndarray.max()</td>
<td>cupy.ndarray.max()</td>
</tr>
<tr>
<td>numpy.ndarray.mean()</td>
<td>cupy.ndarray.mean()</td>
</tr>
<tr>
<td>numpy.ndarray.min()</td>
<td>cupy.ndarray.min()</td>
</tr>
<tr>
<td>numpy.ndarray.newbyteorder()</td>
<td>-</td>
</tr>
<tr>
<td>numpy.ndarray.nonzero()</td>
<td>cupy.ndarray.nonzero()</td>
</tr>
<tr>
<td>numpy.ndarray.partition()</td>
<td>cupy.ndarray.partition()</td>
</tr>
<tr>
<td>numpy.ndarray.prod()</td>
<td>cupy.ndarray.prod()</td>
</tr>
<tr>
<td>numpy.ndarray.ptp()</td>
<td>cupy.ndarray.ptp()</td>
</tr>
<tr>
<td>numpy.ndarray.put()</td>
<td>cupy.ndarray.put()</td>
</tr>
<tr>
<td>numpy.ndarray.ravel()</td>
<td>cupy.ndarray.ravel()</td>
</tr>
<tr>
<td>numpy.ndarray.repeat()</td>
<td>cupy.ndarray.repeat()</td>
</tr>
<tr>
<td>numpy.ndarray.reshape()</td>
<td>cupy.ndarray.reshape()</td>
</tr>
<tr>
<td>numpy.ndarray.resize()</td>
<td>-</td>
</tr>
<tr>
<td>numpy.ndarray.round()</td>
<td>cupy.ndarray.round()</td>
</tr>
<tr>
<td>numpy.ndarray.searchsorted()</td>
<td>-</td>
</tr>
<tr>
<td>numpy.ndarray.setfield()</td>
<td>-</td>
</tr>
<tr>
<td>numpy.ndarray.setflags()</td>
<td>-</td>
</tr>
<tr>
<td>numpy.ndarray.sort()</td>
<td>cupy.ndarray.sort()</td>
</tr>
<tr>
<td>numpy.ndarray.squeeze()</td>
<td>cupy.ndarray.squeeze()</td>
</tr>
<tr>
<td>numpy.ndarray.std()</td>
<td>cupy.ndarray.std()</td>
</tr>
<tr>
<td>numpy.ndarray.sum()</td>
<td>cupy.ndarray.sum()</td>
</tr>
<tr>
<td>numpy.ndarray.swapaxes()</td>
<td>cupy.ndarray.swapaxes()</td>
</tr>
<tr>
<td>numpy.ndarray.take()</td>
<td>cupy.ndarray.take()</td>
</tr>
<tr>
<td>numpy.ndarray.tobytes()</td>
<td>cupy.ndarray.tobytes()</td>
</tr>
<tr>
<td>numpy.ndarray.tofile()</td>
<td>cupy.ndarray.tofile()</td>
</tr>
<tr>
<td>numpy.ndarray.tolist()</td>
<td>cupy.ndarray.tolist()</td>
</tr>
<tr>
<td>numpy.ndarray.tostring()</td>
<td>-</td>
</tr>
<tr>
<td>numpy.ndarray.trace()</td>
<td>cupy.ndarray.trace()</td>
</tr>
<tr>
<td>numpy.ndarray.transpose()</td>
<td>cupy.ndarray.transpose()</td>
</tr>
<tr>
<td>numpy.ndarray.var()</td>
<td>cupy.ndarray.var()</td>
</tr>
<tr>
<td>numpy.ndarray.view()</td>
<td>cupy.ndarray.view()</td>
</tr>
</tbody>
</table>
Linear Algebra

<table>
<thead>
<tr>
<th>NumPy</th>
<th>CuPy</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>numpy.linalg.cholesky</code></td>
<td><code>cupy.linalg.cholesky</code></td>
</tr>
<tr>
<td><code>numpy.linalg.cond</code></td>
<td>-</td>
</tr>
<tr>
<td><code>numpy.linalg.det</code></td>
<td><code>cupy.linalg.det</code></td>
</tr>
<tr>
<td><code>numpy.linalg.eig</code></td>
<td>-</td>
</tr>
<tr>
<td><code>numpy.linalg.eigh</code></td>
<td><code>cupy.linalg.eigh</code></td>
</tr>
<tr>
<td><code>numpy.linalg.eigvals</code></td>
<td>-</td>
</tr>
<tr>
<td><code>numpy.linalg.eigvalsh</code></td>
<td><code>cupy.linalg.eigvalsh</code></td>
</tr>
<tr>
<td><code>numpy.linalg.inv</code></td>
<td><code>cupy.linalg.inv</code></td>
</tr>
<tr>
<td><code>numpy.linalg.lstsq</code></td>
<td><code>cupy.linalg.lstsq</code></td>
</tr>
<tr>
<td><code>numpy.linalg.matrix_power</code></td>
<td><code>cupy.linalg.matrix_power</code></td>
</tr>
<tr>
<td><code>numpy.linalg.matrix_rank</code></td>
<td><code>cupy.linalg.matrix_rank</code></td>
</tr>
<tr>
<td><code>numpy.linalg.multi_dot</code></td>
<td>-</td>
</tr>
<tr>
<td><code>numpy.linalg.norm</code></td>
<td><code>cupy.linalg.norm</code></td>
</tr>
<tr>
<td><code>numpy.linalg.pinv</code></td>
<td><code>cupy.linalg.pinv</code></td>
</tr>
<tr>
<td><code>numpy.linalg.qr</code></td>
<td><code>cupy.linalg.qr</code></td>
</tr>
<tr>
<td><code>numpy.linalg.slogdet</code></td>
<td><code>cupy.linalg.slogdet</code></td>
</tr>
<tr>
<td><code>numpy.linalg.solve</code></td>
<td><code>cupy.linalg.solve</code></td>
</tr>
<tr>
<td><code>numpy.linalg.svd</code></td>
<td><code>cupy.linalg.svd</code></td>
</tr>
<tr>
<td><code>numpy.linalg.tensorinv</code></td>
<td><code>cupy.linalg.tensorinv</code></td>
</tr>
<tr>
<td><code>numpy.linalg.tensorsolve</code></td>
<td><code>cupy.linalg.tensorsolve</code></td>
</tr>
</tbody>
</table>

Discrete Fourier Transform

<table>
<thead>
<tr>
<th>NumPy</th>
<th>CuPy</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>numpy.fft.fft</code></td>
<td><code>cupy.fft.fft</code></td>
</tr>
<tr>
<td><code>numpy.fft.fft2</code></td>
<td><code>cupy.fft.fft2</code></td>
</tr>
<tr>
<td><code>numpy.fft.fftfreq</code></td>
<td><code>cupy.fft.fftfreq</code></td>
</tr>
<tr>
<td><code>numpy.fft.fftn</code></td>
<td><code>cupy.fft.fftn</code></td>
</tr>
<tr>
<td><code>numpy.fft.fftshift</code></td>
<td><code>cupy.fft.fftshift</code></td>
</tr>
<tr>
<td><code>numpy.fft.hfft</code></td>
<td><code>cupy.fft.hfft</code></td>
</tr>
<tr>
<td><code>numpy.fft.ifft</code></td>
<td><code>cupy.fft.ifft</code></td>
</tr>
<tr>
<td><code>numpy.fft.ifft2</code></td>
<td><code>cupy.fft.ifft2</code></td>
</tr>
<tr>
<td><code>numpy.fft.ifftn</code></td>
<td><code>cupy.fft.ifftn</code></td>
</tr>
<tr>
<td><code>numpy.fft.ifftshift</code></td>
<td><code>cupy.fft.ifftshift</code></td>
</tr>
<tr>
<td><code>numpy.fft.ihfft</code></td>
<td><code>cupy.fft.ihfft</code></td>
</tr>
<tr>
<td><code>numpy.fft.irfft</code></td>
<td><code>cupy.fft.irfft</code></td>
</tr>
<tr>
<td><code>numpy.fft.irfft2</code></td>
<td><code>cupy.fft.irfft2</code></td>
</tr>
<tr>
<td><code>numpy.fft.irfftn</code></td>
<td><code>cupy.fft.irfftn</code></td>
</tr>
<tr>
<td><code>numpy.fft.rfft</code></td>
<td><code>cupy.fft.rfft</code></td>
</tr>
<tr>
<td><code>numpy.fft.rfft2</code></td>
<td><code>cupy.fft.rfft2</code></td>
</tr>
<tr>
<td><code>numpy.fft.rfftfreq</code></td>
<td><code>cupy.fft.rfftfreq</code></td>
</tr>
<tr>
<td><code>numpy.fft.rfftn</code></td>
<td><code>cupy.fft.rfftn</code></td>
</tr>
</tbody>
</table>

Random Sampling
<table>
<thead>
<tr>
<th>NumPy</th>
<th>CuPy</th>
</tr>
</thead>
<tbody>
<tr>
<td>numpy.random.beta</td>
<td>cupy.random.beta</td>
</tr>
<tr>
<td>numpy.random.binomial</td>
<td>cupy.random.binomial</td>
</tr>
<tr>
<td>numpy.random.bytes</td>
<td>cupy.random.bytes</td>
</tr>
<tr>
<td>numpy.random.chisquare</td>
<td>cupy.random.chisquare</td>
</tr>
<tr>
<td>numpy.random.choice</td>
<td>cupy.random.choice</td>
</tr>
<tr>
<td>numpy.random.default_rng</td>
<td>-</td>
</tr>
<tr>
<td>numpy.random.dirichlet</td>
<td>cupy.random.dirichlet</td>
</tr>
<tr>
<td>numpy.random.exponential</td>
<td>cupy.random.exponential</td>
</tr>
<tr>
<td>numpy.random.f</td>
<td>cupy.random.f</td>
</tr>
<tr>
<td>numpy.random.gamma</td>
<td>cupy.random.gamma</td>
</tr>
<tr>
<td>numpy.random.geometric</td>
<td>cupy.random.geometric</td>
</tr>
<tr>
<td>numpy.random.get_state</td>
<td>-</td>
</tr>
<tr>
<td>numpy.random.gumbel</td>
<td>cupy.random.gumbel</td>
</tr>
<tr>
<td>numpy.random.hypergeometric</td>
<td>cupy.random.hypergeometric</td>
</tr>
<tr>
<td>numpy.random.laplace</td>
<td>cupy.random.laplace</td>
</tr>
<tr>
<td>numpy.random.logistic</td>
<td>cupy.random.logistic</td>
</tr>
<tr>
<td>numpy.random.lognormal</td>
<td>cupy.random.lognormal</td>
</tr>
<tr>
<td>numpy.random.logseries</td>
<td>cupy.random.logseries</td>
</tr>
<tr>
<td>numpy.random.multinomial</td>
<td>cupy.random.multinomial</td>
</tr>
<tr>
<td>numpy.random.multivariate_normal</td>
<td>cupy.random.multivariate_normal</td>
</tr>
<tr>
<td>numpy.random.negative_binomial</td>
<td>cupy.random.negative_binomial</td>
</tr>
<tr>
<td>numpy.random.noncentral_chisquare</td>
<td>cupy.random.noncentral_chisquare</td>
</tr>
<tr>
<td>numpy.random.noncentral_f</td>
<td>cupy.random.noncentral_f</td>
</tr>
<tr>
<td>numpy.random.normal</td>
<td>cupy.random.normal</td>
</tr>
<tr>
<td>numpy.random.pareto</td>
<td>cupy.random.pareto</td>
</tr>
<tr>
<td>numpy.random.permutation</td>
<td>cupy.random.permutation</td>
</tr>
<tr>
<td>numpy.random.poisson</td>
<td>cupy.random.poisson</td>
</tr>
<tr>
<td>numpy.random.power</td>
<td>cupy.random.power</td>
</tr>
<tr>
<td>numpy.random.ran</td>
<td>cupy.random.ran</td>
</tr>
<tr>
<td>numpy.random.randint</td>
<td>cupy.random.randint</td>
</tr>
<tr>
<td>numpy.random.rand</td>
<td>cupy.random.rand</td>
</tr>
<tr>
<td>numpy.random.random_integers</td>
<td>cupy.random.random_integers</td>
</tr>
<tr>
<td>numpy.random.random_sample</td>
<td>cupy.random.random_sample</td>
</tr>
<tr>
<td>numpy.random.random_set_state</td>
<td>-</td>
</tr>
<tr>
<td>numpy.random.shuffle</td>
<td>cupy.random.shuffle</td>
</tr>
<tr>
<td>numpy.random.standard_cauchy</td>
<td>cupy.random.standard_cauchy</td>
</tr>
<tr>
<td>numpy.random.standard_exponential</td>
<td>cupy.random.standard_exponential</td>
</tr>
<tr>
<td>numpy.random.standard_gamma</td>
<td>cupy.random.standard_gamma</td>
</tr>
<tr>
<td>numpy.random.standard_normal</td>
<td>cupy.random.standard_normal</td>
</tr>
<tr>
<td>numpy.random.standard_t</td>
<td>cupy.random.standard_t</td>
</tr>
<tr>
<td>numpy.random.triangular</td>
<td>cupy.random.triangular</td>
</tr>
<tr>
<td>numpy.random.uniform</td>
<td>cupy.random.uniform</td>
</tr>
<tr>
<td>numpy.random.vonmises</td>
<td>cupy.random.vonmises</td>
</tr>
<tr>
<td>numpy.random.wald</td>
<td>cupy.random.wald</td>
</tr>
<tr>
<td>numpy.random.weibull</td>
<td>cupy.random.weibull</td>
</tr>
</tbody>
</table>

Continued on next page
CuPy Documentation, Release 8.2.0

Table 119 – continued from previous page

<table>
<thead>
<tr>
<th>NumPy</th>
<th>CuPy</th>
</tr>
</thead>
<tbody>
<tr>
<td>numpy.random.zipf</td>
<td>cupy.random.zipf</td>
</tr>
</tbody>
</table>

4.16.2 SciPy / CuPy APIs

Discrete Fourier Transform

<table>
<thead>
<tr>
<th>SciPy</th>
<th>CuPy</th>
</tr>
</thead>
<tbody>
<tr>
<td>scipy.fftpack.cc_diff</td>
<td>-</td>
</tr>
<tr>
<td>scipy.fftpack.cs_diff</td>
<td>-</td>
</tr>
<tr>
<td>scipy.fftpack.dct</td>
<td>-</td>
</tr>
<tr>
<td>scipy.fftpack.dctn</td>
<td>-</td>
</tr>
<tr>
<td>scipy.fftpack.diff</td>
<td>-</td>
</tr>
<tr>
<td>scipy.fftpack.dst</td>
<td>-</td>
</tr>
<tr>
<td>scipy.fftpack.dstn</td>
<td>-</td>
</tr>
<tr>
<td>scipy.fftpack.fft</td>
<td>cupyx.scipy.fftpack.fft</td>
</tr>
<tr>
<td>scipy.fftpack.fft2</td>
<td>cupyx.scipy.fftpack.fft2</td>
</tr>
<tr>
<td>scipy.fftpack.fftfreq</td>
<td>-</td>
</tr>
<tr>
<td>scipy.fftpack.fftn</td>
<td>cupyx.scipy.fftpack.fftn</td>
</tr>
<tr>
<td>scipy.fftpack.fftshift</td>
<td>-</td>
</tr>
<tr>
<td>scipy.fftpack.hilbert</td>
<td>-</td>
</tr>
<tr>
<td>scipy.fftpack.idct</td>
<td>-</td>
</tr>
<tr>
<td>scipy.fftpack.idctn</td>
<td>-</td>
</tr>
<tr>
<td>scipy.fftpack.idst</td>
<td>-</td>
</tr>
<tr>
<td>scipy.fftpack.idstn</td>
<td>-</td>
</tr>
<tr>
<td>scipy.fftpack.ifft</td>
<td>cupyx.scipy.fftpack.ifft</td>
</tr>
<tr>
<td>scipy.fftpack.ifft2</td>
<td>cupyx.scipy.fftpack.ifft2</td>
</tr>
<tr>
<td>scipy.fftpack.ifftn</td>
<td>cupyx.scipy.fftpack.ifftn</td>
</tr>
<tr>
<td>scipy.fftpack.ifftshift</td>
<td>-</td>
</tr>
<tr>
<td>scipy.fftpack.ihilbert</td>
<td>-</td>
</tr>
<tr>
<td>scipy.fftpack.irfft</td>
<td>cupyx.scipy.fftpack.irfft</td>
</tr>
<tr>
<td>scipy.fftpack.itilbert</td>
<td>-</td>
</tr>
<tr>
<td>scipy.fftpack.next_fast_len</td>
<td>-</td>
</tr>
<tr>
<td>scipy.fftpack.rfft</td>
<td>cupyx.scipy.fftpack.rfft</td>
</tr>
<tr>
<td>scipy.fftpack.rfftfreq</td>
<td>-</td>
</tr>
<tr>
<td>scipy.fftpack.sc_diff</td>
<td>-</td>
</tr>
<tr>
<td>scipy.fftpack.ss_diff</td>
<td>-</td>
</tr>
<tr>
<td>scipy.fftpack.tilbert</td>
<td>-</td>
</tr>
</tbody>
</table>
## Sparse Matrices

<table>
<thead>
<tr>
<th>SciPy</th>
<th>CuPy</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>scipy.sparse.block_diag</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.sparse.bmat</code></td>
<td><code>cupyx.scipy.sparse.bmat</code></td>
</tr>
<tr>
<td><code>scipy.sparse.diags</code></td>
<td><code>cupyx.scipy.sparse.diags</code></td>
</tr>
<tr>
<td><code>scipy.sparse.eye</code></td>
<td><code>cupyx.scipy.sparse.eye</code></td>
</tr>
<tr>
<td><code>scipy.sparse.find</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.sparse.hstack</code></td>
<td><code>cupyx.scipy.sparse.hstack</code></td>
</tr>
<tr>
<td><code>scipy.sparse.identity</code></td>
<td><code>cupyx.scipy.sparse.identity</code></td>
</tr>
<tr>
<td><code>scipy.sparse.issparse</code></td>
<td><code>cupyx.scipy.sparse.issparse</code></td>
</tr>
<tr>
<td><code>scipy.sparse.isspmatrix</code></td>
<td><code>cupyx.scipy.sparse.isspmatrix</code></td>
</tr>
<tr>
<td><code>scipy.sparse.isspmatrix_bsr</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.sparse.isspmatrix_coo</code></td>
<td><code>cupyx.scipy.sparse.isspmatrix_coo</code></td>
</tr>
<tr>
<td><code>scipy.sparse.isspmatrix_csc</code></td>
<td><code>cupyx.scipy.sparse.isspmatrix_csc</code></td>
</tr>
<tr>
<td><code>scipy.sparse.isspmatrix_csr</code></td>
<td><code>cupyx.scipy.sparse.isspmatrix_csr</code></td>
</tr>
<tr>
<td><code>scipy.sparse.isspmatrix_dia</code></td>
<td><code>cupyx.scipy.sparse.isspmatrix_dia</code></td>
</tr>
<tr>
<td><code>scipy.sparse.isspmatrix_dok</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.sparse.isspmatrix_lil</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.sparse.kron</code></td>
<td><code>cupyx.scipy.sparse.kron</code></td>
</tr>
<tr>
<td><code>scipy.sparse.kronsum</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.sparse.load_npz</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.sparse.rand</code></td>
<td><code>cupyx.scipy.sparse.rand</code></td>
</tr>
<tr>
<td><code>scipy.sparse.random</code></td>
<td><code>cupyx.scipy.sparse.random</code></td>
</tr>
<tr>
<td><code>scipy.sparse.save_npz</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.sparse.spdiags</code></td>
<td><code>cupyx.scipy.sparse.spdiags</code></td>
</tr>
<tr>
<td><code>scipy.sparse.tril</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.sparse.triu</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.sparse.vstack</code></td>
<td><code>cupyx.scipy.sparse.vstack</code></td>
</tr>
</tbody>
</table>
## Sparse Linear Algebra

<table>
<thead>
<tr>
<th>SciPy</th>
<th>CuPy</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>scipy.sparse.linalg.aslinearoperator</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.sparse.linalg.bicg</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.sparse.linalg.bicgstab</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.sparse.linalg.cg</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.sparse.linalg.cgs</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.sparse.linalg.eigs</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.sparse.linalg.eigsh</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.sparse.linalg.expm</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.sparse.linalg.expm_multiply</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.sparse.linalg.factorized</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.sparse.linalg.gcrotmk</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.sparse.linalg.gmres</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.sparse.linalg.inv</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.sparse.linalg.lgmres</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.sparse.linalg.lobpcg</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.sparse.linalg.lsmr</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.sparse.linalg.lsmr</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.sparse.linalg.mlinalg.norm</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.sparse.linalg.mnorm</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.sparse.linalg.onenormest</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.sparse.linalg.qmr</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.sparse.linalg.spilu</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.sparse.linalg.splu</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.sparse.linalg.spsolve</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.sparse.linalg.spsolve_triangular</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.sparse.linalg.svds</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.sparse.linalg.use_solver</code></td>
<td>-</td>
</tr>
</tbody>
</table>

## Advanced Linear Algebra

<table>
<thead>
<tr>
<th>SciPy</th>
<th>CuPy</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>scipy.linalg.block_diag</code></td>
<td><code>cupyx.scipy.linalg.block_diag</code></td>
</tr>
<tr>
<td><code>scipy.linalg.cdf2rdf</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.linalg.cho_factor</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.linalg.cho_solve</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.linalg.cho_solve_banded</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.linalg.cholesky_banded</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.linalg.circulant</code></td>
<td><code>cupyx.scipy.linalg.circulant</code></td>
</tr>
<tr>
<td><code>scipy.linalg.clarkson_woodruff_transform</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.linalg.companion</code></td>
<td><code>cupyx.scipy.linalg.companion</code></td>
</tr>
<tr>
<td><code>scipy.linalg.convolution_matrix</code></td>
<td><code>cupyx.scipy.linalg.convolution_matrix</code></td>
</tr>
<tr>
<td><code>scipy.linalg.diagsvd</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.linalg.cosm</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.linalg.cossin</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.linalg.dft</code></td>
<td><code>cupyx.scipy.linalg.dft</code></td>
</tr>
<tr>
<td><code>scipy.linalg.diagsvd</code></td>
<td>-</td>
</tr>
</tbody>
</table>

Continued on next page
## Table 121 – continued from previous page

<table>
<thead>
<tr>
<th>SciPy</th>
<th>CuPy</th>
</tr>
</thead>
<tbody>
<tr>
<td>scipy.linalg.eig_banded</td>
<td>-</td>
</tr>
<tr>
<td>scipy.linalg.eigh_tridiagonal</td>
<td>-</td>
</tr>
<tr>
<td>scipy.linalg.eigvals_banded</td>
<td>-</td>
</tr>
<tr>
<td>scipy.linalg.eigvalsh_tridiagonal</td>
<td>-</td>
</tr>
<tr>
<td>scipy.linalg.expm</td>
<td>-</td>
</tr>
<tr>
<td>scipy.linalg.expm_cond</td>
<td>-</td>
</tr>
<tr>
<td>scipy.linalg.expm_frechet</td>
<td>-</td>
</tr>
<tr>
<td>scipy.linalg.fiedler</td>
<td>cupyx.scipy.linalg.fiedler</td>
</tr>
<tr>
<td>scipy.linalg.fiedler_companion</td>
<td>cupyx.scipy.linalg.fiedler_companion</td>
</tr>
<tr>
<td>scipy.linalg.find_best_blas_type</td>
<td>-</td>
</tr>
<tr>
<td>scipy.linalg.fractional_matrix_power</td>
<td>-</td>
</tr>
<tr>
<td>scipy.linalg.funm</td>
<td>-</td>
</tr>
<tr>
<td>scipy.linalg.get_blas_funics</td>
<td>-</td>
</tr>
<tr>
<td>scipy.linalg.get_lapack_funics</td>
<td>-</td>
</tr>
<tr>
<td>scipy.linalg.hadamard</td>
<td>cupyx.scipy.linalg.hadamard</td>
</tr>
<tr>
<td>scipy.linalg.hankel</td>
<td>cupyx.scipy.linalg.hankel</td>
</tr>
<tr>
<td>scipy.linalg.helmert</td>
<td>cupyx.scipy.linalg.helmert</td>
</tr>
<tr>
<td>scipy.linalg.hessenberg</td>
<td>-</td>
</tr>
<tr>
<td>scipy.linalg.hilbert</td>
<td>cupyx.scipy.linalg.hilbert</td>
</tr>
<tr>
<td>scipy.linalg.invhilbert</td>
<td>-</td>
</tr>
<tr>
<td>scipy.linalg.invpascal</td>
<td>-</td>
</tr>
<tr>
<td>scipy.linalg.khatri_rao</td>
<td>-</td>
</tr>
<tr>
<td>scipy.linalg.kron</td>
<td>cupyx.scipy.linalg.kron</td>
</tr>
<tr>
<td>scipy.linalg.ldl</td>
<td>-</td>
</tr>
<tr>
<td>scipy.linalg.leslie</td>
<td>cupyx.scipy.linalg.leslie</td>
</tr>
<tr>
<td>scipy.linalg.logm</td>
<td>-</td>
</tr>
<tr>
<td>scipy.linalg.lu</td>
<td>-</td>
</tr>
<tr>
<td>scipy.linalg.lu_factor</td>
<td>cupyx.scipy.linalg.lu_factor</td>
</tr>
<tr>
<td>scipy.linalg.lu_solve</td>
<td>cupyx.scipy.linalg.lu_solve</td>
</tr>
<tr>
<td>scipy.linalg.matrix_balance</td>
<td>-</td>
</tr>
<tr>
<td>scipy.linalg.null_space</td>
<td>-</td>
</tr>
<tr>
<td>scipy.linalg.ordqz</td>
<td>-</td>
</tr>
<tr>
<td>scipy.linalg.orth</td>
<td>-</td>
</tr>
<tr>
<td>scipy.linalg.orthogonal_procrustes</td>
<td>-</td>
</tr>
<tr>
<td>scipy.linalg.pascal</td>
<td>-</td>
</tr>
<tr>
<td>scipy.linalg.pinv2</td>
<td>-</td>
</tr>
<tr>
<td>scipy.linalg.pinvh</td>
<td>-</td>
</tr>
<tr>
<td>scipy.linalg.polar</td>
<td>-</td>
</tr>
<tr>
<td>scipy.linalg.qr_delete</td>
<td>-</td>
</tr>
<tr>
<td>scipy.linalg.qr_insert</td>
<td>-</td>
</tr>
<tr>
<td>scipy.linalg.qr_multiply</td>
<td>-</td>
</tr>
<tr>
<td>scipy.linalg.qr_update</td>
<td>-</td>
</tr>
<tr>
<td>scipy.linalg.qz</td>
<td>-</td>
</tr>
<tr>
<td>scipy.linalg.rq</td>
<td>-</td>
</tr>
<tr>
<td>scipy.linalg.rsf2csf</td>
<td>-</td>
</tr>
<tr>
<td>scipy.linalg.schur</td>
<td>-</td>
</tr>
<tr>
<td>scipy.linalg.signm</td>
<td>-</td>
</tr>
<tr>
<td>scipy.linalg.sinhm</td>
<td>-</td>
</tr>
<tr>
<td>scipy.linalg.sinm</td>
<td>-</td>
</tr>
</tbody>
</table>

Continued on next page
### Table 121 – continued from previous page

<table>
<thead>
<tr>
<th>SciPy</th>
<th>CuPy</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>scipy.linalg.solve_banded</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.linalg.solve_circulant</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.linalg.solve_continuous_are</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.linalg.solve_continuous_lyapunov</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.linalg.solve_discrete_are</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.linalg.solve_discrete_lyapunov</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.linalg.solve_lyapunov</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.linalg.solve_sylvester</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.linalg.solve_toeplitz</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.linalg.solve_triangular</code></td>
<td><code>cupyx.scipy.linalg.solve_triangular</code></td>
</tr>
<tr>
<td><code>scipy.linalg.solveveh_banded</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.linalg.sqrtm</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.linalg.subspace_angles</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.linalg.svdvals</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.linalg.tanhm</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.linalg.tanm</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.linalg.toeplitz</code></td>
<td><code>cupyx.scipy.linalg.toeplitz</code></td>
</tr>
<tr>
<td><code>scipy.linalg.tri</code></td>
<td><code>cupyx.scipy.linalg.tri</code></td>
</tr>
<tr>
<td><code>scipy.linalg.tril</code></td>
<td><code>cupyx.scipy.linalg.tril</code></td>
</tr>
<tr>
<td><code>scipy.linalg.triu</code></td>
<td><code>cupyx.scipy.linalg.triu</code></td>
</tr>
</tbody>
</table>

### Multidimensional Image Processing

<table>
<thead>
<tr>
<th>SciPy</th>
<th>CuPy</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>scipy.ndimage.affine_transform</code></td>
<td><code>cupyx.scipy.ndimage.affine_transform</code></td>
</tr>
<tr>
<td><code>scipy.ndimage.binary_closing</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.ndimage.binary_dilation</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.ndimage.binary_erosion</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.ndimage.binary_fill_holes</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.ndimage.binary_hit_or_miss</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.ndimage.binary_opening</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.ndimage.binary_propagation</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.ndimage.black_tophat</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.ndimage.center_of_mass</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.ndimage.convolve</code></td>
<td><code>cupyx.scipy.ndimage.convolve</code></td>
</tr>
<tr>
<td><code>scipy.ndimage.convolve1d</code></td>
<td><code>cupyx.scipy.ndimage.convolve1d</code></td>
</tr>
<tr>
<td><code>scipy.ndimage.correlate</code></td>
<td><code>cupyx.scipy.ndimage.correlate</code></td>
</tr>
<tr>
<td><code>scipy.ndimage.correlate1d</code></td>
<td><code>cupyx.scipy.ndimage.correlate1d</code></td>
</tr>
<tr>
<td><code>scipy.ndimage.distance_transform_bf</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.ndimage.distance_transform_cdt</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.ndimage.distance_transform_edt</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.ndimage.extrema</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.ndimage.find_objects</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.ndimage.fourier_ellipsoid</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.ndimage.fourier_gaussian</code></td>
<td><code>cupyx.scipy.ndimage.fourier_gaussian</code></td>
</tr>
<tr>
<td><code>scipy.ndimage.fourier_shift</code></td>
<td><code>cupyx.scipy.ndimage.fourier_shift</code></td>
</tr>
<tr>
<td><code>scipy.ndimage.fourier_uniform</code></td>
<td><code>cupyx.scipy.ndimage.fourier_uniform</code></td>
</tr>
<tr>
<td><code>scipy.ndimage.gaussian_filter</code></td>
<td><code>cupyx.scipy.ndimage.gaussian_filter</code></td>
</tr>
</tbody>
</table>

Continued on next page

---

4.16. **Comparison Table**
Table 122 – continued from previous page

<table>
<thead>
<tr>
<th>SciPy</th>
<th>CuPy</th>
</tr>
</thead>
<tbody>
<tr>
<td>scipy.ndimage.gaussian_filter1d</td>
<td>cupyx.scipy.ndimage.gaussian_filter1d</td>
</tr>
<tr>
<td>scipy.ndimage.gaussian_gradient_magnitude</td>
<td>cupyx.scipy.ndimage.gaussian_gradient_magnitude</td>
</tr>
<tr>
<td>scipy.ndimage.gaussian_laplace</td>
<td>cupyx.scipy.ndimage.gaussian_laplace</td>
</tr>
<tr>
<td>scipy.ndimage.generate_binary_structure</td>
<td>-</td>
</tr>
<tr>
<td>scipy.ndimage.generic_filter</td>
<td>cupyx.scipy.ndimage.generic_filter</td>
</tr>
<tr>
<td>scipy.ndimage.generic_filter1d</td>
<td>cupyx.scipy.ndimage.generic_filter1d</td>
</tr>
<tr>
<td>scipy.ndimage.generic_gradient_magnitude</td>
<td>cupyx.scipy.ndimage.generic_gradient_magnitude</td>
</tr>
<tr>
<td>scipy.ndimage.generic_laplace</td>
<td>cupyx.scipy.ndimage.generic_laplace</td>
</tr>
<tr>
<td>scipy.ndimage.geometric_transform</td>
<td>-</td>
</tr>
<tr>
<td>scipy.ndimage.grey_closing</td>
<td>cupyx.scipy.ndimage.grey_closing</td>
</tr>
<tr>
<td>scipy.ndimage.grey_dilation</td>
<td>cupyx.scipy.ndimage.grey_dilation</td>
</tr>
<tr>
<td>scipy.ndimage.grey_erosion</td>
<td>cupyx.scipy.ndimage.grey_erosion</td>
</tr>
<tr>
<td>scipy.ndimage.grey_opening</td>
<td>cupyx.scipy.ndimage.grey_opening</td>
</tr>
<tr>
<td>scipy.ndimage.histogram</td>
<td>-</td>
</tr>
<tr>
<td>scipy.ndimage.iterate_structure</td>
<td>-</td>
</tr>
<tr>
<td>scipy.ndimage.label</td>
<td>cupyx.scipy.ndimage.label</td>
</tr>
<tr>
<td>scipy.ndimage.labeled_comprehension</td>
<td>-</td>
</tr>
<tr>
<td>scipy.ndimage.laplace</td>
<td>cupyx.scipy.ndimage.laplace</td>
</tr>
<tr>
<td>scipy.ndimage.map_coordinates</td>
<td>cupyx.scipy.ndimage.map_coordinates</td>
</tr>
<tr>
<td>scipy.ndimage.maximum</td>
<td>-</td>
</tr>
<tr>
<td>scipy.ndimage.maximum_filter</td>
<td>cupyx.scipy.ndimage.maximum_filter</td>
</tr>
<tr>
<td>scipy.ndimage.maximum_filter1d</td>
<td>cupyx.scipy.ndimage.maximum_filter1d</td>
</tr>
<tr>
<td>scipy.ndimage.maximum_position</td>
<td>-</td>
</tr>
<tr>
<td>scipy.ndimage.mean</td>
<td>cupyx.scipy.ndimage.mean</td>
</tr>
<tr>
<td>scipy.ndimage.median</td>
<td>-</td>
</tr>
<tr>
<td>scipy.ndimage.median_filter</td>
<td>cupyx.scipy.ndimage.median_filter</td>
</tr>
<tr>
<td>scipy.ndimage.minimum</td>
<td>-</td>
</tr>
<tr>
<td>scipy.ndimage.minimum_filter</td>
<td>cupyx.scipy.ndimage.minimum_filter</td>
</tr>
<tr>
<td>scipy.ndimage.minimum_filter1d</td>
<td>cupyx.scipy.ndimage.minimum_filter1d</td>
</tr>
<tr>
<td>scipy.ndimage.minimum_position</td>
<td>-</td>
</tr>
<tr>
<td>scipy.ndimage.morphological_gradient</td>
<td>-</td>
</tr>
<tr>
<td>scipy.ndimage.morphological_laplace</td>
<td>-</td>
</tr>
<tr>
<td>scipy.ndimage.percentile_filter</td>
<td>cupyx.scipy.ndimage.percentile_filter</td>
</tr>
<tr>
<td>scipy.ndimage.prewitt</td>
<td>cupyx.scipy.ndimage.prewitt</td>
</tr>
<tr>
<td>scipy.ndimage.rank_filter</td>
<td>cupyx.scipy.ndimage.rank_filter</td>
</tr>
<tr>
<td>scipy.ndimage.rotate</td>
<td>cupyx.scipy.ndimage.rotate</td>
</tr>
<tr>
<td>scipy.ndimage.shift</td>
<td>cupyx.scipy.ndimage.shift</td>
</tr>
<tr>
<td>scipy.ndimage.sobel</td>
<td>cupyx.scipy.ndimage.sobel</td>
</tr>
<tr>
<td>scipy.ndimage.spline_filter</td>
<td>-</td>
</tr>
<tr>
<td>scipy.ndimage.spline_filter1d</td>
<td>-</td>
</tr>
<tr>
<td>scipy.ndimage.standard_deviation</td>
<td>cupyx.scipy.ndimage.standard_deviation</td>
</tr>
<tr>
<td>scipy.ndimage.sum</td>
<td>cupyx.scipy.ndimage.sum</td>
</tr>
<tr>
<td>scipy.ndimage.uniform_filter</td>
<td>cupyx.scipy.ndimage.uniform_filter</td>
</tr>
<tr>
<td>scipy.ndimage.uniform_filter1d</td>
<td>cupyx.scipy.ndimage.uniform_filter1d</td>
</tr>
<tr>
<td>scipy.ndimage.variance</td>
<td>cupyx.scipy.ndimage.variance</td>
</tr>
<tr>
<td>scipy.ndimage.watershed_ift</td>
<td>-</td>
</tr>
<tr>
<td>scipy.ndimage.white_tophat</td>
<td>-</td>
</tr>
<tr>
<td>scipy.ndimage.zoom</td>
<td>cupyx.scipy.ndimage.zoom</td>
</tr>
</tbody>
</table>
## Special Functions

<table>
<thead>
<tr>
<th>SciPy</th>
<th>CuPy</th>
</tr>
</thead>
<tbody>
<tr>
<td>scipy.special.agm</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.ai_zeros</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.airy</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.airye</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.assoc_laguerre</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.bdtr</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.bdtrc</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.bdtri</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.bdtrik</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.bdtrin</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.bel</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.bei_zeros</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.beip</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.beip_zeros</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.ber</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.ber_zeros</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.bernoulli</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.berp</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.berp_zeros</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.besselpoly</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.beta</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.betainc</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.betaincinv</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.betaln</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.bi_zeros</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.binom</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.boxcox</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.boxcoxlp</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.btdtr</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.btdtrc</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.btdtri</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.btdtria</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.btdtrib</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.c_roots</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.cbpt</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.cg_roots</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.chdtr</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.chdtrc</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.chdtrc</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.chdtrc</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.chdtriv</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.chebyc</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.chebys</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.chebyt</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.chebyu</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.chndtr</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.chndtridf</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.chndtrinc</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.chndtrix</td>
<td>-</td>
</tr>
</tbody>
</table>

Continued on next page
Table 123 – continued from previous page

<table>
<thead>
<tr>
<th>SciPy</th>
<th>CuPy</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>scipy.special.clpmn</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.comb</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.cosdg</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.cosm1</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.cotdg</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.dawsn</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.digamma</code></td>
<td><code>cupyx.scipy.special.digamma</code></td>
</tr>
<tr>
<td><code>scipy.special.dirc</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.ellip_harm</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.ellip_harm_2</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.ellip_normal</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.ellipe</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.ellipeinc</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.ellipj</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.ellipk</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.ellipkinc</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.ellipkm1</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.entr</code></td>
<td><code>cupyx.scipy.special.entr</code></td>
</tr>
<tr>
<td><code>scipy.special.erf</code></td>
<td><code>cupyx.scipy.special.erf</code></td>
</tr>
<tr>
<td><code>scipy.special.erf_zeros</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.erfc</code></td>
<td><code>cupyx.scipy.special.erfc</code></td>
</tr>
<tr>
<td><code>scipy.special.erfcinv</code></td>
<td><code>cupyx.scipy.special.erfcinv</code></td>
</tr>
<tr>
<td><code>scipy.special.erfcx</code></td>
<td><code>cupyx.scipy.special.erfcx</code></td>
</tr>
<tr>
<td><code>scipy.special.erfi</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.erfinv</code></td>
<td><code>cupyx.scipy.special.erfinv</code></td>
</tr>
<tr>
<td><code>scipy.special.euler</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.eval_chebyc</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.eval_chebys</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.eval_chebyt</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.eval_chebyu</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.eval_gegenbauer</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.eval_genlaguerre</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.eval_hermite</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.eval_hermitenorm</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.eval_jacobi</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.eval_laguerre</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.eval_legendre</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.eval_sh_chebyt</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.eval_sh_chebyu</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.eval_sh_jacobi</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.eval_sh_legendre</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.expl</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.exp10</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.exp2</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.expl</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.expit</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.expm1</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.expn</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.exprel</code></td>
<td>-</td>
</tr>
</tbody>
</table>

Continued on next page
Table 123 – continued from previous page

<table>
<thead>
<tr>
<th>SciPy</th>
<th>CuPy</th>
</tr>
</thead>
<tbody>
<tr>
<td>scipy.special.factorial</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.factorial2</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.factorialk</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.fdtr</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.fdtrc</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.fdtri</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.fdtridfd</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.fresnel</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.fresnel_zeros</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.fresnelc_zeros</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.fresnels_zeros</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.gamma</td>
<td>cupyx.scipy.special.gamma</td>
</tr>
<tr>
<td>scipy.special.gammainc</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.gammaincnc</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.gammaincncinv</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.gammaincinv</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.gammaian</td>
<td>cupyx.scipy.special.gammaian</td>
</tr>
<tr>
<td>scipy.special.gammaasgn</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.gdtr</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.gdtrc</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.gdtria</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.gdtrib</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.gdtrix</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.gegenbauer</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.genlaquerre</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.geterr</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.h1vp</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.h2vp</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.h_roots</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.hankel1</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.hankel1e</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.hankel2</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.hankel2e</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.he_roots</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.hermite</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.hermitenorm</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.huber</td>
<td>cupyx.scipy.special.huber</td>
</tr>
<tr>
<td>scipy.special.hyp0f1</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.hyp1f1</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.hyp2f1</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.hyperu</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.i0</td>
<td>cupyx.scipy.special.i0</td>
</tr>
<tr>
<td>scipy.special.i0e</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.i1</td>
<td>cupyx.scipy.special.i1</td>
</tr>
<tr>
<td>scipy.special.inv_boxcox</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.inv_boxcox1p</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.it2i0k0</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.it2j0y0</td>
<td>-</td>
</tr>
</tbody>
</table>

Continued on next page
Table 123 – continued from previous page

<table>
<thead>
<tr>
<th>Scipy</th>
<th>CuPy</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>scipy.special.it2struve0</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.itairy</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.it10k0</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.itj0y0</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.itmodstruve0</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.itstruve0</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.iv</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.ive</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.ivp</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.j0</code></td>
<td><code>cupyx.scipy.special.j0</code></td>
</tr>
<tr>
<td><code>scipy.special.j1</code></td>
<td><code>cupyx.scipy.special.j1</code></td>
</tr>
<tr>
<td><code>scipy.special.j_roots</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.jacobi</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.jn</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.jn_zeros</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.jnjnp_zeros</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.jnp_zeros</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.jnyn_zeros</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.js_roots</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.jv</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.jve</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.jvp</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.k0</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.k0e</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.k1</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.k1e</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.kei</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.kei_zeros</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.keip</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.keip_zeros</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.kelvin</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.kelvin_zeros</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.ker</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.ker_zeros</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.kerp</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.kerp_zeros</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.kl_div</code></td>
<td><code>cupyx.scipy.special.kl_div</code></td>
</tr>
<tr>
<td><code>scipy.special.kn</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.kolmogi</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.kolmogorov</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.kv</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.kve</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.kvp</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.l_roots</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.la_roots</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.laguerre</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.lambertw</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.legendre</code></td>
<td>-</td>
</tr>
<tr>
<td><code>scipy.special.lmbda</code></td>
<td>-</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>SciPy</th>
<th>CuPy</th>
</tr>
</thead>
<tbody>
<tr>
<td>scipy.special.log1p</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.log_ndtr</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.log_softmax</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.loggamma</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.logit</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.logsumexp</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.ipmn</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.ipmv</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.ipn</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.lqmn</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.lqn</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.mathieu_a</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.mathieu_b</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.mathieu_cem</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.mathieu_even_coef</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.mathieu_modcem1</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.mathieu_modcem2</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.mathieu_modsem1</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.mathieu_modsem2</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.mathieu_odd_coef</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.mathieu_sem</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.mathieu_odd</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.modfresneim</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.modfresnelp</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.modstruve</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.multigammaln</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.nbdtr</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.nbdtrc</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.nbdtri</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.nbdtrik</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.nbdtrin</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.ncfdtr</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.ncfdtri</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.ncfdtridfd</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.ncfdtridfn</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.ncfdtrinc</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.nctdtr</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.nctdtridf</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.nctdtrinc</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.nctdtrit</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.ndtr</td>
<td>cupyx.scipy.special.ndtr</td>
</tr>
<tr>
<td>scipy.special.ndtri</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.nrdtrimn</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.nrdtrisd</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.obl_ang1</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.obl_ang1_cv</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.obl_cv</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.obl_cv_seq</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.obl_rad1</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.obl_rad1_cv</td>
<td>-</td>
</tr>
</tbody>
</table>
Table 123 – continued from previous page

<table>
<thead>
<tr>
<th>SciPy</th>
<th>CuPy</th>
</tr>
</thead>
<tbody>
<tr>
<td>scipy.special.obl_rad2</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.obl_rad2_cv</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.owens_t</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.p_roots</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.pbdn_seq</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.pbdv</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.pbdv_seq</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.pbvv</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.pbvv_seq</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.pbwa</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.pdtr</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.pdtc</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.pdti</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.pdtrik</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.perm</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.poch</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.polygamma</td>
<td>cupyx.scipy.special.polygamma</td>
</tr>
<tr>
<td>scipy.special.pro_ang1</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.pro_ang1_cv</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.pro_cv</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.pro_cv_seq</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.pro_rad1</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.pro_rad1_cv</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.pro_rad2</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.pro_rad2_cv</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.ps_roots</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.pseudo_huber</td>
<td>cupyx.scipy.special.pseudo_huber</td>
</tr>
<tr>
<td>scipy.special.psi</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.radian</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.rel_entr</td>
<td>cupyx.scipy.special.rel_entr</td>
</tr>
<tr>
<td>scipy.special.rgamma</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.riccati_jn</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.riccati_yn</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.roots_chebyc</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.roots_chebys</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.roots_chebyt</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.roots_chebyu</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.roots_gegenbauer</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.roots_genlaguerre</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.roots_hermite</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.roots_hermitegener</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.roots_jacobi</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.roots_laguerre</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.roots_legendre</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.roots_sh_chebyt</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.roots_sh_chebyu</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.roots_sh_jacobi</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.roots_sh_legendre</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.round</td>
<td>-</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>SciPy</th>
<th>CuPy</th>
</tr>
</thead>
<tbody>
<tr>
<td>scipy.special.s_roots</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.seterr</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.sh_chebyt</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.sh_chebyu</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.sh_jacobi</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.sh_legendre</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.shichi</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.sici</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.sinc</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.smindg</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.smirnov</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.smirnovi</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.softmax</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.spence</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.sph_harm</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.spherical_in</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.spherical_jn</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.spherical_kn</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.spherical_yn</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.stdtr</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.stdtridf</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.stdtrit</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.struve</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.t_roots</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.tandg</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.tklmbda</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.ts_roots</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.us_roots</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.voigt_profile</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.wofz</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.wrightomega</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.xlog1py</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.xlogy</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.y0</td>
<td>cupyx.scipy.special.y0</td>
</tr>
<tr>
<td>scipy.special.y0_zeros</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.y1</td>
<td>cupyx.scipy.special.y1</td>
</tr>
<tr>
<td>scipy.special.y1_zeros</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.ylp_zeros</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.yn</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.yn_zeros</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.ynp_zeros</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.yv</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.yve</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.yvp</td>
<td>-</td>
</tr>
<tr>
<td>scipy.special.zeta</td>
<td>cupyx.scipy.special.zeta</td>
</tr>
<tr>
<td>scipy.special.zetac</td>
<td>-</td>
</tr>
</tbody>
</table>
4.17 Miscellaneous functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cupy.may_share_memory</td>
<td>Checks if memory between two arrays is shared.</td>
</tr>
<tr>
<td>cupy.shares_memory</td>
<td>Checks if memory between two arrays is shared.</td>
</tr>
<tr>
<td>cupy.show_config</td>
<td>Prints the current runtime configuration to standard output.</td>
</tr>
<tr>
<td>cupy.who</td>
<td>Prints the CuPy arrays in the given dictionary.</td>
</tr>
</tbody>
</table>

4.17.1 cupy.may_share_memory

cupy.may_share_memory(a, b, max_work=None)

4.17.2 cupy.shares_memory

cupy.shares_memory(a, b, max_work=None)

4.17.3 cupy.show_config

cupy.show_config()

Prints the current runtime configuration to standard output.

4.17.4 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**

```python
>>> a = cupy.arange(10)
>>> b = cupy.ones(20)
>>> cupy.who()

<table>
<thead>
<tr>
<th>Name</th>
<th>Shape</th>
<th>Bytes</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>10</td>
<td>80</td>
<td>int64</td>
</tr>
<tr>
<td>b</td>
<td>20</td>
<td>160</td>
<td>float64</td>
</tr>
</tbody>
</table>

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)

<table>
<thead>
<tr>
<th>Name</th>
<th>Shape</th>
<th>Bytes</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>txt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>idx</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

(continues on next page)
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>2</td>
<td>16</td>
<td>float64</td>
</tr>
<tr>
<td>y</td>
<td>3</td>
<td>24</td>
<td>float64</td>
</tr>
</tbody>
</table>

Upper bound on total bytes = 40
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.

5.1 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 addition and extension to the APIs keeping the supported backward compatibility.
- **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.

5.2 Processes to Break Backward Compatibilities

5.2.1 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 deprecation warning (if possible). There should be another way to reimplement the same things previously written with 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 the future warning (if possible).

The actual dropping should be done through the following steps:

- Make the API deprecated. At this point, users should not need 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.

### 5.2.2 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.

### 5.3 Supported Backward Compatibility

This section defines backward compatibilities that minor updates must maintain.

#### 5.3.1 Documented Interface

CuPy has the 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.

#### 5.3.2 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.

#### 5.3.3 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.

5.3.4 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 the codes depend on how the attributes and properties are implemented.

5.3.5 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 the codes depend on how the methods and callable attributes are implemented.

5.3.6 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.

5.4 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.
CHAPTER
SIX

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 User Group
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.

<table>
<thead>
<tr>
<th>Date</th>
<th>ver X</th>
<th>ver Y</th>
<th>ver Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 weeks</td>
<td>$X.0.0rc1$</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>4 weeks</td>
<td>$X.0.0$</td>
<td>$Y.0.0a1$</td>
<td>–</td>
</tr>
<tr>
<td>8 weeks</td>
<td>$X.1.0*$</td>
<td>$Y.0.0b1$</td>
<td>–</td>
</tr>
<tr>
<td>12 weeks</td>
<td>$X.2.0*$</td>
<td>$Y.0.0rc1$</td>
<td>–</td>
</tr>
<tr>
<td>16 weeks</td>
<td>–</td>
<td>$Y.0.0$</td>
<td>$Z.0.0a1$</td>
</tr>
</tbody>
</table>

(* 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.
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 Travis CI for Linux and Mac OS X, and on AppVeyor for Windows. Your PR needs to pass at least the test for Linux on Travis CI. After the automatic test passes, some of the 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 Travis-CI. 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 build, 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`:

```python
import unittest
from cupy.testing import attr

class TestMyFunc(unittest.TestCase):
    ...

    @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:

```python
import unittest
from cupy.testing import attr

class TestMyFunc(unittest.TestCase):
    ...
```

(continues on next page)
@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:

```python
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`:

```bash
$ pip install sphinx sphinx_rtd_theme
```

Then you can build the documentation in HTML format locally:

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

### 7.1 Requirements

- AMD GPU supported by ROCm
- **ROCm**: v3.5+
  - See the ROCm Installation Guide for details.

The following ROCm libraries are required:

```bash
$ sudo apt install hipblas hipsparse rocsparse rocrand rocthrust rocsolver rocfft
˓→ hipcub rocprim
```

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

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

### 7.2 Building CuPy for ROCm

Currently, you need to build CuPy from source to run on AMD GPU.

```bash
$ export HCC_AMDGPU_TARGET=gfx900  # This value should be changed based on your GPU
$ export __HIP_PLATFORM_HCC__
$ export CUPY_INSTALL_USE_HIP=1
$ pip install cupy
```

Note that `HCC_AMDGPU_TARGET` must be set to the ISA name supported by your GPU. Run `rocminfo` and use the value displayed in `Name:` line (e.g., `gfx900`).

You may also need to set `ROCM_HOME` (e.g., `ROCM_HOME=/opt/rocm`).
This is a list of changes introduced in each release that users should be aware of when migrating from older versions.

8.1 CuPy v8

8.1.1 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.

8.1.2 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.

8.1.3 Update of Docker Images

- CuPy official Docker images (see Installation Guide for details) are now updated to use CUDA 10.2 and Python 3.6.
- SciPy and Optuna are now pre-installed.

8.1.4 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 Guide 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.

8.1.5 API Changes

- cupy.scatter_add, which was deprecated in CuPy v4, has been removed. Use cupyx.scatter_add() instead.
CuPy Documentation, Release 8.2.0

- `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.

8.2 CuPy v7

8.2.1 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 Guide`.

8.3 CuPy v6

8.3.1 Binary Packages Ignore `LD_LIBRARY_PATH`

Prior to CuPy v6, `LD_LIBRARY_PATH` environment variable can be used to overrides 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.

8.4 CuPy v5

8.4.1 `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.

8.4.2 Dropped Support for CUDA 7.0 / 7.5

CuPy v5 no longer supports CUDA 7.0 / 7.5.
8.4.3 Update of Docker Images

CuPy official Docker images (see Installation Guide 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.

8.5 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.

8.5.1 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.

```python
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.

```python
import cupy
cupy.cuda.set_allocator(None)
cupy.cuda.set_pinned_memory_allocator(None)
```
8.5.2 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.

8.5.3 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.

8.5.4 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.

8.5.5 Update of Docker Images

CuPy official Docker images (see Installation Guide 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.

8.6 CuPy v2

8.6.1 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.
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.

9.1 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
9.2 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, 78
cupy.random, 151
cupy.testing, 350
cupyx, 197
cupyx.optimizing, 346
cupyx.scipy, 200
cupyx.scipy.fft, 200
cupyx.scipy.fftpack, 209
cupyx.scipy.ndimage, 217
cupyx.scipy.sparse, 244
Symbols

__bool__() (cupyx.scipy.sparse.coo_matrix method), 251
__bool__() (cupyx.scipy.sparse.csc_matrix method), 259
__bool__() (cupyx.scipy.sparse.csr_matrix method), 268
__bool__() (cupyx.scipy.sparse.dia_matrix method), 273
__bool__() (cupyx.scipy.sparse.spmatrix method), 277
__call__() (cupy.ElementwiseKernel method), 339
__call__() (cupy.RawKernel method), 342
__call__() (cupy.ReductionKernel method), 341
__call__() (cupy.poly1d method), 149
__call__() (cupy.prof.TimeRangeDecorator method), 364
__copy__() (cupy.ndarray method), 21
__enter__() (cupy.cuda.Device method), 300
__enter__() (cupy.cuda.ExternalStream method), 318
__enter__() (cupy.cuda.MemoryHook method), 312
__enter__() (cupy.cuda.Stream method), 317
__enter__() (cupy.cuda.memory_hooks.DebugPrintHook method), 314
__enter__() (cupy.cuda.memory_hooks.LineProfileHook method), 316
__exit__() (cupy.cuda.memory_hooks.LineProfileHook method), 316
__exit__() (cupy.prof.TimeRangeDecorator method), 364
__ge__() (cupyx.scipy.sparse.coo_matrix method), 251
__ge__() (cupyx.scipy.sparse.csc_matrix method), 259
__ge__() (cupyx.scipy.sparse.csr_matrix method), 268
__ge__() (cupyx.scipy.sparse.dia_matrix method), 273
__ge__() (cupyx.scipy.sparse.spmatrix method), 277
__getitem__() (cupy.flatiter method), 100
__getitem__() (cupy.ndarray method), 20
__getitem__() (cupy.poly1d method), 149
__getitem__() (cupy.prof.TimeRangeDecorator method), 364
__gt__() (cupyx.scipy.sparse.coo_matrix method), 251
__gt__() (cupyx.scipy.sparse.csc_matrix method), 259
__gt__() (cupyx.scipy.sparse.csr_matrix method), 268
__gt__() (cupyx.scipy.sparse.dia_matrix method), 273
__gt__() (cupyx.scipy.sparse.spmatrix method), 277
__iter__() (cupy.flatiter method), 100
__iter__() (cupy.ndarray method), 21
__iter__() (cupy.poly1d method), 149
__iter__() (cupy.prof.TimeRangeDecorator method), 364
__le__() (cupyx.scipy.sparse.coo_matrix method), 251
__le__() (cupyx.scipy.sparse.csc_matrix method), 259
__le__() (cupyx.scipy.sparse.csr_matrix method), 268
__le__() (cupyx.scipy.sparse.dia_matrix method), 273
__le__() (cupyx.scipy.sparse.spmatrix method), 277

INDEX

421
CuPy Documentation, Release 8.2.0

_\texttt{__len__}() (cupy.scipy.sparse.coo_matrix method), 100
_\texttt{__len__}() (cupy.ndarray method), 21
_\texttt{__len__}() (cupy.poly1d method), 149
_\texttt{__len__}() (cupyx.scipy.sparse.coo_matrix method), 245
_\texttt{__len__}() (cupyx.scipy.sparse.csc_matrix method), 252
_\texttt{__len__}() (cupyx.scipy.sparse.csr_matrix method), 260
_\texttt{__len__}() (cupyx.scipy.sparse.dia_matrix method), 269
_\texttt{__len__}() (cupyx.scipy.sparse.spmatrix method), 274
_\texttt{__lt__}() (cupyx.scipy.sparse.coo_matrix method), 250
_\texttt{__lt__}() (cupyx.scipy.sparse.csc_matrix method), 259
_\texttt{__lt__}() (cupyx.scipy.sparse.csr_matrix method), 267
_\texttt{__lt__}() (cupyx.scipy.sparse.dia_matrix method), 273
_\texttt{__lt__}() (cupyx.scipy.sparse.spmatrix method), 277
_\texttt{__next__}() (cupyx.cuda.memory_hooks.DebugPrintHook method), 100
_\texttt{__nonzero__}() (cupyx.scipy.sparse.coo_matrix method), 251
_\texttt{__nonzero__}() (cupyx.scipy.sparse.csc_matrix method), 259
_\texttt{__nonzero__}() (cupyx.scipy.sparse.csr_matrix method), 267
_\texttt{__nonzero__}() (cupyx.scipy.sparse.dia_matrix method), 273
_\texttt{__nonzero__}() (cupyx.scipy.sparse.spmatrix method), 277
_\texttt{__next__}() (cupyx.cuda.memory_hooks.DebugPrintHook method), 100
_\texttt{__nonzero__}() (cupyx.scipy.sparse.spmatrix method), 277
_\texttt{__next__}() (cupyx.cuda.memory_hooks.DebugPrintHook method), 100
_\texttt{arcsin()} (cupyx.scipy.sparse.coo_matrix method), 251
_\texttt{arcsin()} (cupyx.scipy.sparse.csc_matrix method), 259
_\texttt{arcsin()} (cupyx.scipy.sparse.csr_matrix method), 267
_\texttt{arcsin()} (cupyx.scipy.sparse.dia_matrix method), 273
_\texttt{arcsin()} (cupyx.scipy.sparse.spmatrix method), 277
_\texttt{abort()} (cupyx.cuda.nccl.NcclCommunicator method), 329
_\texttt{absolute} (in module cupy), 36
_\texttt{add} (in module cupy), 34
_\texttt{add_callback} (cupy.cuda.ExternalStream attribute), 319
_\texttt{add_callback} (cupy.cuda.Stream attribute), 317
_\texttt{affine_transform()} (in module cupyx.scipy.ndimage), 235
_\texttt{all()} (cupy.ndarray method), 21
_\texttt{all()} (in module cupy), 126
_\texttt{allclose()} (in module cupy), 131
_\texttt{allGather()} (cupyx.cuda.nccl.NcclCommunicator method), 329
_\texttt{alloc()} (in module cupy), 269
_\texttt{alloc_pinned_memory()} (in module cupy), 307
_\texttt{alloc_postprocess} (cupy.cuda.memory_hooks.LineProfileHook attribute), 316
_\texttt{alloc_postprocess} (cupy.cuda.MemoryHook attribute), 312
_\texttt{alloc_postprocess()} (cupy.cuda.memory_hooks.LineProfileHook attribute), 314
_\texttt{alloc_preprocess} (cupy.cuda.memory_hooks.DebugPrintHook method), 314
_\texttt{alloc_preprocess} (cupy.cuda.MethodHook attribute), 312
_\texttt{alloc_preprocess()} (cupy.cuda.memory_hooks.LineProfileHook method), 316
_\texttt{allow_synchronize()} (in module cupy), 365
_\texttt{allReduce()} (cupyx.cuda.nccl.NcclCommunicator method), 329
_\texttt{amax()} (in module cupy), 190
_\texttt{amin()} (in module cupy), 189
_\texttt{angle()} (in module cupy), 139
_\texttt{arange()} (in module cupy), 149
_\texttt{arccos} (in module cupy), 40
_\texttt{arccosh} (in module cupy), 41
_\texttt{arcsin} (in module cupy), 40
_\texttt{arcsin()} (cupyx.scipy.sparse.coo_matrix method), 251
_\texttt{arcsin()} (cupyx.scipy.sparse.csc_matrix method), 252
_\texttt{arcsin()} (cupyx.scipy.sparse.csr_matrix method), 261
_\texttt{arcsin()} (cupyx.scipy.sparse.dia_matrix method), 269
_\texttt{arcsin()} (in module cupy), 41
_\texttt{arcsinh} (cupyx.scipy.sparse.coo_matrix method), 251
_\texttt{arcsinh} (cupyx.scipy.sparse.csc_matrix method), 252
_\texttt{arcsinh} (cupyx.scipy.sparse.csr_matrix method), 261
_\texttt{arcsinh} (cupyx.scipy.sparse.dia_matrix method), 269
_\texttt{arcsinh} (in module cupy), 41
bitwise_and (in module cupy), 42
bitwise_not (in module cupy), 76
bitwise_or (in module cupy), 42
bitwise_xor (in module cupy), 43
blackman() (in module cupy), 142
block_diag() (in module cupyx.scipy.linalg), 122
bmat() (in module cupyx.scipy.sparse), 278
broadcast (class in cupy), 65
broadcast() (cupy.cudnn.NcclCommunicator method), 329
broadcast_arrays() (in module cupy), 66
broadcast_to() (in module cupy), 66
bytes() (in module cupy.random), 156

C
c (cupy.poly1d attribute), 149
c (in module cupy), 89
cache_mode_ca (cupy.RawKernel attribute), 343
can_cast() (in module cupy), 77
cbrt (in module cupy), 141
cell (in module cupy), 49
cell() (cupyx.scipy.sparse.coo_matrix method), 246
cell() (cupyx.scipy.sparse.csc_matrix method), 253
cell() (cupyx.scipy.sparse.csr_matrix method), 262
cell() (cupyx.scipy.sparse.dia_matrix method), 269
ChannelFormatDescriptor (class in cupyx.cudarray), 321
chDesc (cupy.cuda.texture.ResourceDescriptor attribute), 324
check_async_error()
  (cupy.cudnn.NcclCommunicator method), 329
chisquare() (cupy.random.RandomState method), 176
chisquare() (in module cupy.random), 159
choice() (cupy.random.RandomState method), 176
choice() (in module cupy.random), 156
cholesky() (in module cupy.linalg), 108
choose() (cupy.ndarray method), 22
choose() (in module cupy), 94
choose_conv_method() (in module cupyx.scipy.signal), 291
circulant() (in module cupyx.scipy.linalg), 121
clear_memo() (in module cupy), 338
clip() (cupy.ndarray method), 22
clip() (in module cupy), 141
code (cupy.RawKernel attribute), 343
code (cupy.RawModule attribute), 346
coeff (cupy.poly1d attribute), 149
coefficients (cupy.poly1d attribute), 149
coeffs (cupy.poly1d attribute), 149
column_stack() (in module cupy), 69
common_type() (in module cupy), 78
companion() (in module cupyx.scipy.linalg), 123
compile() (cupy.RawKernel method), 342
compile() (cupy.RawModule method), 345
compress() (cupy.ndarray method), 22
compress() (in module cupy), 94
compute_capability (cupy.cuda.Device attribute), 301
concatenate() (in module cupy), 68
conj (in module cupy), 140
conj() (cupy.ndarray method), 23
conj() (cupyx.scipy.sparse.coo_matrix method), 246
conj() (cupyx.scipy.sparse.csc_matrix method), 253
conj() (cupyx.scipy.sparse.csr_matrix method), 262
conj() (cupyx.scipy.sparse.dia_matrix method), 269
conj() (cupyx.scipy.sparse.spmatrix method), 274
conjugate() (cupy.ndarray method), 23
conjugate()
  (cupyx.scipy.sparse.coo_matrix method), 246
conjugate()
  (cupyx.scipy.sparse.csc_matrix method), 253
conjugate()
  (cupyx.scipy.sparse.csr_matrix method), 262
conjugate()
  (cupyx.scipy.sparse.dia_matrix method), 270
conjugate()
  (cupyx.scipy.sparse.spmatrix method), 274
const_size_bytes (cupy.RawKernel attribute), 343
convolution_matrix() (in module cupyx.scipy.linalg), 125
convolve() (in module cupy), 140
convolve() (in module cupyx.scipy.ndimage), 218
convolve() (in module cupyx.scipy.signal), 291
convolve1d() (in module cupyx.scipy.ndimage), 218
convolve2d() (in module cupyx.scipy.signal), 292
coo_matrix (class in cupyx.scipy.sparse), 245
copy() (cupy.flatiter method), 100
copy() (cupy.ndarray method), 23
copy() (cupyx.scipy.sparse.coo_matrix method), 246
copy() (cupyx.scipy.sparse.csc_matrix method), 253
copy() (cupyx.scipy.sparse.csr_matrix method), 262
copy() (cupyx.scipy.sparse.dia_matrix method), 270
copy() (cupyx.scipy.sparse.spmatrix method), 275
copy() (in module cupy), 55
copy_from() (cupy.cuda.MemcpyPointer method), 304
copy_from()
  (cupy.cuda.texture.CUDArray method), 322
copy_from_async() (cupy.cuda.MemcpyPointer method), 304
copy_from_device() (cupy.cuda.MemcpyPointer method), 305
copy_from_device_async()
  (cupy.cuda.MemcpyPointer method), 305
copy_from_host() (cupy.cuda.MemcpyPointer method), 305
copy_from_host_async()  (cupy.cuda.MemoryPointer method), 305
 copy_to()  (cupy.cuda.texture.CUDAarray method), 322
 copy_to_host()  (cupy.cuda.MemoryPointer method), 305
 copy_to_host_async()  (cupy.cuda.MemoryPointer method), 305
copiesign  (in module cupy), 47
copyto()  (in module cupy), 60
correlate()  (in module cupy), 196
correlate()  (in module cupyx.scipy.ndimage), 219
correlate()  (in module cupyx.scipy.signal), 293
correlate1d()  (in module cupyx.scipy.ndimage), 220
correlate2d()  (in module cupyx.scipy.signal), 294
cos  (in module cupy), 39
cosh  (in module cupy), 41
count_nonzero()  (cupyx.scipy.sparse.coo_matrix method), 246
count_nonzero()  (cupyx.scipy.sparse.csc_matrix method), 253
count_nonzero()  (cupyx.scipy.sparse.csr_matrix method), 262
count_nonzero()  (cupyx.scipy.sparse.dia_matrix method), 270
count_nonzero()  (cupyx.scipy.sparse.spmatrix method), 275
count_nonzero()  (in module cupy), 188
cov()  (in module cupy), 197
cross()  (in module cupy), 104
csc_matrix  (class in cupyx.scipy.sparse), 251
csr_matrix  (class in cupyx.scipy.sparse), 260
cstruct  (cupy.ndarray attribute), 29
cuArr  (cupy.cuda.texture.ResourceDescriptor attribute), 324
cublas_handle  (cupy.cuda.Device attribute), 301
cudadarray  (class in cupy.cuda.texture), 321
cumprod()  (cupy.ndarray method), 23
cumprod()  (in module cupy), 136
cumsum()  (cupy.ndarray method), 23
cumsum()  (in module cupy), 136
cupy  (module), 1
cupy.fft  (module), 78
cupy.random  (module), 151
cupy.testing  (module), 350
cupyx  (module), 197
cupyx.optimizing  (module), 346
cupyx.scipy  (module), 200
cupyx.scipy.fft  (module), 200
cupyx.scipy.fftpack  (module), 209
cupyx.scipy.ndimage  (module), 217
cupyx.scipy.sparse  (module), 244
cusolver_handle  (cupy.cuda.Device attribute), 301
cusolver_sp_handle  (cupy.cuda.Device attribute), 301
cusparse_handle  (cupy.cuda.Device attribute), 301
data  (cupy.ndarray attribute), 29
debugPrintHook  (class in cupy.cuda.memory_hooks), 313
deg2rad  (in module cupy), 42
deg2rad()  (cupyx.scipy.sparse.coo_matrix method), 247
deg2rad()  (cupyx.scipy.sparse.csc_matrix method), 254
deg2rad()  (cupyx.scipy.sparse.csr_matrix method), 262
deg2rad()  (cupyx.scipy.sparse.dia_matrix method), 270
degrees  (in module cupy), 133
depth  (cupy.cuda.texture.CUDAarray attribute), 322
deriv()  (cupy.poly1d method), 149
desc  (cupy.cuda.texture.CUDAArray attribute), 322
destroy()  (cupy.cuda.nnc1.NcclCommunicator method), 329
det()  (in module cupy.linalg), 111
device  (class in cupy.cuda), 300
device  (cupy.cuda.Device attribute), 303
device  (cupy.cuda.MemoryPointer attribute), 306
device  (cupy.cuda.UnownedMemory attribute), 303
device  (cupy.ndarray attribute), 29
device  (cupyx.scipy.sparse.coo_matrix attribute), 251
device  (cupyx.scipy.sparse.csc_matrix attribute), 259
device  (cupyx.scipy.sparse.csr_matrix attribute), 268
device  (cupyx.scipy.sparse.dia_matrix attribute), 273
device  (cupyx.scipy.sparse.spmatrix attribute), 277
device_id  (cupy.cuda.Memory attribute), 303
device_id  (cupy.cuda.MemoryPointer attribute), 306
device_id  (cupy.cuda.UnownedMemory attribute), 303
device_id()  (cupy.cuda.nnc1.NcclCommunicator method), 329
deviceCanAccessPeer()  (in module cupy.cuda.runtime), 333
deviceEnablePeerAccess()  (in module cupy.cuda.runtime), 333
deviceGetAttribute()  (in module cupy.cuda.runtime), 333
deviceGetByPCIBusId()  (in module cupy.cuda.runtime), 333
deviceGetLimit()  (in module cupy.cuda.runtime), 333
deviceSetLimit()  (in module cupy.cuda.runtime), 333
deviceSynchronize() (in module cupy.cuda.runtime), 333
DeviceSynchronized, 365
dft() (in module cupy.scipy.linalg), 124
dia_matrix (class in cupy.scipy.sparse), 269
diag() (in module cupy), 58
diag_indices() (in module cupy), 94
diag_indices_from() (in module cupy), 95
diagflat() (in module cupy), 59
diagonal() (cupy.ndarray method), 23
diagonal() (cupy.scipy.sparse.coo_matrix method), 247
diagonal() (cupy.scipy.sparse.csc_matrix method), 254
diagonal() (cupy.scipy.sparse.csr_matrix method), 262
diagonal() (cupy.scipy.sparse.dia_matrix method), 270
diagonal() (cupy.scipy.sparsespmatrix method), 275
diagonal() (in module cupy), 95
diags() (in module cupy.scipy.sparse), 278
diff() (in module cupy), 137
digamma (in module cupy.scipy.special), 289
dirichlet() (cupy.random.RandomState method), 176
dirichlet() (in module cupy.random), 159
divide (in module cupy), 35
divmod (in module cupy), 139
done (cupy.cuda.Event attribute), 320
done (cupy.cuda.ExternalStream attribute), 319
done (cupy.cuda.Stream attribute), 318
dot() (cupy.ndarray method), 23
dot() (cupy.scipy.sparse.coo_matrix method), 247
dot() (cupy.scipy.sparse.csc_matrix method), 254
dot() (cupy.scipy.sparse.csr_matrix method), 263
dot() (cupy.scipy.sparse.dia_matrix method), 270
dot() (cupy.scipy.sparse.spmatrix method), 275
dot() (in module cupy), 104
driverGetVersion() (in module cupy.cuda.runtime), 332
dsplit() (in module cupy), 71
dstack() (in module cupy), 69
dtype (cupy.ndarray attribute), 29
dtype (cupy.scipy.sparse.coo_matrix attribute), 251
dtype (cupy.scipy.sparse.csr_matrix attribute), 259
dtype (cupy.scipy.sparse.csc_matrix attribute), 268
dtype (cupy.scipy.sparse.dia_matrix attribute), 273
dump() (cupy.ndarray method), 23
dumps() (cupy.ndarray method), 23

eigh() (in module cupy.linalg), 110
eigvalsh() (in module cupy.linalg), 110
einsum() (in module cupy.linalg), 107
ElementwiseKernel (class in cupy), 339
eliminate_zeros() (cupy.scipy.sparse.coo_matrix method), 247
eliminate_zeros() (cupy.scipy.sparse.csc_matrix method), 254
eliminate_zeros() (cupy.scipy.sparse.csr_matrix method), 263
empty() (in module cupy), 50
empty_like() (in module cupy), 50
enable Cooperative Groups (cupy.RawKernel attribute), 343
enable Cooperative Groups (cupy.RawModule attribute), 346
entr (in module cupy.scipy.special), 287
equal (in module cupy), 45
erf (in module cupy.scipy.special), 289
erfc (in module cupy.scipy.special), 289
erfcinv (in module cupy.scipy.special), 290
erfcx (in module cupy.scipy.special), 290
erfinv (in module cupy.scipy.special), 290
Event (class in cupy.cuda), 319
eventCreate() (in module cupy.cuda.runtime), 337
eventCreateWithFlags() (in module cupy.cuda.runtime), 337
eventDestroy() (in module cupy.cuda.runtime), 337
eventElapsedTime() (in module cupy.cuda.runtime), 337
eventQuery() (in module cupy.cuda.runtime), 337
eventRecord() (in module cupy.cuda.runtime), 337
eventSynchronize() (in module cupy.cuda.runtime), 337
exp (in module cupy), 37
exp2 (in module cupy), 37
expand_dims() (in module cupy), 66
expm1 (in module cupy), 38
expm1() (cupy.scipy.sparse.coo_matrix method), 247
expm1() (cupy.scipy.sparse.csc_matrix method), 254
expm1() (cupy.scipy.sparse.csr_matrix method), 263
expm1() (cupy.scipy.sparse.csc_matrix method), 270
expm1() (cupy.scipy.sparse.dia_matrix method), 270
exponential() (cupy.random.RandomState method), 176
exponential() (in module cupy.random), 160
ExternalStream (class in cupy.cuda), 318
extract() (in module cupy), 96
eye() (in module cupy), 51
eye() (in module cupy.scipy.sparse), 279

F
f() (cupy.random.RandomState method), 176
f() (in module cupy.random), 160
fft() (in module cupy.fft), 79
fft() (in module cupy.scipy.fft), 201
fft() (in module cupy.scipy.fftpack), 210
fft2() (in module cupy.fft), 80
fft2() (in module cupyx.scipy.fft), 202
fft2() (in module cupyx.scipy.fft.packed), 211
fftn() (in module cupy.fft), 80
fftn() (in module cupyx.scipy.fft), 203
fftn() (in module cupyx.scipy.fft.packed), 212
fftfshift() (in module cupy.fft), 86
fiedler() (in module cupyx.scipy.linalg), 124
fiedler_companion() (in module cupyx.scipy.linalg), 125
file_path (cupy.RawKernel attribute), 343
file_path (cupy.RawModule attribute), 346
fill() (cupy.ndarray method), 23
fill_diagonal() (in module cupy), 99
fix (in module cupy), 135
flags (cupy.cuda.texture.CUDAarray attribute), 322
flags (cupy.ndarray attribute), 29
flat (cupy.ndarray attribute), 29
flatiter (class in cupy), 100
flatnonzero() (in module cupy), 187
flatten() (cupy.ndarray method), 24
flip() (in module cupy), 74
flipud() (in module cupy), 74
floor (in module cupy), 48
floor() (cupyx.scipy.sparse.coo_matrix method), 247
floor() (cupyx.scipy.sparse.csr_matrix method), 254
floor() (cupyx.scipy.sparse.csc_matrix method), 263
floor() (cupyx.scipy.sparse.dia_matrix method), 270
floor_divide (in module cupy), 35
fmax (in module cupy), 46
fmin (in module cupy), 46
fmod (in module cupy), 36
for_all_dtypes() (in module cupy.testing), 359
for_all_dtypes_combination() (in module cupy.testing), 361
for_CF_orders() (in module cupy.testing), 363
for_complex_dtypes() (in module cupy.testing), 361
for_dtypes() (in module cupy.testing), 358
for_dtypes_combination() (in module cupy.testing), 361
for_float_dtypes() (in module cupy.testing), 359
for_int_dtypes() (in module cupy.testing), 360
for_int_dtypes_combination() (in module cupy.testing), 362
for_orders() (in module cupy.testing), 363
for_signed_dtypes() (in module cupy.testing), 360
for_signed_dtypes_combination() (in module cupy.testing), 362
for_unsigned_dtypes() (in module cupy.testing), 360
for_unsigned_dtypes_combination() (in module cupy.testing), 362
format (cupyx.scipy.sparse.coo_matrix attribute), 251
format (cupyx.scipy.sparse.csc_matrix attribute), 259
format (cupyx.scipy.sparse.csr_matrix attribute), 268
format (cupyx.scipy.sparse.dia_matrix attribute), 274
fourier_gaussian() (in module cupyx.scipy.ndimage), 234
fourier_shift() (in module cupyx.scipy.ndimage), 234
fourier_uniform() (in module cupyx.scipy.ndimage), 234
free() (cupy.cudadMHz.PinnedMemoryPool method), 310
free() (in module cupy.cuda.runtime), 334
free_all_blocks() (cupy.cuda.MemPoi method), 308
free_all_blocks() (cupy.cudadMHz.PinnedMemoryPool method), 310
free_all_free() (cupy.cuda.MemPoi method), 308
free_bytes() (cupy.cuda.MemPoi method), 308
free_postprocess (cupy.cuda.memory_hooks.LineProfileHook attribute), 316
free_postprocess (cupy.cuda.MemPoi attribute), 312
free_postprocess (cupy.cuda.memory_hooks.DebugPrintHook method), 314
free_preprocess (cupy.cuda.memory_hooks.DebugPrintHook attribute), 314
free_preprocess (cupy.cuda.memory_hooks.LineProfileHook attribute), 316
free_preprocess (cupy.cuda.MemPoi attribute), 312
freeArray() (in module cupy.cuda.runtime), 334
freeHost() (in module cupy.cuda.runtime), 334
frexp (in module cupy), 48
from_pci_bus_id() (cupy.cuda.Device method), 300
fromDlpack() (in module cupy), 349
fromfile() (in module cupy), 55
full() (in module cupy), 53
full_like() (in module cupy), 53
fuse() (in module cupy), 346

gamma (in module cupyx.scipy.special), 288
gamma() (cupy.random.RandomState method), 176
gamma() (in module cupy.random), 161
gammaln (in module cupyx.scipy.special), 288
gaussian_filter() (in module cupyx.scipy.ndimage), 220
gaussian_filter1d() (in module cupyx.scipy.ndimage), 221
CuPy Documentation, Release 8.2.0

gaussian_gradient_magnitude() (in module cupyx.scipy.ndimage), 222
gaussian_laplace() (in module cupyx.scipy.ndimage), 222
gcd (in module cupy), 39
generic_filter() (in module cupyx.scipy.ndimage), 223
generic_filter1d() (in module cupyx.scipy.ndimage), 224
generic_gradient_magnitude() (in module cupyx.scipy.ndimage), 225
generic_laplace() (in module cupyx.scipy.ndimage), 225
geometric() (cupy.random.RandomState method), 176
geometric() (in module cupy.random), 161
glearch() (cupy.ndarray method), 30
get() (cupy.poly1d method), 149
get() (cupyx.scipy.sparse.coo_matrix method), 247
get() (cupyx.scipy.sparse.csc_matrix method), 254
get() (cupyx.scipy.sparse.csr_matrix method), 263
get() (cupyx.scipy.sparse.dia_matrix method), 270
getAllocator() (in module cupy), 307
getArray_module() (in module cupy), 30
getArray_module() (in module cupyx.scipy), 30
get_build_version() (in module cupyx.cudnn), 330
get_channel_format() (cupyx.cuda.texture.ChannelFormatDescriptor method), 321
get_current_stream() (in module cupy.cudnn), 319
get_default_memory_pool() (in module cupy), 302
get_default_pinned_memory_pool() (in module cupyx.cudnn), 302
get_elapsed_time() (in module cupy.cudnn), 320
get_fft_plan() (in module cupyx.scipy.fft), 215
get_function() (cupyx.RawModule method), 345
get_global() (cupyx.RawModule method), 345
get_limit() (cupy.cuda.MemoryPool method), 308
get_plan_cache() (in module cupy.fft), 87
get_random_state() (in module cupy.random), 181
get_resource_desc() (cupyx.cuda.texture.ResourceDescriptor method), 323
get_shape() (cupyx.scipy.sparse.coo_matrix method), 247
get_shape() (cupyx.scipy.sparse.csc_matrix method), 254
get_shape() (cupyx.scipy.sparse.csr_matrix method), 263
get_shape() (cupyx.scipy.sparse.dia_matrix method), 271
get_shape() (cupyx.scipy.sparse.spmatrix method), 275
getxtref() (cupyx.RawModule method), 346
get_texture_desc() (cupyx.cuda.texture.TextureDescriptor method), 324
get_unique_id() (in module cupy.cuda.nccl), 330
get_version() (in module cupy.cuda.cudnn), 330
getCol() (cupyx.scipy.sparse.csc_matrix method), 254
getCol() (cupyx.scipy.sparse.csr_matrix method), 263
getDevice() (in module cupyx.cudnn), 332
getDeviceCount() (in module cupyx.cudnn), 333
getformat() (cupyx.scipy.sparse.coo_matrix method), 247
getformat() (cupyx.scipy.sparse.csc_matrix method), 255
getformat() (cupyx.scipy.sparse.csr_matrix method), 263
getformat() (cupyx.scipy.sparse.dia_matrix method), 271
getformat() (cupyx.scipy.sparse.spmatrix method), 275
getH() (cupyx.scipy.sparse.coo_matrix method), 247
getH() (cupyx.scipy.sparse.csc_matrix method), 254
getH() (cupyx.scipy.sparse.csr_matrix method), 263
getH() (cupyx.scipy.sparse.dia_matrix method), 271
getH() (cupyx.scipy.sparse.spmatrix method), 275
getmaxprint() (cupyx.scipy.sparse.coo_matrix method), 247
getmaxprint() (cupyx.scipy.sparse.csc_matrix method), 255
getmaxprint() (cupyx.scipy.sparse.csr_matrix method), 263
getmaxprint() (cupyx.scipy.sparse.dia_matrix method), 271
getmaxprint() (cupyx.scipy.sparse.spmatrix method), 275
getnnz() (cupyx.scipy.sparse.coo_matrix method), 247
getnnz() (cupyx.scipy.sparse.csc_matrix method), 255
getnnz() (cupyx.scipy.sparse.csr_matrix method), 263
getnnz() (cupyx.scipy.sparse.dia_matrix method), 271
getnnz() (cupyx.scipy.sparse.spmatrix method), 275
greater (in module cupy), 44
greater_equal (in module cupy), 44
grey_closing() (in module cupyx.scipy.ndimage), 241
grey_dilation() (in module cupyx.scipy.ndimage), 242
grey_erosion() (in module cupyx.scipy.ndimage), 243
max() (cupyx.scipy.sparse.csr_matrix method), 263
max_dynamic_shared_size_bytes (cupy.RawKernel attribute), 343
max_threads_per_block (cupy.RawKernel attribute), 343
maximum (in module cupy), 46
maximum() (cupyx.scipy.sparse.coo_matrix method), 247
maximum() (cupyx.scipy.sparse.csc_matrix method), 255
maximum() (cupyx.scipy.sparse.csr_matrix method), 264
maximum() (cupyx.scipy.sparse.dia_matrix method), 271
maximum() (cupyx.scipy.sparse.spmatrix method), 275
maximum_filter () (in module cupyx.scipy.ndimage), 227
maximum_filter1d () (in module cupyx.scipy.ndimage), 227
may_share_memory() (in module cupy), 394
mean() (cupy.ndarray method), 24
mean() (cupyx.scipy.sparse.coo_matrix method), 247
mean() (cupyx.scipy.sparse.csc_matrix method), 255
mean() (cupyx.scipy.sparse.csr_matrix method), 264
mean() (cupyx.scipy.sparse.dia_matrix method), 271
mean() (cupyx.scipy.sparse.spmatrix method), 275
mean() (in module cupy), 192
mean() (in module cupyx.scipy.signal), 296
medfilt() (in module cupyx.scipy.signal), 296
median_filter() (in module cupyx.scipy.ndimage), 228
mem (cupy.cuda.MemoryPointer attribute), 306
mem (cupy.cuda.PinnedMemoryPointer attribute), 306
mem_info (cupy.cuda.Device attribute), 301
memAdvise() (in module cupy.cuda.runtime), 336
memcpy() (in module cupy.cuda.runtime), 334
memcpy2D() (in module cupy.cuda.runtime), 335
memcpy2DAsync() (in module cupy.cuda.runtime), 335
memcpy2DFromArray() (in module cupy.cuda.runtime), 335
memcpy2DToArray() (in module cupy.cuda.runtime), 335
memcpy2DToArrayAsync() (in module cupy.cuda.runtime), 335
memcpy2DFromArray() (in module cupy.cuda.runtime), 335
memcpy2DFromArrayAsync() (in module cupy.cuda.runtime), 335
memcpy2DtoArray() (in module cupy.cuda.runtime), 335
memcpy2DtoArrayAsync() (in module cupy.cuda.runtime), 335
memcpy3D() (in module cupy.cuda.runtime), 335
memcpy3DAsync() (in module cupy.cuda.runtime), 336
memcpyAsync() (in module cupy.cuda.runtime), 335
memcpyPeer() (in module cupy.cuda.runtime), 335
memcpyPeerAsync() (in module cupy.cuda.runtime), 335
memGetInfo() (in module cupy.cuda.runtime), 334
memoize() (in module cupy), 338
Memory (class in cupy.cuda), 302
MemoryHook (class in cupy.cuda), 311
MemoryPointer (class in cupy.cuda), 304
MemoryPool (class in cupy.cuda), 308
memPrefetchAsync() (in module cupy.cuda.runtime), 336
memset() (cupy.cuda.MemoryPointer method), 305
memset() (in module cupy.cuda.runtime), 336
memset_async() (cupy.cuda.MemoryPointer method), 306
memsetAsync() (in module cupy.cuda.runtime), 336
meshgrid() (in module cupy), 57
mgrid (in module cupy), 58
min() (cupy.ndarray method), 24
min() (cupyx.scipy.sparse.csc_matrix method), 256
min() (cupyx.scipy.sparse.csr_matrix method), 264
minimum (in module cupy), 46
minimum() (cupyx.scipy.sparse.coo_matrix method), 248
minimum() (cupyx.scipy.sparse.csc_matrix method), 256
minimum() (cupyx.scipy.sparse.csr_matrix method), 265
minimum() (cupyx.scipy.sparse.dia_matrix method), 271
minimum() (cupyx.scipy.sparse.spmatrix method), 276
minimum_filter () (in module cupyx.scipy.ndimage), 228
minimum_filter1d () (in module cupyx.scipy.ndimage), 229
mod (in module cupy), 36
modf (in module cupy), 48
module (cupy.RawModule attribute), 346
moveaxis() (in module cupy), 62
msort() (in module cupy), 183
multinomial() (in module cupy.random), 164
multiply (in module cupy), 35
multiply() (cupyx.scipy.sparse.coo_matrix method), 248
multiply() (cupyx.scipy.sparse.csc_matrix method), 256
multiply() (cupyx.scipy.sparse.csr_matrix method), 265
multiply() (cupyx.scipy.sparse.dia_matrix method), 271
multiply() (cupyx.scipy.sparse.spmatrix method), 276
multivariate_normal() (cupy.random.RandomState method), 177
multivariate_normal() (in module cupy.random), 165
N
n_free_blocks() (cupy.cuda.MemorPool method), 309
n_free_blocks() (cupy.cuda.PinnedMemoryPool method), 310
name (cupy.cuda.memory_hooks.DebugPrintHook attribute), 315
name (cupy.cuda.memory_hooks.LineProfileHook attribute), 317
name (cupy.cuda.MemoryHook attribute), 313
name (cupy.ElementwiseKernel attribute), 340
name (cupy.RawKernel attribute), 343
name (cupy.ReductionKernel attribute), 341
name (cupy.ufunc attribute), 33
name_expressions (cupy.RawModule attribute), 346
nan_to_num (in module cupy), 141
nanargmax() (in module cupy), 186
nanargmin() (in module cupy), 187
nanmax() (in module cupy), 191
nanmean() (in module cupy), 194
nanmin() (in module cupy), 190
nanprod() (in module cupy), 137
nansum() (in module cupy), 137
nanvar() (in module cupy), 194
nargs (cupy.ElementwiseKernel attribute), 340
nargs (cupy.ReductionKernel attribute), 341
nargs (cupy.ufunc attribute), 33
nbytes (cupy.ndarray attribute), 29
NcclCommunicator (class in cupy.cuda.nccl), 328
nd (cupy.broadcast attribute), 65
ndarray (class in cupy), 19
ndarray (class in cupy), 19
ndim (cupy.cuda.texture.CUDAArray attribute), 323
ndim (cupy.ndarray attribute), 29
ndim (cupyx.scipy.sparse.coo_matrix attribute), 251
ndim (cupyx.scipy.sparse.csc_matrix attribute), 260
ndim (cupyx.scipy.sparse.csr_matrix attribute), 268
ndim (cupyx.scipy.sparse.dia_matrix attribute), 274
ndim (cupyx.scipy.sparse.spmatrix attribute), 277
ndtr (in module cupyx.scipy.special), 289
negative (in module cupy), 36
negative_binomial() (cupy.random.RandomState method), 177
negative_binomial() (in module cupy.random), 166
next_fast_len() (in module cupyx.scipy.fft), 209
nextafter() (in module cupy), 48
nin (cupy.ElementwiseKernel attribute), 340
nin (cupy.ReductionKernel attribute), 341
nin (cupy.ufunc attribute), 33
nnz (cupyx.scipy.sparse.coo_matrix attribute), 251
nnz (cupyx.scipy.sparse.csc_matrix attribute), 260
nnz (cupyx.scipy.sparse.csr_matrix attribute), 268
nnz (cupyx.scipy.sparse.dia_matrix attribute), 274
nnz (cupyx.scipy.sparse.spmatrix attribute), 277
no_return (cupy.ElementwiseKernel attribute), 340
noncentral_chisquare() (cupy.random.RandomState method), 178
noncentral_chisquare() (in module cupy.random), 166
noncentral_f() (cupy.random.RandomState method), 178
noncentral_f() (in module cupy.random), 167
nonzero() (cupy.ndarray method), 25
nonzero() (in module cupy), 89
norm() (in module cupy.linalg), 111
norm() (in module cupyx.scipy.sparse.linalg), 285
normal() (cupy.random.RandomState method), 178
normal() (in module cupy.random), 167
not_equal (in module cupy), 45
nout (cupy.ElementwiseKernel attribute), 340
nout (cupy.ReductionKernel attribute), 341
nout (cupy.ufunc attribute), 33
null (cupy.cuda.ExternalStream attribute), 319
null (cupy.cuda.Stream attribute), 318
num_regs (cupy.RawKernel attribute), 343
numpy_cupy_allclose() (in module cupy.testing), 354
numpy_cupy_array_almost_equal() (in module cupy.testing), 355
numpy_cupy_array_almost_equal_nulp() (in module cupy.testing), 355
numpy_cupy_array_equal() (in module cupy.testing), 356
numpy_cupy_array_less() (in module cupy.testing), 357
numpy_cupy_array_list_equal() (in module cupy.testing), 357
numpy_cupy_array_max_ulp() (in module cupy.testing), 356
numpy_cupy_array_equal() (in module cupy.testing), 356
numpy_cupy_array_almost_equal() (in module cupy.testing), 356
numpy_cupy_array_almost_equal_nulp() (in module cupy.testing), 356
numpy_cupy_allclose() (in module cupy.testing), 354
numpy_cupy_array_almost_equal() (in module cupy.testing), 355
numpy_cupy_array_almost_equal_nulp() (in module cupy.testing), 355
numpy_cupy_array_equal() (in module cupy.testing), 356
numpy_cupy_array_less() (in module cupy.testing), 357
numpy_cupy_array_list_equal() (in module cupy.testing), 357
numpy_cupy_array_max_ulp() (in module cupy.testing), 356

O
o (cupy.poly1d attribute), 149
ogrid (in module cupy), 58
ones() (in module cupy), 51
ones_like() (in module cupy), 52
operation (cupy.ElementwiseKernel attribute), 340
optimize() (in module cupyx.optimizing), 347
options (cupy.RawKernel attribute), 343
options (cupy.RawModule attribute), 346
options (cupy.ReductionKernel attribute), 341
order (cupy.poly1d attribute), 149
order_filter() (in module cupyx.scipy.signal), 295
out_params (cupy.ElementwiseKernel attribute), 340
out_params (cupy.ReductionKernel attribute), 341
outer() (in module cupy), 105
packbits() (in module cupy), 76
pad() (in module cupy), 144
params (cupy.ElementwiseKernel attribute), 340
params (cupy.ReductionKernel attribute), 341
pareto() (cupy.random.RandomState method), 178
pareto() (in module cupy.random), 168
partition() (cupy.ndarray method), 25
partition() (in module cupy), 184
pci_bus_id (cupy.cudadev.Device attribute), 301
percentile() (in module cupy), 191
percentile_filter() (in module cupyx.scipy.ndimage), 230
permutation() (cupy.random.RandomState method), 178
permutation() (in module cupy.random), 157
piecewise() (in module cupy), 88
PinnedMemory (in cupy.cudadev), 304
PinnedMemoryPointer (class in cupy.cudadev), 306
PinnedMemoryPool (class in cupy.cudadev), 310
pinv() (in module cupy.linalg), 115
place() (in module cupy), 97
pointerGetAttributes() (in module cupy.cuda.runtime), 336
poisson() (cupy.random.RandomState method), 178
poisson() (in module cupy), 168
poly1d (class in cupy), 148
polyadd() (in module cupy), 151
polycompanion() (in module cupy.polynomial.polynomial), 147
polygamma() (in module cupyx.scipy.special), 288
polymul() (in module cupy), 151
polysub() (in module cupy), 151
polyval() (in module cupy), 150
polyvander() (in module cupy.polynomial.polynomial), 147
post_map_expr (cupy.ReductionKernel attribute), 341
power (in module cupy), 36
power() (cupy.random.RandomState method), 178
power() (cupyx.scipy.sparse.coo_matrix method), 248
power() (cupyx.scipy.sparse.csc_matrix method), 256
power() (cupyx.scipy.sparse.csr_matrix method), 265
power() (cupyx.scipy.sparse.dia_matrix method), 271
power() (cupyx.scipy.sparse.spmatrix method), 276
power() (in module cupy.random), 169
preamble (cupy.ElementwiseKernel attribute), 340
preamble (cupy.ReductionKernel attribute), 341
preferred_shared_memory_carveout (cupy.RawKernel attribute), 343
prewitt() (in module cupyx.scipy.ndimage), 230
print_report() (cupy.cudamemtx.HookProfileSample method), 316
prod() (cupy.ndarray method), 25
prod() (in module cupy), 135
profile() (in module cupy.cudarray), 326
pseudo_huber (in module cupyx.scipy.special), 287
ptp() (cupy.ndarray method), 25
ptr (cupy.cudamemtx.Memory attribute), 303
ptr (cupy.cudamemtx.MemoryPointer attribute), 306
ptr (cupy.cudamemtx.PinnedMemoryPointer attribute), 306
ptr (cupy.cudamemtx.Texture.ChannelFormatDescriptor attribute), 321
ptr (cupy.cudamemtx.CUDAArrary attribute), 323
ptr (cupy.cudamemtx.Texture.ResourceDescriptor attribute), 324
ptr (cupy.cudamemtx.SurfaceObject attribute), 325
ptr (cupy.cudamemtx.TextureDescriptor attribute), 324
ptr (cupy.cudamemtx.UnownedMemory attribute), 303
ptrx_version (cupy.RawKernel attribute), 343
put() (cupy.ndarray method), 25
put() (in module cupy), 98
putmask() (in module cupy), 98
PythonFunctionAllocator (class in cupy.cudarray), 310
radians (in module cupy), 133
radians() (cupy.random.RandomState method), 179
rand() (cupy.random.RandomState method), 152
rand() (in module cupy.random), 153
rand() (in module cupy.random), 153
rand() (in module cupy.random), 152
rand() (cupy.random.RandomState method), 179
rand() (in module cupy.random), 153
rand() (in module cupy.random), 153
random() (in module cupy.random), 155
random() (in module cupy.random), 282
random_integers() (in module cupy.random), 154
random_sample() (cupy.random.RandomState method), 179
random_sample() (in module cupy.random), 154
ranf() (in module cupy.random), 155
r_ (in module cupy.linalg), 109
qr() (in module cupy.linalg), 109
rad2deg (in module cupy), 42
rad2deg() (cupyx.scipy.sparse.coo_matrix method), 248
rad2deg() (cupyx.scipy.sparse.csc_matrix method), 256
rad2deg() (cupyx.scipy.sparse.csr_matrix method), 265
rad2deg() (cupyx.scipy.sparse.dia_matrix method), 271
rad2deg() (cupyx.scipy.sparse.spmatrix method), 276
rad2deg() (in module cupy.random), 169
ravel() (cupy.ElementwiseKernel attribute), 340
ravel() (cupy.ReductionKernel attribute), 341
ravel() (cupy.RawKernel attribute), 343
ravel() (cupy.cudakernel.LineProfileHook method), 316
ravel() (in module cupy.linalg), 109
ravel() (in module cupy.random), 155
}

Index
RangePush() (in module cupy.cuda.mtx), 328
RangePushC() (in module cupy.cuda.mtx), 328
rank_filter() (in module cupy.scipy.ndimage), 231
rank_id() (cupy.cuda.nccl.NcclCommunicator method), 329
ravel() (cupy.ndarray method), 25
ravel() (in module cupy), 62
ravel_multi_index() (in module cupy), 91
RayKernel (class in cupy), 342
RawModule (class in cupy), 344
rayleigh() (cupy.random.RandomState method), 179
rayleigh() (cupy.random.RandomState method), 179
real (cupy.ndarray attribute), 29
real() (in module cupy), 140
reciprocal (in module cupy), 38
record (cupy.cuda.Event attribute), 320
record (cupy.cuda.ExternalStream attribute), 319
record (cupy.cuda.Stream attribute), 319
recv() (cupy.cuda.ncll.NcclCommunicator method), 329
reduce() (cupy.cuda.ncll.NcclCommunicator method), 329
reduce_dims (cupy.ElementwiseKernel attribute), 340
reduce_dims (cupy.ReductionKernel attribute), 341
reduce_expr (cupy.ReductionKernel attribute), 341
reduce_type (cupy.ReductionKernel attribute), 341
reduced_view() (cupy.ndarray method), 25
reduceScatter() (cupy.cuda.ncll.NcclCommunicator method), 329
ReductionKernel (class in cupy), 340
rel_entr (in module cupyx.scipy.special), 287
remainder (in module cupy), 36
repeat() (cupy.ndarray method), 26
repeat() (in module cupy), 72
require() (in module cupy), 67
ResDesc (cupy.cuda.texture.SurfaceObject attribute), 325
ResDesc (cupy.cuda.texture.TextureObject attribute), 325
ResDesc (cupy.cuda.texture.TextureReference attribute), 326
reshape() (cupy.ndarray method), 26
reshape() (cupy.scipy.sparse.coo_matrix method), 248
reshape() (cupy.scipy.sparse.csc_matrix method), 256
reshape() (cupy.scipy.sparse.csr_matrix method), 265
reshape() (cupy.scipy.sparse.csr_matrix method), 265
reshape() (cupy.scipy.sparse.csc_matrix method), 256
reshape() (cupy.scipy.sparse.dia_matrix method), 271
reshape() (cupy.scipy.sparse.spmatrix method), 276
reshape() (in module cupy), 61
ResourceDescriptor (class in cupy.cuda.texture), 323
result_type() (in module cupy), 78
return_tuple (cupy.ElementwiseKernel attribute), 340
rfft() (in module cupy.fft), 81
rfft() (in module cupy.fft), 81
rfft() (in module cupy.fft), 204
rfft() (in module cupy.fft), 214
rfft2() (in module cupy.fft), 82
rfft2() (in module cupy.fft), 205
rfftfreq() (in module cupy.fft), 85
rfftn() (in module cupy.fft), 83
rfftn() (in module cupy.fft), 206
right_shift (in module cupy), 43
rint (in module cupy), 37
rint() (cupy.scipy.sparse.coo_matrix method), 248
rint() (cupy.scipy.sparse.csr_matrix method), 256
rint() (cupy.scipy.sparse.csc_matrix method), 265
rint() (cupy.scipy.sparse.dia_matrix method), 272
roll() (in module cupy), 75
rollaxis() (in module cupy), 63
roots (cupy.poly1d attribute), 150
roots() (in module cupy), 150
rot90() (in module cupy), 75
rotate() (in module cupyx.scipy.ndimage), 237
round() (cupy.ndarray method), 26
round_() (in module cupy), 134
rsqrt() (in module cupy), 198
runtimeGetVersion() (in module cupy.cuda.runtime), 332
S
sample() (in module cupy.random), 155
save() (in module cupy), 101
savez() (in module cupy), 101
savez_compressed() (in module cupy), 102
scatter_add() (cupy.ndarray method), 26
scatter_add() (in module cupy), 198
scatter_max() (cupy.ndarray method), 26
scatter_max() (in module cupy), 199
scatter_min() (cupy.ndarray method), 26
scatter_min() (in module cupy), 199
searchsorted() (in module cupy), 188
seed() (cupy.random.RandomState method), 179
seed() (in module cupy.random), 181
select() (in module cupy), 96
send() (cupy.cuda.ncll.NcclCommunicator method), 330
set() (cupy.ndarray method), 26
set() (cupy.poly1d method), 149
set_allocator() (in module cupy.cua), 307
set_cufft_gpus() (in module cupy.fft.config), 86
set_limit() (cupy.cuda.MemoryPool method), 309
set_pinned_memory_allocator() (in module cupy.cuda), 308
set_random_state() (in module cupy.random), 182
set_shape() (cupyx.scipy.sparse.coo_matrix method), 248
set_shape() (cupyx.scipy.sparse.csc_matrix method), 256
set_shape() (cupyx.scipy.sparse.csr_matrix method), 265
set_shape() (cupyx.scipy.sparse.dia_matrix method), 272
set_shape() (cupyx.scipy.sparse.spmatrix attribute), 276
setDevice() (in module cupy.cuda.runtime), 333
shape (cupy.broadcast attribute), 65
shape (cupy.ndarray attribute), 303
shape (cupy.scipy.sparse.coo_matrix attribute), 248
shape (cupy.scipy.sparse.csc_matrix attribute), 251
shape (cupy.scipy.sparse.csr_matrix attribute), 260
shape (cupy.scipy.sparse.csr_matrix method), 274
shape (cupy.scipy.sparse.csr_matrix method), 277
shape (cupy.scipy.sparse.csc_matrix method), 269
shape (cupy.scipy.sparse.csc_matrix method), 260
shape (cupy.scipy.sparse.dia_matrix attribute), 274
shape (cupy.scipy.sparse.dia_matrix method), 277
shape (cupy.scipy.sparse.dia_matrix method), 274
shape (cupy.scipy.sparse.spmatrix attribute), 277
shape (cupy.cuda.PinnedMemoryPointer method), 306
shape (cupyx.scipy.sparse.dia_matrix method), 265
shape (cupyx.scipy.sparse.dia_matrix method), 257
shape (cupyx.scipy.sparse.dia_matrix method), 272
size (cupyx.scipy.sparse.csr_matrix attribute), 268
size (cupyx.scipy.sparse.csr_matrix attribute), 269
size (cupyx.scipy.sparse.csr_matrix attribute), 277
size (cupy.cuda.nvcl.NcclCommunicator method), 330
size (cupyx.scipy.sparse.csr_matrix method), 265
slogdet() (in module cupy.linalg), 112
solve() (in module cupy.linalg), 113
solve_triangular() (in module cupyx.scipy.linalg), 118
sort() (cupy.ndarray method), 26
sort() (in module cupy), 182
sort_complex() (in module cupy), 184
sort_indices() (cupyx.scipy.sparse.csc_matrix method), 257
sort_indices() (cupyx.scipy.sparse.csr_matrix method), 265
sorted_indices() (cupyx.scipy.sparse.csc_matrix method), 257
sorted_indices() (cupyx.scipy.sparse.csr_matrix method), 265
spdiags() (in module cupyx.scipy.sparse), 281
split() (in module cupy), 70
cupyx.scipy.sparse.coo_matrix attribute
spmatrix (class in cupyx.scipy.sparse), 274
sqrt (in module cupy), 38
sqrt() (cupyx.scipy.sparse.coo_matrix method), 248
sqrt() (cupyx.scipy.sparse.csc_matrix method), 257
sqrt() (cupyx.scipy.sparse.csr_matrix method), 265
sqrt() (cupyx.scipy.sparse.dia_matrix method), 272
cupyx.scipy.sparse.dia_matrix method
square (in module cupy), 38
sqrt() (cupyx.scipy.sparse.csr_matrix method), 265
cupyx.scipy.sparse.dia_matrix method
squeeze() (cupy.ndarray method), 27
sqrt() (cupyx.scipy.sparse.dia_matrix method), 272
stack() (in module cupy), 66
stack() (in module cupy), 68
standard_cauchy() (cupy.random.RandomState method), 179
cupyx.scipy.sparse.csr_matrix method
standard_cauchy() (in module cupy.random), 170
standard_deviation() (in module cupyx.scipy.sparse.ndimage), 240
standard_exp() (cupy.random.RandomState method), 180
cupyx.scipy.sparse.csr_matrix method
standard_exp() (in module cupy.random), 170
standard_deviation() (in module cupy.random), 170
standard_exponential() (cupy.random.RandomState method), 180
standard_exponential() (in module cupy.random), 170
standard_exponential() (in module cupy.random), 170
standard_gamma() (cupy.random.RandomState method), 180
standard_cauchy() (in module cupy.random), 170
standard_cauchy() (in module cupy.random), 170
standard_normal() (cupy.random.RandomState method), 180
standard_normal() (in module cupy.random), 170
standard_normal() (in module cupy.random), 170
standard_t() (cupy.random.RandomState method), 180
standard_t() (in module cupy.random), 171
start() (in module cupy.cuda.profiler), 327
std() (cupy.ndarray method), 27
CuPy Documentation, Release 8.2.0

Index

435
std() (in module cupy), 193
stop() (in module cupy.cudarray), 327
Stream (class in cupy.cuda), 317
streamAddCallback() (in module cupy.cuda.runtime), 336
streamCreate() (in module cupy.cuda.runtime), 336
streamCreateWithFlags() (in module cupy.cuda.runtime), 336
streamDestroy() (in module cupy.cuda.runtime), 336
streamQuery() (in module cupy.cuda.runtime), 337
streamSynchronize() (in module cupy.cuda.runtime), 336
streamWaitEvent() (in module cupy.cuda.runtime), 337
strides (cupy.ndarray attribute), 30
subtract (in module cupy), 34
sum() (cupy.ndarray method), 27
sum() (cupyx.scipy.sparse.coo_matrix method), 248
sum() (cupyx.scipy.sparse.csc_matrix method), 257
sum() (cupyx.scipy.sparse.csr_matrix method), 266
sum() (cupyx.scipy.sparse.dia_matrix method), 272
sum() (cupyx.scipy.sparse.spmatrix method), 276
sum() (in module cupy), 135
sum() (in module cupy.cudarray), 240
sum_duplicates() (cupyx.scipy.sparse.coo_matrix method), 249
sum_duplicates() (cupyx.scipy.sparse.csc_matrix method), 257
sum_duplicates() (cupyx.scipy.sparse.csr_matrix method), 266

SurfaceObject (class in cupy.cuda.textures), 325
svd() (in module cupy.linalg), 109
swapaxes() (cupy.ndarray method), 27
swapaxes() (in module cupy), 63
synchronize (cupy.cuda.Event attribute), 320
synchronize (cupy.cuda.ExternalStream attribute), 319
synchronize (cupy.cuda.Stream attribute), 318
synchronize() (cupy.cuda.Device method), 300

t (cupy.ndarray attribute), 29
t (cupyx.scipy.sparse.coo_matrix attribute), 251
t (cupyx.scipy.sparse.csc_matrix attribute), 259
t (cupyx.scipy.sparse.csr_matrix attribute), 268
t (cupyx.scipy.sparse.dia_matrix attribute), 273
t (cupyx.scipy.sparse.spmatrix attribute), 277
take() (cupy.ndarray method), 27
take() (in module cupy), 93
take_along_axis() (in module cupy), 93
tan (in module cupy), 40
tan() (cupyx.scipy.sparse.coo_matrix method), 249
tan() (cupyx.scipy.sparse.csr_matrix method), 257
tan() (cupyx.scipy.sparse.csc_matrix method), 266
tanh() (cupyx.scipy.sparse.coo_matrix method), 249
tanh() (cupyx.scipy.sparse.csr_matrix method), 257
tanh() (cupyx.scipy.sparse.csc_matrix method), 266
tanh() (cupyx.scipy.sparse.dia_matrix method), 272
tensor() (in module cupy), 106
tensorinv() (in module cupy.linalg), 116
tensorsolve() (in module cupy.linalg), 114
texDesc (cupy.cuda.textures.TextureObject attribute), 325
texDesc (cupy.cuda.textures.TextureReference attribute), 326
texref (cupy.cuda.textures.TextureReference attribute), 326
TextureDescriptor (class in cupy.cuda.texture), 324
TextureObject (class in cupy.cuda.texture), 325
TextureReference (class in cupy.cuda.texture), 325
tile() (in module cupy), 71
time_range() (in module cupy.prof), 364
timeRangeDecorator (class in cupy.prof), 363
toarray() (cupyx.scipy.sparse.coo_matrix method), 249
toarray() (cupyx.scipy.sparse.csc_matrix method), 258
toarray() (cupyx.scipy.sparse.csr_matrix method), 266
toarray() (cupyx.scipy.sparse.dia_matrix method), 272
toarray() (cupyx.scipy.sparse.spmatrix method), 276
tobsr() (cupyx.scipy.sparse.coo_matrix method), 249
tobsr() (cupyx.scipy.sparse.csc_matrix method), 258
tobsr() (cupyx.scipy.sparse.csr_matrix method), 266
tobsr() (cupyx.scipy.sparse.dia_matrix method), 272
tobytes() (cupy.ndarray method), 28
tocoo() (cupyx.scipy.sparse.coo_matrix method), 250
tocoo() (cupyx.scipy.sparse.csr_matrix method), 258
tocoo() (cupyx.scipy.sparse.csc_matrix method), 266
tocoo() (cupyx.scipy.sparse.dia_matrix method), 272
tocoo() (cupyx.scipy.sparse.spmatrix method), 276
tocsr() (cupyx.scipy.sparse.coo_matrix method), 250
tocsr() (cupyx.scipy.sparse.csc_matrix method), 258
tocsr() (cupyx.scipy.sparse.csr_matrix method), 266
tocsr() (cupyx.scipy.sparse.dia_matrix method), 272
tocsr() (cupyx.scipy.sparse.spmatrix method), 276
todense() (cupy.scipy.sparse.coo_matrix method), 250
todense() (cupy.scipy.sparse.csc_matrix method), 258
todense() (cupy.scipy.sparse.csr_matrix method), 267
todense() (cupy.scipy.sparse.dia_matrix method), 273
todense() (cupy.scipy.sparse.spmatrix method), 276
todense() (cupy.scipy.sparse.spmatrix method), 276
todense() (cupy.scipy.sparse.spmatrix method), 276
todense() (cupy.scipy.sparse.spmatrix method), 276
todense() (cupy.scipy.sparse.spmatrix method), 276
todense() (cupy.scipy.sparse.spmatrix method), 276
todense() (cupy.scipy.sparse.spmatrix method), 276
todense() (cupy.scipy.sparse.spmatrix method), 276
todense() (cupy.scipy.sparse.spmatrix method), 276
todense() (cupy.scipy.sparse.spmatrix method), 276
todense() (cupy.scipy.sparse.spmatrix method), 276
todense() (cupy.scipy.sparse.spmatrix method), 276
todense() (cupy.scipy.sparse.spmatrix method), 276
tri() (in module cupy), 63
trim_coef() (in module cupy.polynomial.polynomial), 148
trimseq() (in module cupy.polynomial.polynomial), 148
triu() (in module cupy), 60
triu() (in module cupy.scipy.linalg), 120
true_divide (in module cupy), 35
trunc (in module cupy), 49
trunc() (cupy.scipy.sparse.coo_matrix method), 250
trunc() (cupy.scipy.sparse.csc_matrix method), 259
trunc() (cupy.scipy.sparse.csr_matrix method), 267
trunc() (cupy.scipy.sparse.dia_matrix method), 273
types (cupy.ufunc attribute), 33

U

ufunc (class in cupy), 33
uniform() (cupy.random.RandomState method), 180
uniform() (in module cupy.random), 172
uniform_filter() (in module cupyx.scipy.ndimage), 232
uniform_filter1d() (in module cupyx.scipy.ndimage), 233
unique() (in module cupy), 72
UnownedMemory (class in cupy.cuda), 303
unpackbits() (in module cupy), 77
unravel_index() (in module cupy), 92
unwrap() (in module cupy), 133
use (cupy.cuda.ExternalStream attribute), 319
use (cupy.cuda.Stream attribute), 318
use() (cupy.cuda.Device method), 300
used_bytes() (cupy.cuda.MemoryPool method), 309
using_allocator() (in module cupy.cuda), 307

V

values (cupy.broadcast attribute), 65
var() (cupy.ndarray method), 28
var() (in module cupy), 193
variable (cupy.poly1d attribute), 150
variance() (in module cupyx.scipy.ndimage), 241
vdot() (in module cupy), 105
view() (cupy.ndarray method), 28
vonmises() (cupy.random.RandomState method), 180
vonmises() (in module cupy.random), 173
vsplit() (in module cupy), 71
vstack() (in module cupy), 70
vstack() (in module cupy.scipy.sparse), 282

W

wait_event (cupy.cuda.ExternalStream attribute), 319
wait_event (cupy.cuda.Stream attribute), 318
wald() (cupy.random.RandomState method), 181
wald() (in module cupy.random), 173
weibull() (cupy.random.RandomState method), 181
weibull() (in module cupy.random), 174
where() (in module cupy), 90
who() (in module cupy), \texttt{394}
width (cupy.cudenarray.CUDArray attribute), \texttt{323}
wiener() (in module cupy.scipy.signal), \texttt{297}

\texttt{Y}

y0 (in module cupy.scipy.special), \texttt{286}
y1 (in module cupy.scipy.special), \texttt{286}

\texttt{Z}

zeros() (in module cupy), \texttt{52}
zeros_like() (in module cupy), \texttt{53}
zeta (in module cupy.scipy.special), \texttt{290}
zipf() (cupy.random.RandomState method), \texttt{181}
zipf() (in module cupy.random), \texttt{174}
zoom() (in module cupy.scipy.ndimage), \texttt{238}