# CuPy Documentation

***Release 7.8.0***

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

**Aug 19, 2020**

# Contents

This is the CuPy documentation.

# CHAPTER 1

# Overview

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

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

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

CHAPTER 2

# Tutorial

## 2.1 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

### 2.1.1 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:

```
>>> 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.

```
>>> 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.

```
>>> 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:

```
>>> 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.

### 2.1.2 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.

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

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

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

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

```
>>> 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:

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

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

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

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

### 2.1.3 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:

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

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

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

---

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

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

---

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

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

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

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

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

---

**Note:** If you work with Chainer, you can also use to_cpu() and to_gpu() to move arrays back and forth between a device and a host, or between different devices. Note that to_gpu() has device option to specify the device which arrays are transferred.

---

### 2.1.4 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:

```
>>> # 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.

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

## 2.2 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.

### 2.2.1 Basics of elementwise kernels

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

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

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

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

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

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

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

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

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

### 2.2.2 Type-generic kernels

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

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

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

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

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

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

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

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

### 2.2.3 Raw argument specifiers

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

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

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

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

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

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()`.

### 2.2.4 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:

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

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

### 2.2.5 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.

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

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

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

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

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

```
>>> add_kernel = cp.RawKernel(r'''
... extern "C" __global__
... void my_add(const float* x1, const float* x2, float* y) {
...     int tid = blockDim.x * blockIdx.x + threadIdx.x;
...     y[tid] = x1[tid] + x2[tid];
```

(continues on next page)

(continued from previous page)

```
... }
... ''', '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` arugment. The static CUDA device runtime library (`cudadevrt`) is automatically discovered by CuPy. For further detail, see CUDA Toolkit's documentation.

Accessing texture memory in *RawKernel* is supported via CUDA Runtime's Texture Object API, see *TextureObject*'s documentation 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` and `bool` map to `long long`, `double` 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.

---

### 2.2.6 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.

```
>>> 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)
```

(continues on next page)

(continued from previous page)

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

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

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.)

### 2.2.7 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:

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

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

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

(continues on next page)

(continued from previous page)

```
>>> squared_diff(x_np, y_np)
array([81, 49, 25,  9,  1,  1,  9, 25, 49, 81])
```

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

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

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

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

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

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

CHAPTER 3

# Reference Manual

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

- genindex
- modindex

## 3.1 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.

| | |
|---|---|
| *cupy.ndarray* | Multi-dimensional array on a CUDA device. |

### 3.1.1 cupy.ndarray

**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** (*{'C', 'F'}*) – Row-major (C-style) or column-major (Fortran-style) order.

**Variables**

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

  **See also:**

  Data type objects (dtype)
- ***size*** (*int*) – Number of elements this array holds.

  This is equivalent to product over the shape tuple.

  **See also:**

  numpy.ndarray.size

### Methods

**__getitem__**()

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

Supports both basic and advanced indexing.

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

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

#### Example

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

**__setitem__**()

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

Supports both basic and advanced indexing.

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

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

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

---

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

```
>>> import cupy
>>> a = cupy.zeros((2,))
>>> 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__**()
Return len(self).

**__iter__**()
Implement iter(self).

**__copy__**(*self*)

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

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

**argmax**(*self*, *axis=None*, *out=None*, *dtype=None*, *keepdims=False*) → ndarray
Returns the indices of the maximum along a given axis.

---

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

---

---

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

---

**See also:**

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

**argmin**(*self*, *axis=None*, *out=None*, *dtype=None*, *keepdims=False*) → ndarray
Returns the indices of the minimum along a given axis.

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

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

**See also:**

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

**argpartition**(*self*, *kth*, *axis=-1*) → ndarray
Returns the indices that would partially sort an array.

**Parameters**

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

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

**Return type** *cupy.ndarray*

**See also:**

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

**argsort**(*self*, *axis=-1*) → ndarray
Returns the indices that would sort an array with stable sorting

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

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

**Return type** *cupy.ndarray*

**See also:**

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

**astype**(*self*, *dtype*, *order=u'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=u'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()

**conj**(*self*) → ndarray

**copy**(*self*, *order=u'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*, *dtype=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*, *dtype=None*, *keepdims=False*) → ndarray
Returns the minimum along a given axis.

**See also:**

*cupy.amin()* for full documentation, numpy.ndarray.min()

**nonzero**(*self*) → tuple
Return the indices of the elements that are non-zero.

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

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

**Return type** tuple of arrays

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

**See also:**

numpy.nonzero()

**partition**(*self*, *kth*, *int axis=-1*)
Partitions an array.

**Parameters**

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

**See also:**

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

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

**See also:**

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

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

**See also:**

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

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

**See also:**

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

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

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

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

**Return type** *cupy.ndarray*

**repeat**(*self*, *repeats*, *axis=None*)
Returns an array with repeated arrays along an axis.

**See also:**

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

**reshape**(*self*, **shape*, *order='C'*)
Returns an array of a different shape and the same content.

**See also:**

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

**round**(*self*, *decimals=0*, *out=None*) → ndarray
Returns an array with values rounded to the given number of decimals.

**See also:**

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

**scatter_add**(*self*, *slices*, *value*)
Adds given values to specified elements of an array.

**See also:**

*cupyx.scatter_add()* for full documentation.

**scatter_max**(*self*, *slices*, *value*)
Stores a maximum value of elements specified by indices to an array.

**See also:**

*cupyx.scatter_max()* for full documentation.

**scatter_min**(*self*, *slices*, *value*)
Stores a minimum value of elements specified by indices to an array.

**See also:**

*cupyx.scatter_min()* for full documentation.

**set**(*self*, *arr*, *stream=None*)
Copies an array on the host memory to *cupy.ndarray*.

**Parameters**

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

**sort**(*self*, *int axis=-1*)
Sort an array, in-place with a stable sorting algorithm.

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

---

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

---

**See also:**

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

**squeeze** (*self*, *axis=None*) → ndarray
Returns a view with size-one axes removed.

**See also:**

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

**std** (*self*, *axis=None*, *dtype=None*, *out=None*, *ddof=0*, *keepdims=False*) → ndarray
Returns the standard deviation along a given axis.

**See also:**

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

**sum** (*self*, *axis=None*, *dtype=None*, *out=None*, *keepdims=False*) → ndarray
Returns the sum along a given axis.

**See also:**

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

**swapaxes** (*self*, *Py_ssize_t axis1*, *Py_ssize_t axis2*) → ndarray
Returns a view of the array with two axes swapped.

**See also:**

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

**take** (*self*, *indices*, *axis=None*, *out=None*) → ndarray
Returns an array of elements at given indices along the axis.

**See also:**

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

**toDlpack** (*self*)
Zero-copy conversion to a DLPack tensor.

DLPack is a open in memory tensor structure proposed in this repository: dmlc/dlpack.

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

**Returns**

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

**Return type** 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**

```
>>> 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:**

`numpy.ndarray.tofile()`

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

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

**Return type** list

**See also:**

`numpy.ndarray.tolist()`

**trace** (*self*, *offset=0*, *axis1=0*, *axis2=1*, *dtype=None*, *out=None*) → ndarray
Returns the sum along diagonals of the array.

**See also:**

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

**transpose** (*self*, **axes*)
Returns a view of the array with axes permuted.

**See also:**

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

**var** (*self*, *axis=None*, *dtype=None*, *out=None*, *ddof=0*, *keepdims=False*) → ndarray
Returns the variance along a given axis.

**See also:**

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

**view** (*self*, *dtype=None*) → ndarray
Returns a view of the array.

**Parameters** **dtype** – If this is different from the data type of the array, the returned view 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

**T**
Shape-reversed view of the array.

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

**base**

**cstruct**
C representation of the array.

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

**data**

**device**
CUDA device on which this array resides.

**dtype**

**flags**
Object containing memory-layout information.

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

**See also:**

`numpy.ndarray.flags`

**imag**

**itemsize**
Size of each element in bytes.

**See also:**

`numpy.ndarray.itemsize`

**nbytes**
Total size of all elements in bytes.

It does not count skips between elements.

**See also:**

`numpy.ndarray.nbytes`

**ndim**
Number of dimensions.

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

**See also:**

`numpy.ndarray.ndim`

**real**

**shape**
Lengths of axes.

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

**size**

**strides**
Strides of axes in bytes.

**See also:**

`numpy.ndarray.strides`

### 3.1.2 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.

| | |
|---|---|
| *cupy.get_array_module* | Returns the array module for arguments. |
| *cupyx.scipy.get_array_module* | Returns the array module for arguments. |

#### cupy.get_array_module

cupy.**get_array_module**(**args*)
Returns the array module for arguments.

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

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

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

**Return type** module

**Example**

A NumPy/CuPy generic function can be written as follows

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

#### cupyx.scipy.get_array_module

cupyx.scipy.**get_array_module**(**args*)
Returns the array module for arguments.

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

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

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

**Return type** module

### 3.1.3 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*s 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.

| | |
|---|---|
| *cupy.array* | Creates an array on the current device. |
| *cupy.asarray* | Converts an object to array. |
| *cupy.asnumpy* | Returns an array on the host memory from an arbitrary source array. |

### cupy.array

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

Creates an array on the current device.

This function currently does not support the subok option.

**Parameters**

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

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

**Return type** *cupy.ndarray*

---

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

---

**See also:**

numpy.array()

### cupy.asarray

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

Converts an object to array.

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

**Parameters**

- **a** – The source object.

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

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

**Return type** *cupy.ndarray*

**See also:**

`numpy.asarray()`

#### cupy.asnumpy

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

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

**Parameters**

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

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

**Return type** numpy.ndarray

## 3.2 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`.

### 3.2.1 Ufunc class

| | |
|---|---|
| *cupy.ufunc* | Universal function. |

#### cupy.ufunc

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

Universal function.

**Variables**

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

#### Methods

**__call__**()
Applies the universal function to arguments elementwise.

**Parameters**

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

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

#### 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 '->'.

### 3.2.2 Available ufuncs

#### Math operations

| | |
|---|---|
| *cupy.add* | Adds two arrays elementwise. |
| *cupy.subtract* | Subtracts arguments elementwise. |
| *cupy.multiply* | Multiplies two arrays elementwise. |
| *cupy.divide* | Elementwise true division (i.e. |
| *cupy.logaddexp* | Computes `log(exp(x1) + exp(x2))` elementwise. |
| *cupy.logaddexp2* | Computes `log2(exp2(x1) + exp2(x2))` elementwise. |
| *cupy.true_divide* | Elementwise true division (i.e. |
| *cupy.floor_divide* | Elementwise floor division (i.e. |
| *cupy.negative* | Takes numerical negative elementwise. |
| *cupy.power* | Computes `x1 ** x2` elementwise. |

Continued on next page

Table 5 – continued from previous page

| | |
|---|---|
| *cupy.remainder* | Computes the remainder of Python division elementwise. |
| *cupy.mod* | Computes the remainder of Python division elementwise. |
| *cupy.fmod* | Computes the remainder of C division elementwise. |
| *cupy.absolute* | Elementwise absolute value function. |
| *cupy.rint* | Rounds each element of an array to the nearest integer. |
| *cupy.sign* | Elementwise sign function. |
| *cupy.exp* | Elementwise exponential function. |
| *cupy.exp2* | Elementwise exponentiation with base 2. |
| *cupy.log* | Elementwise natural logarithm function. |
| *cupy.log2* | Elementwise binary logarithm function. |
| *cupy.log10* | Elementwise common logarithm function. |
| *cupy.expm1* | Computes `exp(x) - 1` elementwise. |
| *cupy.log1p* | Computes `log(1 + x)` elementwise. |
| *cupy.sqrt* | Elementwise square root function. |
| *cupy.square* | Elementwise square function. |
| *cupy.reciprocal* | Computes `1 / x` elementwise. |

### cupy.add

cupy.**add = <ufunc 'cupy_add'>**
Adds two arrays elementwise.

**See also:**

numpy.add

### cupy.subtract

cupy.**subtract = <ufunc 'cupy_subtract'>**
Subtracts arguments elementwise.

**See also:**

numpy.subtract

### cupy.multiply

cupy.**multiply = <ufunc 'cupy_multiply'>**
Multiplies two arrays elementwise.

**See also:**

numpy.multiply

### cupy.divide

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

**See also:**

numpy.true_divide

### cupy.logaddexp

cupy.**logaddexp = <ufunc 'cupy_logaddexp'>**
Computes `log(exp(x1) + exp(x2))` elementwise.

**See also:**

numpy.logaddexp

### cupy.logaddexp2

cupy.**logaddexp2 = <ufunc 'cupy_logaddexp2'>**
Computes `log2(exp2(x1) + exp2(x2))` elementwise.

**See also:**

numpy.logaddexp2

### cupy.true_divide

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

**See also:**

numpy.true_divide

### cupy.floor_divide

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

**See also:**

numpy.floor_divide

### cupy.negative

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

**See also:**

numpy.negative

### cupy.power

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

**See also:**

numpy.power

### cupy.remainder

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

**See also:**

numpy.remainder

### cupy.mod

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

**See also:**

numpy.remainder

### cupy.fmod

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

**See also:**

numpy.fmod

### cupy.absolute

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

**See also:**

numpy.absolute

### cupy.rint

cupy.**rint = <ufunc 'cupy_rint'>**
Rounds each element of an array to the nearest integer.

**See also:**

numpy.rint

### cupy.sign

cupy.**sign = <ufunc 'cupy_sign'>**
Elementwise sign function.

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

**See also:**

numpy.sign

### cupy.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`

### Trigonometric functions

| | |
|---|---|
| *cupy.sin* | Elementwise sine function. |
| *cupy.cos* | Elementwise cosine function. |
| *cupy.tan* | Elementwise tangent function. |
| *cupy.arcsin* | Elementwise inverse-sine function (a.k.a. |
| *cupy.arccos* | Elementwise inverse-cosine function (a.k.a. |

Continued on next page

Table 6 – continued from previous page

| | |
|---|---|
| *cupy.arctan* | Elementwise inverse-tangent function (a.k.a. |
| *cupy.arctan2* | Elementwise inverse-tangent of the ratio of two arrays. |
| *cupy.hypot* | Computes the hypoteneous of orthogonal vectors of given length. |
| *cupy.sinh* | Elementwise hyperbolic sine function. |
| *cupy.cosh* | Elementwise hyperbolic cosine function. |
| *cupy.tanh* | Elementwise hyperbolic tangent function. |
| *cupy.arcsinh* | Elementwise inverse of hyperbolic sine function. |
| *cupy.arccosh* | Elementwise inverse of hyperbolic cosine function. |
| *cupy.arctanh* | Elementwise inverse of hyperbolic tangent function. |
| *cupy.deg2rad* | Converts angles from degrees to radians elementwise. |
| *cupy.rad2deg* | Converts angles from radians to degrees elementwise. |

### cupy.sin

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

**See also:**

numpy.sin

### cupy.cos

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

**See also:**

numpy.cos

### cupy.tan

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

**See also:**

numpy.tan

### cupy.arcsin

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

**See also:**

numpy.arcsin

### cupy.arccos

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

**See also:**

numpy.arccos

### cupy.arctan

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

**See also:**

numpy.arctan

### cupy.arctan2

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

**See also:**

numpy.arctan2

### cupy.hypot

cupy.**hypot = <ufunc 'cupy_hypot'>**
Computes the hypoteneous of orthogonal vectors of given length.

This is equivalent to `sqrt(x1 **2 + x2 ** 2)`, while this function is more efficient.

**See also:**

numpy.hypot

### cupy.sinh

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

**See also:**

numpy.sinh

### cupy.cosh

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

**See also:**

numpy.cosh

### cupy.tanh

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

**See also:**

numpy.tanh

### cupy.arcsinh

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

**See also:**

numpy.arcsinh

### cupy.arccosh

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

**See also:**

numpy.arccosh

### cupy.arctanh

cupy.**arctanh = <ufunc 'cupy_arctanh'>**
Elementwise inverse of hyperbolic tangent function.

**See also:**

numpy.arctanh

### cupy.deg2rad

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

**See also:**

numpy.deg2rad, numpy.radians

### cupy.rad2deg

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

**See also:**

numpy.rad2deg, numpy.degrees

### Bit-twiddling functions

| | |
|---|---|
| *cupy.bitwise_and* | Computes the bitwise AND of two arrays elementwise. |
| *cupy.bitwise_or* | Computes the bitwise OR of two arrays elementwise. |
| *cupy.bitwise_xor* | Computes the bitwise XOR of two arrays elementwise. |
| *cupy.invert* | Computes the bitwise NOT of an array elementwise. |
| *cupy.left_shift* | Shifts the bits of each integer element to the left. |
| *cupy.right_shift* | Shifts the bits of each integer element to the right. |

#### cupy.bitwise_and

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

Only integer and boolean arrays are handled.

**See also:**

numpy.bitwise_and

#### cupy.bitwise_or

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

Only integer and boolean arrays are handled.

**See also:**

numpy.bitwise_or

#### cupy.bitwise_xor

cupy.**bitwise_xor = <ufunc 'cupy_bitwise_xor'>**
Computes the bitwise XOR of two arrays elementwise.

Only integer and boolean arrays are handled.

**See also:**

numpy.bitwise_xor

#### cupy.invert

cupy.**invert = <ufunc 'cupy_invert'>**
Computes the bitwise NOT of an array elementwise.

Only integer and boolean arrays are handled.

**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

| | |
|---|---|
| *cupy.greater* | Tests elementwise if `x1 > x2`. |
| *cupy.greater_equal* | Tests elementwise if `x1 >= x2`. |
| *cupy.less* | Tests elementwise if `x1 < x2`. |
| *cupy.less_equal* | Tests elementwise if `x1 <= x2`. |
| *cupy.not_equal* | Tests elementwise if `x1 != x2`. |
| *cupy.equal* | Tests elementwise if `x1 == x2`. |
| *cupy.logical_and* | Computes the logical AND of two arrays. |
| *cupy.logical_or* | Computes the logical OR of two arrays. |
| *cupy.logical_xor* | Computes the logical XOR of two arrays. |
| *cupy.logical_not* | Computes the logical NOT of an array. |
| *cupy.maximum* | Takes the maximum of two arrays elementwise. |
| *cupy.minimum* | Takes the minimum of two arrays elementwise. |
| *cupy.fmax* | Takes the maximum of two arrays elementwise. |
| *cupy.fmin* | Takes the minimum of two arrays elementwise. |

### cupy.greater

cupy.**greater = <ufunc 'cupy_greater'>**
Tests elementwise if `x1 > x2`.

**See also:**

numpy.greater

### cupy.greater_equal

cupy.**greater_equal = <ufunc 'cupy_greater_equal'>**
Tests elementwise if `x1 >= x2`.

**See also:**

numpy.greater_equal

### cupy.less

cupy.**less = <ufunc 'cupy_less'>**
Tests elementwise if `x1 < x2`.

**See also:**

numpy.less

### cupy.less_equal

cupy.**less_equal = <ufunc 'cupy_less_equal'>**
Tests elementwise if `x1 <= x2`.

**See also:**

numpy.less_equal

### cupy.not_equal

cupy.**not_equal = <ufunc 'cupy_not_equal'>**
Tests elementwise if `x1 != x2`.

**See also:**

numpy.equal

### cupy.equal

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

**See also:**

numpy.equal

### cupy.logical_and

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

**See also:**

numpy.logical_and

### cupy.logical_or

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

**See also:**

numpy.logical_or

### cupy.logical_xor

cupy.**logical_xor = <ufunc 'cupy_logical_xor'>**
Computes the logical XOR of two arrays.

**See also:**

numpy.logical_xor

### cupy.logical_not

cupy.**logical_not = <ufunc 'cupy_logical_not'>**
Computes the logical NOT of an array.

**See also:**

numpy.logical_not

### cupy.maximum

cupy.**maximum = <ufunc 'cupy_maximum'>**
Takes the maximum of two arrays elementwise.

If NaN appears, it returns the NaN.

**See also:**

numpy.maximum

### cupy.minimum

cupy.**minimum = <ufunc 'cupy_minimum'>**
Takes the minimum of two arrays elementwise.

If NaN appears, it returns the NaN.

**See also:**

numpy.minimum

### cupy.fmax

cupy.**fmax = <ufunc 'cupy_fmax'>**
Takes the maximum of two arrays elementwise.

If NaN appears, it returns the other operand.

**See also:**

numpy.fmax

### cupy.fmin

cupy.**fmin = <ufunc 'cupy_fmin'>**
Takes the minimum of two arrays elementwise.

If NaN appears, it returns the other operand.

**See also:**

numpy.fmin

### Floating functions

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

### cupy.isfinite

cupy.**isfinite = <ufunc 'cupy_isfinite'>**
Tests finiteness elementwise.

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

**See also:**

numpy.isfinite

### cupy.isinf

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

**See also:**

numpy.isinf

### cupy.isnan

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

**See also:**

numpy.isnan

### cupy.signbit

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

**See also:**

numpy.signbit

### cupy.copysign

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

**See also:**

numpy.copysign

### cupy.nextafter

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

---

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

---

**See also:**

numpy.nextafter

### cupy.modf

cupy.**modf = <ufunc 'cupy_modf'>**
Extracts the fractional and integral parts of an array elementwise.

This ufunc returns two arrays.

**See also:**

numpy.modf

#### cupy.ldexp

cupy.**ldexp = <ufunc 'cupy_ldexp'>**
Computes `x1 * 2 ** x2` elementwise.

**See also:**

`numpy.ldexp`

#### cupy.frexp

cupy.**frexp = <ufunc 'cupy_frexp'>**
Decomposes each element to mantissa and two's exponent.

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

**See also:**

`numpy.frexp`

#### cupy.floor

cupy.**floor = <ufunc 'cupy_floor'>**
Rounds each element of an array to its floor integer.

**See also:**

`numpy.floor`

#### cupy.ceil

cupy.**ceil = <ufunc 'cupy_ceil'>**
Rounds each element of an array to its ceiling integer.

**See also:**

`numpy.ceil`

#### cupy.trunc

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

**See also:**

`numpy.trunc`

### 3.2.3 ufunc.at

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

## 3.3 Routines

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

### 3.3.1 Array Creation Routines

#### Basic creation routines

| | |
|---|---|
| *cupy.empty* | Returns an array without initializing the elements. |
| *cupy.empty_like* | Returns a new array with same shape and dtype of a given array. |
| *cupy.eye* | Returns a 2-D array with ones on the diagonals and zeros elsewhere. |
| *cupy.identity* | Returns a 2-D identity array. |
| *cupy.ones* | Returns a new array of given shape and dtype, filled with ones. |
| *cupy.ones_like* | Returns an array of ones with same shape and dtype as a given array. |
| *cupy.zeros* | Returns a new array of given shape and dtype, filled with zeros. |
| *cupy.zeros_like* | Returns an array of zeros with same shape and dtype as a given array. |
| *cupy.full* | Returns a new array of given shape and dtype, filled with a given value. |
| *cupy.full_like* | Returns a full array with same shape and dtype as a given array. |

#### cupy.empty

cupy.**empty**(*shape*, *dtype=<class 'float'>*, *order='C'*)

Returns an array without initializing the elements.

**Parameters**

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

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

**Return type** *cupy.ndarray*

**See also:**

`numpy.empty()`

#### cupy.empty_like

cupy.**empty_like**(*a*, *dtype=None*, *order='K'*, *subok=None*, *shape=None*)

Returns a new array with same shape and dtype of a given array.

This function currently does not support `subok` option.

**Parameters**

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

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

**Return type** *cupy.ndarray*

**See also:**

numpy.empty_like()

### cupy.eye

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

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

**Parameters**

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

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

**Return type** *cupy.ndarray*

**See also:**

numpy.eye()

### cupy.identity

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

Returns a 2-D identity array.

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

**Parameters**

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

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

**Return type** *cupy.ndarray*

**See also:**

`numpy.identity()`

### cupy.ones

`cupy.`**`ones`**(*shape*, *dtype=<class 'float'>*, *order='C'*)
Returns a new array of given shape and dtype, filled with ones.

This function currently does not support `order` option.

**Parameters**

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

**Returns** An array filled with ones.

**Return type** *cupy.ndarray*

**See also:**

`numpy.ones()`

### cupy.ones_like

`cupy.`**`ones_like`**(*a*, *dtype=None*, *order='K'*, *subok=None*, *shape=None*)
Returns an array of ones with same shape and dtype as a given array.

This function currently does not support `subok` option.

**Parameters**

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

**Returns** An array filled with ones.

**Return type** *cupy.ndarray*

**See also:**

`numpy.ones_like()`

### cupy.zeros

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

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

**Parameters**

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

**Returns** An array filled with zeros.

**Return type** *cupy.ndarray*

**See also:**

numpy.zeros()

### cupy.zeros_like

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

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

This function currently does not support `subok` option.

**Parameters**

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

**Returns** An array filled with zeros.

**Return type** *cupy.ndarray*

**See also:**

numpy.zeros_like()

### cupy.full

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

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

This function currently does not support `order` option.

**Parameters**

- **shape** (*int or tuple of ints*) – Dimensionalities of the array.

- **fill_value** – A scalar value to fill a new array.
- **dtype** – Data type specifier.
- **order** (*{'C', 'F'}*) – Row-major (C-style) or column-major (Fortran-style) order.

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

**Return type** *cupy.ndarray*

**See also:**

`numpy.full()`

### cupy.full_like

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

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

This function currently does not support `subok` option.

**Parameters**

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

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

**Return type** *cupy.ndarray*

**See also:**

`numpy.full_like()`

## Creation from other data

| | |
|---|---|
| *cupy.array* | Creates an array on the current device. |
| *cupy.asarray* | Converts an object to array. |
| *cupy.asanyarray* | Converts an object to array. |
| *cupy.ascontiguousarray* | Returns a C-contiguous array. |
| *cupy.copy* | Creates a copy of a given array on the current device. |
| *cupy.fromfile* | Reads an array from a file. |

### cupy.asanyarray

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

Converts an object to array.

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

**See also:**

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

### cupy.ascontiguousarray

cupy.**ascontiguousarray**(*a*, *dtype=None*)

Returns a C-contiguous array.

**Parameters**

- **a** (cupy.ndarray) – Source array.
- **dtype** – Data type specifier.

**Returns** If no copy is required, it returns a. Otherwise, it returns a copy of a.

**Return type** *cupy.ndarray*

**See also:**

numpy.ascontiguousarray()

### cupy.copy

cupy.**copy**(*a*, *order='K'*)

Creates a copy of a given array on the current device.

This function allocates the new array on the current device. If the given array is allocated on the different device, then this function tries to copy the contents over the devices.

**Parameters**

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

**Returns** The copy of a on the current device.

**Return type** *cupy.ndarray*

**See also:**

numpy.copy(), *cupy.ndarray.copy()*

### cupy.fromfile

cupy.**fromfile**(**args*, ***kwargs*)

Reads an array from a file.

**Note:** Uses NumPy's fromfile and coerces the result to a CuPy array.

**See also:**

numpy.fromfile()

### Numerical ranges

| | |
|---|---|
| *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 [start, stop). The first three arguments are mapped like the range built-in function, i.e. start and step are optional.

**Parameters**

- **start** – Start of the interval.
- **stop** – End of the interval.
- **step** – Step width between each pair of consecutive values.
- **dtype** – Data type specifier. It is inferred from other arguments by default.

**Returns** The 1-D array of range values.

**Return type** *cupy.ndarray*

**See also:**

numpy.arange()

### cupy.linspace

cupy.**linspace**(*start*, *stop*, *num=50*, *endpoint=True*, *retstep=False*, *dtype=None*)

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** – 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.
- **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.

**Returns** The 1-D array of ranged values.

**Return type** *cupy.ndarray*

### cupy.logspace

`cupy.`**`logspace`**(*start*, *stop*, *num=50*, *endpoint=True*, *base=10.0*, *dtype=None*)

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

Instead of specifying the step width like *cupy.arange()*, this function requires the total number of elements specified.

**Parameters**

- **start** – Start of the interval.
- **stop** – End of the interval.
- **num** – Number of elements.
- **endpoint** (*bool*) – If `True`, the stop value is included as the last element. Otherwise, the stop value is omitted.
- **base** (*float*) – Base of the log space. The step sizes between the elements on a log-scale are the same as `base`.
- **dtype** – Data type specifier. It is inferred from the start and stop arguments by default.

**Returns** The 1-D array of ranged values.

**Return type** *cupy.ndarray*

**See also:**

`numpy.logspace()`

### cupy.meshgrid

`cupy.`**`meshgrid`**(**xi*, ***kwargs*)

Return coordinate matrices from coordinate vectors.

Given one-dimensional coordinate arrays `x1, x2, ... , xn` this function makes N-D grids.

For one-dimensional arrays `x1, x2, ... , xn` with lengths `Ni = len(xi)`, this function returns `(N1, N2, N3, ..., Nn)` shaped arrays if indexing='ij' or `(N2, N1, N3, ..., Nn)` shaped arrays if indexing='xy'.

Unlike NumPy, CuPy currently only supports 1-D arrays as inputs.

**Parameters**

- **xi** (*tuple of ndarrays*) – 1-D arrays representing the coordinates of a grid.
- **indexing** (*{'xy', 'ij'}, optional*) – Cartesian ('xy', default) or matrix ('ij') indexing of output.

- **sparse** (*bool, optional*) – If `True`, a sparse grid is returned in order to conserve memory. Default is `False`.
- **copy** (*bool, optional*) – If `False`, a view into the original arrays are returned. Default is `True`.

**Returns** list of cupy.ndarray

**See also:**

`numpy.meshgrid()`

### cupy.mgrid

`cupy.`**`mgrid = <cupy.creation.ranges.nd_grid object>`**

Construct a multi-dimensional "meshgrid".

`grid = nd_grid()` creates an instance which will return a mesh-grid when indexed. The dimension and number of the output arrays are equal to the number of indexing dimensions. If the step length is not a complex number, then the stop is not inclusive.

However, if the step length is a **complex number** (e.g. 5j), then the integer part of its magnitude is interpreted as specifying the number of points to create between the start and stop values, where the stop value **is inclusive**.

If instantiated with an argument of `sparse=True`, the mesh-grid is open (or not fleshed out) so that only one-dimension of each returned argument is greater than 1.

**Parameters** **sparse** (*bool, optional*) – Whether the grid is sparse or not. Default is False.

**See also:**

`numpy.mgrid` and `numpy.ogrid`

### cupy.ogrid

`cupy.`**`ogrid = <cupy.creation.ranges.nd_grid object>`**

Construct a multi-dimensional "meshgrid".

`grid = nd_grid()` creates an instance which will return a mesh-grid when indexed. The dimension and number of the output arrays are equal to the number of indexing dimensions. If the step length is not a complex number, then the stop is not inclusive.

However, if the step length is a **complex number** (e.g. 5j), then the integer part of its magnitude is interpreted as specifying the number of points to create between the start and stop values, where the stop value **is inclusive**.

If instantiated with an argument of `sparse=True`, the mesh-grid is open (or not fleshed out) so that only one-dimension of each returned argument is greater than 1.

**Parameters** **sparse** (*bool, optional*) – Whether the grid is sparse or not. Default is False.

**See also:**

`numpy.mgrid` and `numpy.ogrid`

### Matrix creation

| | |
|---|---|
| *cupy.diag* | Returns a diagonal or a diagonal array. |

Continued on next page

Table 13 – continued from previous page

| | |
|---|---|
| *cupy.diagflat* | Creates a diagonal array from the flattened input. |
| *cupy.tri* | Creates an array with ones at and below the given diagonal. |
| *cupy.tril* | Returns a lower triangle of an array. |
| *cupy.triu* | Returns an upper triangle of an array. |

### cupy.diag

cupy.**diag**(*v*, *k=0*)

Returns a diagonal or a diagonal array.

**Parameters**

- **v** (*array-like*) – Array or array-like object.
- **k** (*int*) – Index of diagonals. Zero indicates the main diagonal, a positive value an upper diagonal, and a negative value a lower diagonal.

**Returns** If v indicates a 1-D array, then it returns a 2-D array with the specified diagonal filled by v. If v indicates a 2-D array, then it returns the specified diagonal of v. In latter case, if v is a *cupy.ndarray* object, then its view is returned.

**Return type** *cupy.ndarray*

**See also:**

numpy.diag()

### cupy.diagflat

cupy.**diagflat**(*v*, *k=0*)

Creates a diagonal array from the flattened input.

**Parameters**

- **v** (*array-like*) – Array or array-like object.
- **k** (*int*) – Index of diagonals. See *cupy.diag()* for detail.

**Returns** A 2-D diagonal array with the diagonal copied from v.

**Return type** *cupy.ndarray*

**See also:**

numpy.diagflat()

### cupy.tri

cupy.**tri**(*N*, *M=None*, *k=0*, *dtype=<class 'float'>*)

Creates an array with ones at and below the given diagonal.

**Parameters**

- **N** (*int*) – Number of rows.
- **M** (*int*) – Number of columns. M == N by default.

- **k** (*int*) – The sub-diagonal at and below which the array is filled. Zero is the main diagonal, a positive value is above it, and a negative value is below.
- **dtype** – Data type specifier.

**Returns** An array with ones at and below the given diagonal.

**Return type** *cupy.ndarray*

**See also:**

`numpy.tri()`

#### cupy.tril

`cupy.`**`tril`**(*m*, *k=0*)

Returns a lower triangle of an array.

**Parameters**

- **m** (*array-like*) – Array or array-like object.
- **k** (*int*) – The diagonal above which to zero elements. Zero is the main diagonal, a positive value is above it, and a negative value is below.

**Returns** A lower triangle of an array.

**Return type** *cupy.ndarray*

**See also:**

`numpy.tril()`

#### cupy.triu

`cupy.`**`triu`**(*m*, *k=0*)

Returns an upper triangle of an array.

**Parameters**

- **m** (*array-like*) – Array or array-like object.
- **k** (*int*) – The diagonal below which to zero elements. Zero is the main diagonal, a positive value is above it, and a negative value is below.

**Returns** An upper triangle of an array.

**Return type** *cupy.ndarray*

**See also:**

`numpy.triu()`

### 3.3.2 Array Manipulation Routines

#### Basic operations

| | |
|---|---|
| *cupy.copyto* | Copies values from one array to another with broadcasting. |

### cupy.copyto

cupy.**copyto**(*dst*, *src*, *casting='same_kind'*, *where=None*)

Copies values from one array to another with broadcasting.

This function can be called for arrays on different devices. In this case, casting, `where`, and broadcasting is not supported, and an exception is raised if these are used.

**Parameters**

- **dst** (*cupy.ndarray*) – Target array.
- **src** (*cupy.ndarray*) – Source array.
- **casting** (*str*) – Casting rule. See `numpy.can_cast()` for detail.
- **where** (*cupy.ndarray of bool*) – If specified, this array acts as a mask, and an element is copied only if the corresponding element of `where` is True.

**See also:**

`numpy.copyto()`

### Changing array shape

| | |
|---|---|
| *cupy.reshape* | Returns an array with new shape and same elements. |
| *cupy.ravel* | Returns a flattened array. |

### cupy.reshape

cupy.**reshape**(*a*, *newshape*, *order='C'*)

Returns an array with new shape and same elements.

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

**Parameters**

- **a** (*cupy.ndarray*) – Array to be reshaped.
- **newshape** (*int or tuple of ints*) – The new shape of the array to return. If it is an integer, then it is treated as a tuple of length one. It should be compatible with `a.size`. One of the elements can be -1, which is automatically replaced with the appropriate value to make the shape compatible with `a.size`.
- **order** (*{'C', 'F', 'A'}*) – Read the elements of `a` using this index order, and place the elements into the reshaped array using this index order. 'C' means to read / write the elements using C-like index order, with the last axis index changing fastest, back to the first axis index changing slowest. 'F' means to read / write the elements using Fortran-like index order, with the first index changing fastest, and the last index changing slowest. Note that the 'C' and 'F' options take no account of the memory layout of the underlying array, and only refer to the order of indexing. 'A' means to read / write the elements in Fortran-like index order if a is Fortran contiguous in memory, C-like order otherwise.

**Returns** A reshaped view of `a` if possible, otherwise a copy.

**Return type** *cupy.ndarray*

**See also:**

`numpy.reshape()`

### cupy.ravel

cupy.**ravel**(*a*, *order='C'*)
Returns a flattened array.

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

This function currently does not support the `order = 'K'` option.

**Parameters**

- **a** (`cupy.ndarray`) – Array to be flattened.
- **order** (*{'C', 'F', 'A'}*) – Read the elements of `a` using this index order, and place the elements into the reshaped array using this index order. 'C' means to read / write the elements using C-like index order, with the last axis index changing fastest, back to the first axis index changing slowest. 'F' means to read / write the elements using Fortran-like index order, with the first index changing fastest, and the last index changing slowest. Note that the 'C' and 'F' options take no account of the memory layout of the underlying array, and only refer to the order of indexing. 'A' means to read / write the elements in Fortran-like index order if a is Fortran contiguous in memory, C-like order otherwise.

**Returns** A flattened view of `a` if possible, otherwise a copy.

**Return type** *cupy.ndarray*

**See also:**

`numpy.ravel()`

### Transpose-like operations

| | |
|---|---|
| *cupy.moveaxis* | Moves axes of an array to new positions. |
| *cupy.rollaxis* | Moves the specified axis backwards to the given place. |
| *cupy.swapaxes* | Swaps the two axes. |
| *cupy.transpose* | Permutes the dimensions of an array. |

### cupy.moveaxis

cupy.**moveaxis**(*a*, *source*, *destination*)
Moves axes of an array to new positions.

Other axes remain in their original order.

**Parameters**

- **a** (`cupy.ndarray`) – Array whose axes should be reordered.
- **source** (*int or sequence of int*) – Original positions of the axes to move. These must be unique.
- **destination** (*int or sequence of int*) – Destination positions for each of the original axes. These must also be unique.

**Returns** Array with moved axes. This array is a view of the input array.

**Return type** *cupy.ndarray*

**See also:**

`numpy.moveaxis()`

### cupy.rollaxis

`cupy.`**`rollaxis`**(*a*, *axis*, *start=0*)

Moves the specified axis backwards to the given place.

**Parameters**

- **a** (`cupy.ndarray`) – Array to move the axis.
- **axis** (*int*) – The axis to move.
- **start** (*int*) – The place to which the axis is moved.

**Returns** A view of `a` that the axis is moved to `start`.

**Return type** *cupy.ndarray*

**See also:**

`numpy.rollaxis()`

### cupy.swapaxes

`cupy.`**`swapaxes`**(*a*, *axis1*, *axis2*)

Swaps the two axes.

**Parameters**

- **a** (`cupy.ndarray`) – Array to swap the axes.
- **axis1** (*int*) – The first axis to swap.
- **axis2** (*int*) – The second axis to swap.

**Returns** A view of `a` that the two axes are swapped.

**Return type** *cupy.ndarray*

**See also:**

`numpy.swapaxes()`

### cupy.transpose

`cupy.`**`transpose`**(*a*, *axes=None*)

Permutes the dimensions of an array.

**Parameters**

- **a** (`cupy.ndarray`) – Array to permute the dimensions.
- **axes** (*tuple of ints*) – Permutation of the dimensions. This function reverses the shape by default.

**Returns** A view of `a` that the dimensions are permuted.

**Return type** *cupy.ndarray*

**See also:**

`numpy.transpose()`

**See also:**

*`cupy.ndarray.T`*

### Changing number of dimensions

| | |
|---|---|
| *`cupy.atleast_1d`* | Converts arrays to arrays with dimensions >= 1. |
| *`cupy.atleast_2d`* | Converts arrays to arrays with dimensions >= 2. |
| *`cupy.atleast_3d`* | Converts arrays to arrays with dimensions >= 3. |
| *`cupy.broadcast`* | Object that performs broadcasting. |
| *`cupy.broadcast_to`* | Broadcast an array to a given shape. |
| *`cupy.broadcast_arrays`* | Broadcasts given arrays. |
| *`cupy.expand_dims`* | Expands given arrays. |
| *`cupy.squeeze`* | Removes size-one axes from the shape of an array. |

### 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** **arys** (*tuple of arrays*) – Arrays to be converted. All arguments must be *cupy.ndarray* objects.

**Returns** If there are only one input, then it returns its converted version. Otherwise, it returns a list of converted arrays.

**See also:**

`numpy.atleast_3d()`

### cupy.broadcast

**class** cupy.**broadcast**(**arrays*)

Object that performs broadcasting.

CuPy actually uses this class to support broadcasting in various operations. Note that this class does not provide an iterator.

**Parameters** **arrays** (*tuple of arrays*) – Arrays to be broadcasted.

**Variables**

- ***shape*** (*tuple of ints*) – The broadcasted shape.
- ***nd*** (*int*) – Number of dimensions of the broadcasted shape.
- ***size*** (*int*) – Total size of the broadcasted shape.
- ***values*** (*list of arrays*) – The broadcasted arrays.

**See also:**

`numpy.broadcast`

#### Methods

#### Attributes

**nd**

**shape**

**size**

**values**

### cupy.broadcast_to

cupy.**broadcast_to**(*array*, *shape*)
Broadcast an array to a given shape.

**Parameters**

- **array** (cupy.ndarray) – Array to broadcast.
- **shape** (*tuple of int*) – The shape of the desired array.

**Returns** Broadcasted view.

**Return type** *cupy.ndarray*

**See also:**

numpy.broadcast_to()

### cupy.broadcast_arrays

cupy.**broadcast_arrays**(**args*)
Broadcasts given arrays.

**Parameters** **args** (*tuple of arrays*) – Arrays to broadcast for each other.

**Returns** A list of broadcasted arrays.

**Return type** list

**See also:**

numpy.broadcast_arrays()

### cupy.expand_dims

cupy.**expand_dims**(*a*, *axis*)
Expands given arrays.

**Parameters**

- **a** (cupy.ndarray) – Array to be expanded.
- **axis** (*int*) – Position where new axis is to be inserted.

**Returns**

**The number of dimensions is one greater than that of** the input array.

**Return type** *cupy.ndarray*

**See also:**

numpy.expand_dims()

### cupy.squeeze

cupy.**squeeze**(*a*, *axis=None*)
Removes size-one axes from the shape of an array.

**Parameters**

- **a** (cupy.ndarray) – Array to be reshaped.
- **axis** (*int or tuple of ints*) – Axes to be removed. This function removes all size-one axes by default. If one of the specified axes is not of size one, an exception is raised.

**Returns** An array without (specified) size-one axes.

**Return type** *cupy.ndarray*

**See also:**

numpy.squeeze()

### Changing kind of array

| | |
|---|---|
| *cupy.asarray* | Converts an object to array. |
| *cupy.asanyarray* | Converts an object to array. |
| *cupy.asfortranarray* | Return an array laid out in Fortran order in memory. |
| *cupy.ascontiguousarray* | Returns a C-contiguous array. |

### 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()

### Joining arrays

| | |
|---|---|
| *cupy.concatenate* | Joins arrays along an axis. |
| *cupy.stack* | Stacks arrays along a new axis. |
| *cupy.column_stack* | Stacks 1-D and 2-D arrays as columns into a 2-D array. |
| *cupy.dstack* | Stacks arrays along the third axis. |
| *cupy.hstack* | Stacks arrays horizontally. |
| *cupy.vstack* | Stacks arrays vertically. |

### cupy.concatenate

cupy.**concatenate**(*tup*, *axis=0*)

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.

**Returns** Joined array.

**Return type** *cupy.ndarray*

**See also:**

`numpy.concatenate()`

### cupy.stack

cupy.**stack**(*tup*, *axis=0*)

Stacks arrays along a new axis.

**Parameters**

- **tup** (*sequence of arrays*) – Arrays to be stacked.
- **axis** (*int*) – Axis along which the arrays are stacked.

**Returns** Stacked array.

**Return type** *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

| | |
|---|---|
| *cupy.split* | Splits an array into multiple sub arrays along a given axis. |
| *cupy.array_split* | Splits an array into multiple sub arrays along a given axis. |
| *cupy.dsplit* | Splits an array into multiple sub arrays along the third axis. |
| *cupy.hsplit* | Splits an array into multiple sub arrays horizontally. |
| *cupy.vsplit* | Splits an array into multiple sub arrays along the first axis. |

### cupy.split

cupy.**split**(*ary*, *indices_or_sections*, *axis=0*)
Splits an array into multiple sub arrays along a given axis.

**Parameters**

- **ary** (cupy.ndarray) – Array to split.
- **indices_or_sections** (*int or sequence of ints*) – A value indicating how to divide the axis. If it is an integer, then is treated as the number of sections, and the axis is evenly divided. Otherwise, the integers indicate indices to split at. Note that the sequence on the device memory is not allowed.
- **axis** (*int*) – Axis along which the array is split.

**Returns** A list of sub arrays. Each array is a view of the corresponding input array.

**See also:**

numpy.split()

### cupy.array_split

cupy.**array_split**(*ary*, *indices_or_sections*, *axis=0*)
Splits an array into multiple sub arrays along a given axis.

This function is almost equivalent to *cupy.split()*. The only difference is that this function allows an integer sections that does not evenly divide the axis.

**See also:**

*cupy.split()* for more detail, numpy.array_split()

### cupy.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

| | |
|---|---|
| *cupy.tile* | Construct an array by repeating A the number of times given by reps. |
| *cupy.repeat* | Repeat arrays along an axis. |

### cupy.tile

cupy.**tile**(*A*, *reps*)
Construct an array by repeating A the number of times given by reps.

**Parameters**

- **A** (`cupy.ndarray`) – Array to transform.
- **reps** (*int or tuple*) – The number of repeats.

**Returns** Transformed array with repeats.

**Return type** *cupy.ndarray*

**See also:**

`numpy.tile()`

### cupy.repeat

cupy.**repeat**(*a*, *repeats*, *axis=None*)
Repeat arrays along an axis.

**Parameters**

- **a** (`cupy.ndarray`) – Array to transform.
- **repeats** (*int, list or tuple*) – The number of repeats.
- **axis** (*int*) – The axis to repeat.

**Returns** Transformed array with repeats.

**Return type** *cupy.ndarray*

**See also:**

`numpy.repeat()`

### Adding and removing elements

| | |
|---|---|
| *cupy.unique* | Find the unique elements of an array. |

### cupy.unique

cupy.**unique**(*ar*, *return_index=False*, *return_inverse=False*, *return_counts=False*, *axis=None*)

Find the unique elements of an array.

Returns the sorted unique elements of an array. There are three optional outputs in addition to the unique elements:

- the indices of the input array that give the unique values
- the indices of the unique array that reconstruct the input array
- the number of times each unique value comes up in the input array

**Parameters**

- **ar** (*array_like*) – Input array. This will be flattened if it is not already 1-D.
- **return_index** (*bool, optional*) – If True, also return the indices of *ar* (along the specified axis, if provided, or in the flattened array) that result in the unique array.
- **return_inverse** (*bool, optional*) – If True, also return the indices of the unique array (for the specified axis, if provided) that can be used to reconstruct *ar*.
- **return_counts** (*bool, optional*) – If True, also return the number of times each unique item appears in *ar*.
- **axis** (*int or None, optional*) – Not supported yet.

**Returns**

If there are no optional outputs, it returns the *cupy.ndarray* of the sorted unique values. Otherwise, it returns the tuple which contains the sorted unique values and followings.

- The indices of the first occurrences of the unique values in the original array. Only provided if *return_index* is True.
- The indices to reconstruct the original array from the unique array. Only provided if *return_inverse* is True.
- The number of times each of the unique values comes up in the original array. Only provided if *return_counts* is True.

**Return type** *cupy.ndarray* or tuple

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

**See also:**

*numpy.unique()*

### Rearranging elements

| | |
|---|---|
| *cupy.flip* | Reverse the order of elements in an array along the given axis. |
| *cupy.fliplr* | Flip array in the left/right direction. |
| *cupy.flipud* | Flip array in the up/down direction. |
| *cupy.reshape* | Returns an array with new shape and same elements. |
| *cupy.roll* | Roll array elements along a given axis. |
| *cupy.rot90* | Rotate an array by 90 degrees in the plane specified by axes. |

### cupy.flip

cupy.**flip**(*a*, *axis*)

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*) – Axis in array, which entries are reversed.

**Returns** Output array.

**Return type** *ndarray*

**See also:**

`numpy.flip()`

### cupy.fliplr

cupy.**fliplr**(*a*)

Flip array in the left/right direction.

Flip the entries in each row in the left/right direction. Columns are preserved, but appear in a different order than before.

**Parameters** **a** (ndarray) – Input array.

**Returns** Output array.

**Return type** *ndarray*

**See also:**

`numpy.fliplr()`

### cupy.flipud

cupy.**flipud**(*a*)

Flip array in the up/down direction.

Flip the entries in each column in the up/down direction. Rows are preserved, but appear in a different order than before.

**Parameters** **a** (ndarray) – Input array.

**Returns** Output array.

**Return type** *ndarray*

**See also:**

`numpy.flipud()`

### cupy.roll

`cupy.roll`(*a*, *shift*, *axis=None*)

Roll array elements along a given axis.

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

**Parameters**

- **a** (`ndarray`) – Array to be rolled.
- **shift** (*int or tuple of int*) – The number of places by which elements are shifted. If a tuple, then *axis* must be a tuple of the same size, and each of the given axes is shifted by the corresponding number. If an int while *axis* is a tuple of ints, then the same value is used for all given axes.
- **axis** (*int or tuple of int or None*) – The axis along which elements are shifted. By default, the array is flattened before shifting, after which the original shape is restored.

**Returns** Output array.

**Return type** *ndarray*

**See also:**

`numpy.roll()`

### cupy.rot90

`cupy.rot90`(*a*, *k=1*, *axes=(0, 1)*)

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

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

**Parameters**

- **a** (`ndarray`) – Array of two or more dimensions.
- **k** (*int*) – Number of times the array is rotated by 90 degrees.
- **axes** – (tuple of ints): The array is rotated in the plane defined by the axes. Axes must be different.

**Returns** Output array.

**Return type** *ndarray*

**See also:**

`numpy.rot90()`

### 3.3.3 Binary Operations

#### Elementwise bit operations

| | |
|---|---|
| *cupy.bitwise_and* | Computes the bitwise AND of two arrays elementwise. |
| *cupy.bitwise_or* | Computes the bitwise OR of two arrays elementwise. |
| *cupy.bitwise_xor* | Computes the bitwise XOR of two arrays elementwise. |
| *cupy.invert* | Computes the bitwise NOT of an array elementwise. |
| *cupy.left_shift* | Shifts the bits of each integer element to the left. |
| *cupy.right_shift* | Shifts the bits of each integer element to the right. |

#### Bit packing

| | |
|---|---|
| *cupy.packbits* | Packs the elements of a binary-valued array into bits in a uint8 array. |
| *cupy.unpackbits* | Unpacks elements of a uint8 array into a binary-valued output array. |

#### cupy.packbits

cupy.**packbits**(*myarray*)

Packs the elements of a binary-valued array into bits in a uint8 array.

This function currently does not support `axis` option.

**Parameters** **myarray** (cupy.ndarray) – Input array.

**Returns** The packed array.

**Return type** *cupy.ndarray*

**Note:** When the input array is empty, this function returns a copy of it, i.e., the type of the output array is not necessarily always uint8. This exactly follows the NumPy's behaviour (as of version 1.11), alghough 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* | Return the binary representation of the input number as a string. |

#### cupy.binary_repr

cupy.**binary_repr**(*num*, *width=None*)
Return the binary representation of the input number as a string.

**See also:**

numpy.binary_repr()

### 3.3.4 Data Type Routines

| | |
|---|---|
| *cupy.can_cast* | Returns True if cast between data types can occur according to the casting rule. |
| *cupy.result_type* | Returns the type that results from applying the NumPy type promotion rules to the arguments. |
| *cupy.common_type* | Return a scalar type which is common to the input arrays. |

#### cupy.can_cast

cupy.**can_cast**(*from_*, *to*, *casting='safe'*)
Returns True if cast between data types can occur according to the casting rule. If from is a scalar or array scalar, also returns True if the scalar value can be cast without overflow or truncation to an integer.

**See also:**

numpy.can_cast()

#### cupy.result_type

cupy.**result_type**(**arrays_and_dtypes*)
Returns the type that results from applying the NumPy type promotion rules to the arguments.

**See also:**

numpy.result_type()

#### cupy.common_type

cupy.**common_type**(**arrays*)
Return a scalar type which is common to the input arrays.

**See also:**

numpy.common_type()

| |
|---|
| cupy.promote_types (alias of numpy.promote_types()) |
| cupy.min_scalar_type (alias of numpy.min_scalar_type()) |
| cupy.obj2sctype (alias of numpy.obj2sctype()) |

#### 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.issubsctype (alias of numpy.issubsctype()) |
| 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()) |

### 3.3.5 FFT Functions

#### Standard FFTs

| | |
|---|---|
| *cupy.fft.fft* | Compute the one-dimensional FFT. |
| *cupy.fft.ifft* | Compute the one-dimensional inverse FFT. |
| *cupy.fft.fft2* | Compute the two-dimensional FFT. |
| *cupy.fft.ifft2* | Compute the two-dimensional inverse FFT. |
| *cupy.fft.fftn* | Compute the N-dimensional FFT. |
| *cupy.fft.ifftn* | Compute the N-dimensional inverse FFT. |

### cupy.fft.fft

cupy.fft.**fft**(*a*, *n=None*, *axis=-1*, *norm=None*)
Compute the one-dimensional FFT.

**Parameters**

- **a** (cupy.ndarray) – Array to be transform.
- **n** (*None or int*) – Length of the transformed axis of the output. If n is not given, the length of the input along the axis specified by axis is used.
- **axis** (*int*) – Axis over which to compute the FFT.
- **norm** (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*, *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*, *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*, *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

| | |
|---|---|
| *cupy.fft.rfft* | Compute the one-dimensional FFT for real input. |
| *cupy.fft.irfft* | Compute the one-dimensional inverse FFT for real input. |
| *cupy.fft.rfft2* | Compute the two-dimensional FFT for real input. |
| *cupy.fft.irfft2* | Compute the two-dimensional inverse FFT for real input. |
| *cupy.fft.rfftn* | Compute the N-dimensional FFT for real input. |
| *cupy.fft.irfftn* | Compute the N-dimensional inverse FFT for real input. |

### cupy.fft.rfft

cupy.fft.**rfft**(*a*, *n=None*, *axis=-1*, *norm=None*)
Compute the one-dimensional FFT for real input.

**Parameters**

- **a** (cupy.ndarray) – Array to be transform.
- **n** (*None or int*) – Number of points along transformation axis in the input to use. If n is not given, the length of the input along the axis specified by axis is used.
- **axis** (*int*) – Axis over which to compute the FFT.
- **norm** (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*

**Warning:** The input array may be modified in CUDA 10.1 and above.

**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*

**Warning:** The input array may be modified in CUDA 10.1 and above.

**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*

> **Warning:** The input array may be modified in CUDA 10.1 and above.

**See also:**

`numpy.fft.irfftn()`

### Hermitian FFTs

| | |
|---|---|
| *cupy.fft.hfft* | Compute the FFT of a signal that has Hermitian symmetry. |
| *cupy.fft.ihfft* | Compute the FFT of a signal that has Hermitian symmetry. |

#### cupy.fft.hfft

`cupy.fft.`**`hfft`**(*a*, *n=None*, *axis=-1*, *norm=None*)

Compute the FFT of a signal that has Hermitian symmetry.

**Parameters**

- **a** (`cupy.ndarray`) – Array to be transform.
- **n** (*None or int*) – Length of the transformed axis of the output. For `n` output points, `n//2+1` input points are necessary. If `n` is not given, it is determined from the length of the input along the axis specified by `axis`.
- **axis** (*int*) – Axis over which to compute the FFT.
- **norm** (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.hfft()`

#### cupy.fft.ihfft

`cupy.fft.`**`ihfft`**(*a*, *n=None*, *axis=-1*, *norm=None*)

Compute the FFT of a signal that has Hermitian symmetry.

**Parameters**

- **a** (`cupy.ndarray`) – Array to be transform.

- **n** (*None or int*) – Number of points along transformation axis in the input to use. If `n` is not given, the length of the input along the axis specified by `axis` is used.
- **axis** (*int*) – Axis over which to compute the FFT.
- **norm** (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.ihfft()`

### Helper routines

| | |
|---|---|
| *cupy.fft.fftfreq* | Return the FFT sample frequencies. |
| *cupy.fft.rfftfreq* | Return the FFT sample frequencies for real input. |
| *cupy.fft.fftshift* | Shift the zero-frequency component to the center of the spectrum. |
| *cupy.fft.ifftshift* | The inverse of *fftshift()*. |

### cupy.fft.fftfreq

`cupy.fft.`**`fftfreq`**(*n*, *d=1.0*)

Return the FFT sample frequencies.

**Parameters**

- **n** (*int*) – Window length.
- **d** (*scalar*) – Sample spacing.

**Returns** Array of length `n` containing the sample frequencies.

**Return type** *cupy.ndarray*

**See also:**

`numpy.fft.fftfreq()`

### cupy.fft.rfftfreq

`cupy.fft.`**`rfftfreq`**(*n*, *d=1.0*)

Return the FFT sample frequencies for real input.

**Parameters**

- **n** (*int*) – Window length.
- **d** (*scalar*) – Sample spacing.

**Returns** Array of length `n//2+1` containing the sample frequencies.

**Return type** *cupy.ndarray*

**See also:**

`numpy.fft.rfftfreq()`

### cupy.fft.fftshift

cupy.fft.**fftshift**(*x*, *axes=None*)

Shift the zero-frequency component to the center of the spectrum.

**Parameters**

- **x** (cupy.ndarray) – Input array.
- **axes** (*int or tuple of ints*) – Axes over which to shift. Default is None, which shifts all axes.

**Returns** The shifted array.

**Return type** *cupy.ndarray*

**See also:**

numpy.fft.fftshift()

### cupy.fft.ifftshift

cupy.fft.**ifftshift**(*x*, *axes=None*)

The inverse of *fftshift()*.

**Parameters**

- **x** (cupy.ndarray) – Input array.
- **axes** (*int or tuple of ints*) – Axes over which to shift. Default is None, which shifts all axes.

**Returns** The shifted array.

**Return type** *cupy.ndarray*

**See also:**

numpy.fft.ifftshift()

### Normalization

The default normalization has the direct transforms unscaled and the inverse transforms are scaled by $1/n$. If the ketyword argument norm is "ortho", both transforms will be scaled by $1/\sqrt{n}$.

### Code compatibility features

FFT functions of NumPy alway 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.

### 3.3.6 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.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* | |
| *cupy.diag* | Returns a diagonal or a diagonal array. |
| *cupy.diagonal* | Returns specified diagonals. |
| *cupy.lib.stride_tricks.as_strided* | Create a view into the array with the given shape and strides. |
| *cupy.place* | Change elements of an array based on conditional and input values. |
| *cupy.put* | Replaces specified elements of an array with given values. |
| *cupy.fill_diagonal* | Fills the main diagonal of the given array of any dimensionality. |

#### cupy.c_

cupy.**c_ = <cupy.indexing.generate.CClass object>**

#### cupy.r_

cupy.**r_ = <cupy.indexing.generate.RClass object>**

#### cupy.nonzero

cupy.**nonzero**(*a*)

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

Returns a tuple of arrays, one for each dimension of a, containing the indices of the non-zero elements in that dimension.

**Parameters** **a** (cupy.ndarray) – array

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

**Return type** tuple of arrays

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

**See also:**

`numpy.nonzero()`

### cupy.where

`cupy.`**`where`**(*condition*, *x=None*, *y=None*)

Return elements, either from x or y, depending on condition.

If only condition is given, return `condition.nonzero()`.

**Parameters**

- **`condition`** (`cupy.ndarray`) – When True, take x, otherwise take y.
- **`x`** (`cupy.ndarray`) – Values from which to choose on `True`.
- **`y`** (`cupy.ndarray`) – Values from which to choose on `False`.

**Returns**

**Each element of output contains elements of `x` when** `condition` is `True`, otherwise elements of `y`. If only `condition` is given, return the tuple `condition.nonzero()`, the indices where `condition` is True.

**Return type** *cupy.ndarray*

**Warning:** This function may synchronize the device if both `x` and `y` are omitted.

**See also:**

`numpy.where()`

### cupy.indices

`cupy.`**`indices`**(*dimensions*, *dtype=<class 'int'>*)

Returns an array representing the indices of a grid.

Computes an array where the subarrays contain index values 0,1,... varying only along the corresponding axis.

**Parameters**

- **`dimensions`** – The shape of the grid.
- **`dtype`** – Data type specifier. It is int by default.

**Returns** The array of grid indices, `grid.shape = (len(dimensions),) + tuple(dimensions)`.

**Return type** *ndarray*

#### Examples

```
>>> grid = cupy.indices((2, 3))
>>> grid.shape
(2, 2, 3)
>>> grid[0]         # row indices
array([[0, 0, 0],
       [1, 1, 1]])
>>> grid[1]         # column indices
array([[0, 1, 2],
       [0, 1, 2]])
```

**See also:**

`numpy.indices()`

### cupy.ix_

`cupy.`**`ix_`**`(*args)`

Construct an open mesh from multiple sequences.

This function takes N 1-D sequences and returns N outputs with N dimensions each, such that the shape is 1 in all but one dimension and the dimension with the non-unit shape value cycles through all N dimensions.

Using *ix_* one can quickly construct index arrays that will index the cross product. `a[cupy.ix_([1,3], [2,5])]` returns the array `[[a[1,2] a[1,5]], [a[3,2] a[3,5]]]`.

**Parameters** **`*args`** – 1-D sequences

**Returns** N arrays with N dimensions each, with N the number of input sequences. Together these arrays form an open mesh.

**Return type** tuple of ndarrays

#### Examples

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

**Warning:**

This function may synchronize the device.

**See also:**

`numpy.ix_()`

### cupy.unravel_index

`cupy.`**`unravel_index`**`(indices, dims, order='C')`

Converts array of flat indices into a tuple of coordinate arrays.

**Parameters**

- **indices** (cupy.ndarray) – An integer array whose elements are indices into the flattened version of an array of dimensions dims.
- **dims** (*tuple of ints*) – The shape of the array to use for unraveling indices.
- **order** (*'C' or 'F'*) – Determines whether the indices should be viewed as indexing in row-major (C-style) or column-major (Fortran-style) order.

**Returns** Each array in the tuple has the same shape as the indices array.

**Return type** tuple of ndarrays

### Examples

```
>>> cupy.unravel_index(cupy.array([22, 41, 37]), (7, 6))
(array([3, 6, 6]), array([4, 5, 1]))
>>> cupy.unravel_index(cupy.array([31, 41, 13]), (7, 6), order='F')
(array([3, 6, 6]), array([4, 5, 1]))
```

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

**See also:**

numpy.unravel_index()

## 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.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.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*.
- **vals** (*array-like*) – Values to put into *a*. Only the first N elements are used, where N is the number of True values in *mask*. If *vals* is smaller than N, it will be repeated, and if elements of *a* are to be masked, this sequence must be non-empty.

### Examples

```
>>> arr = np.arange(6).reshape(2, 3)
>>> np.place(arr, arr>2, [44, 55])
>>> arr
array([[ 0,  1,  2],
       [44, 55, 44]])
```

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

**See also:**

`numpy.place()`

### cupy.put

cupy.**put**(*a*, *ind*, *v*, *mode='wrap'*)
Replaces specified elements of an array with given values.

**Parameters**

- **a** (cupy.ndarray) – Target array.
- **ind** (*array-like*) – Target indices, interpreted as integers.
- **v** (*array-like*) – Values to place in *a* at target indices. If *v* is shorter than *ind* it will be repeated as necessary.
- **mode** (*str*) – How out-of-bounds indices will behave. Its value must be either *'raise'*, *'wrap'* or *'clip'*. Otherwise, TypeError is raised.

---

**Note:** Default *mode* is set to *'wrap'* to avoid unintended performance drop. If you need NumPy's behavior, please pass *mode='raise'* manually.

---

**See also:**

numpy.put()

### cupy.fill_diagonal

cupy.**fill_diagonal**(*a*, *val*, *wrap=False*)
Fills the main diagonal of the given array of any dimensionality.

For an array *a* with a.ndim > 2, the diagonal is the list of locations with indices a[i, i, ..., i] all identical. This function modifies the input array in-place, it does not return a value.

**Parameters**

- **a** (cupy.ndarray) – The array, at least 2-D.
- **val** (*scalar*) – The value to be written on the diagonal. Its type must be compatible with that of the array a.
- **wrap** (*bool*) – If specified, the diagonal is "wrapped" after N columns. This affects only tall matrices.

#### Examples

```
>>> a = cupy.zeros((3, 3), int)
>>> cupy.fill_diagonal(a, 5)
>>> a
array([[5, 0, 0],
       [0, 5, 0],
       [0, 0, 5]])
```

**See also:**

numpy.fill_diagonal()

### 3.3.7 Input and Output

#### NumPy binary files (NPY, NPZ)

| | |
|---|---|
| *cupy.load* | Loads arrays or pickled objects from `.npy`, `.npz` or pickled file. |
| *cupy.save* | Saves an array to a binary file in `.npy` format. |
| *cupy.savez* | Saves one or more arrays into a file in uncompressed `.npz` format. |
| *cupy.savez_compressed* | Saves one or more arrays into a file in compressed `.npz` format. |

#### cupy.load

cupy.**load**(*file*, *mmap_mode=None*, *allow_pickle=None*)

Loads arrays or pickled objects from `.npy`, `.npz` or pickled file.

This function just calls `numpy.load` and then sends the arrays to the current device. NPZ file is converted to NpzFile object, which defers the transfer to the time of accessing the items.

**Parameters**

- **file** (*file-like object or string*) – The file to read.
- **mmap_mode** (*None, 'r+', 'r', 'w+', 'c'*) – If not `None`, memory-map the file to construct an intermediate `numpy.ndarray` object and transfer it to the current device.
- **allow_pickle** (*bool*) – Allow loading pickled object arrays stored in 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

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

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()`

## 3.3.8 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. |

Continued on next page

Table 36 – continued from previous page

| | |
|---|---|
| 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*. |
| cupy.kron | Returns the kronecker product of two arrays. |

### cupy.cross

cupy.**cross**(*a*, *b*, *axisa=-1*, *axisb=-1*, *axisc=-1*, *axis=None*)

Returns the cross product of two vectors.

The cross product of a and b in $R^3$ is a vector perpendicular to both a and b. If a and b are arrays of vectors, the vectors are defined by the last axis of a and b by default, and these axes can have dimensions 2 or 3. Where the dimension of either a or b is 2, the third component of the input vector is assumed to be zero and the cross product calculated accordingly. In cases where both input vectors have dimension 2, the z-component of the cross product is returned.

**Parameters**

- **a** (cupy.ndarray) – Components of the first vector(s).
- **b** (cupy.ndarray) – Components of the second vector(s).
- **axisa** (*int, optional*) – Axis of a that defines the vector(s). By default, the last axis.
- **axisb** (*int, optional*) – Axis of b that defines the vector(s). By default, the last axis.
- **axisc** (*int, optional*) – Axis of c containing the cross product vector(s). Ignored if both input vectors have dimension 2, as the return is scalar. By default, the last axis.
- **axis** (*int, optional*) – If defined, the axis of a, b and c that defines the vector(s) and cross product(s). Overrides axisa, axisb and axisc.

**Returns** Vector cross product(s).

**Return type** *cupy.ndarray*

**See also:**

numpy.cross()

### cupy.dot

cupy.**dot**(*a*, *b*, *out=None*)

Returns a dot product of two arrays.

For arrays with more than one axis, it computes the dot product along the last axis of a and the second-to-last axis of b. This is just a matrix product if the both arrays are 2-D. For 1-D arrays, it uses their unique axis as an axis to take dot product over.

**Parameters**

- **a** (cupy.ndarray) – The left argument.

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

**Returns** The dot product of a and b.

**Return type** *cupy.ndarray*

**See also:**

numpy.dot()

### cupy.vdot

cupy.**vdot**(*a*, *b*)

Returns the dot product of two vectors.

The input arrays are flattened into 1-D vectors and then it performs inner product of these vectors.

**Parameters**

- **a** (cupy.ndarray) – The first argument.
- **b** (cupy.ndarray) – The second argument.

**Returns** Zero-dimensional array of the dot product result.

**Return type** *cupy.ndarray*

**See also:**

numpy.vdot()

### cupy.inner

cupy.**inner**(*a*, *b*)

Returns the inner product of two arrays.

It uses the last axis of each argument to take sum product.

**Parameters**

- **a** (cupy.ndarray) – The first argument.
- **b** (cupy.ndarray) – The second argument.

**Returns** The inner product of a and b.

**Return type** *cupy.ndarray*

**See also:**

numpy.inner()

### cupy.outer

cupy.**outer**(*a*, *b*, *out=None*)

Returns the outer product of two vectors.

The input arrays are flattened into 1-D vectors and then it performs outer product of these vectors.

**Parameters**

- **a** (cupy.ndarray) – The first argument.
- **b** (cupy.ndarray) – The second argument.
- **out** (cupy.ndarray) – Output array.

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

**Return type** *cupy.ndarray*

**See also:**

numpy.outer()

### cupy.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*.

---

..seealso:: `numpy.linalg.matrix_power()`

### cupy.kron

cupy.**kron**(*a*, *b*)

Returns the kronecker product of two arrays.

**Parameters**

- **a** (ndarray) – The first argument.
- **b** (ndarray) – The second argument.

**Returns** Output array.

**Return type** *ndarray*

**See also:**

numpy.kron()

## Decompositions

| | |
|---|---|
| *cupy.linalg.cholesky* | Cholesky decomposition. |
| *cupy.linalg.qr* | QR decomposition. |
| *cupy.linalg.svd* | Singular Value Decomposition. |

### cupy.linalg.cholesky

cupy.linalg.**cholesky**(*a*)

Cholesky decomposition.

Decompose a given two-dimensional square matrix into `L * L.T`, where `L` is a lower-triangular matrix and `.T` is a conjugate transpose operator.

**Parameters** **a** (cupy.ndarray) – The input matrix with dimension `(N, N)`

**Returns** The lower-triangular matrix.

**Return type** *cupy.ndarray*

**Warning:** This function calls one or more cuSOLVER routine(s) which may yield invalid results if input conditions are not met. To detect these invalid results, you can set the *linalg* configuration to a value that is not *ignore* in `cupyx.errstate()` or `cupyx.seterr()`.

**See also:**

numpy.linalg.cholesky()

### cupy.linalg.qr

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

QR decomposition.

Decompose a given two-dimensional matrix into `Q * R`, where `Q` is an orthonormal and `R` is an upper-triangular matrix.

**Parameters**

- **a** (cupy.ndarray) – The input matrix.
- **mode** (*str*) – The mode of decomposition. Currently 'reduced', 'complete', 'r', and 'raw' modes are supported. The default mode is 'reduced', in which matrix `A = (M, N)` is decomposed into `Q`, `R` with dimensions `(M, K)`, `(K, N)`, where `K = min(M, N)`.

**Returns** Although the type of returned object depends on the mode, it returns a tuple of `(Q, R)` by default. For details, please see the document of `numpy.linalg.qr()`.

**Return type** *cupy.ndarray*, or tuple of ndarray

**Warning:** This function calls one or more cuSOLVER routine(s) which may yield invalid results if input conditions are not met. To detect these invalid results, you can set the *linalg* configuration to a value that is not *ignore* in `cupyx.errstate()` or `cupyx.seterr()`.

**See also:**

`numpy.linalg.qr()`

### cupy.linalg.svd

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

Singular Value Decomposition.

Factorizes the matrix `a` as `u * np.diag(s) * v`, where `u` and `v` are unitary and `s` is an one-dimensional array of `a`'s singular values.

**Parameters**

- **a** (cupy.ndarray) – The input matrix with dimension `(M, N)`.
- **full_matrices** (*bool*) – If True, it returns u and v with dimensions `(M, M)` and `(N, N)`. Otherwise, the dimensions of u and v are 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 * np.diag(s) * v`.

**Return type** tuple of *cupy.ndarray*

**Warning:** This function calls one or more cuSOLVER routine(s) which may yield invalid results if input conditions are not met. To detect these invalid results, you can set the *linalg* configuration to a value that is not *ignore* in `cupyx.errstate()` or `cupyx.seterr()`.

**See also:**

`numpy.linalg.svd()`

### Matrix eigenvalues

| | |
|---|---|
| *cupy.linalg.eigh* | Eigenvalues and eigenvectors of a symmetric matrix. |
| *cupy.linalg.eigvalsh* | Calculates eigenvalues of a symmetric matrix. |

### cupy.linalg.eigh

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

Eigenvalues and eigenvectors of a symmetric matrix.

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

**Note:** Currently only 2-D matrix is supported.

**Parameters**

- **a** (cupy.ndarray) – A symmetric 2-D square matrix.
- **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].

**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 symmetric matrix.

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

**Note:** Currenlty only 2-D matrix is supported.

**Parameters**

- **a** (cupy.ndarray) – A symmetric 2-D square matrix.
- **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.

**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.

| | |
|---|---|
| *cupy.linalg.det* | Returns the determinant of an array. |
| *cupy.linalg.norm* | Returns one of matrix norms specified by `ord` parameter. |
| *cupy.linalg.matrix_rank* | Return matrix rank of array using SVD method |
| *cupy.linalg.slogdet* | Returns sign and logarithm of the determinant of an array. |
| *cupy.trace* | Returns the sum along the diagonals of an array. |

### 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

### cupy.linalg.matrix_rank

cupy.linalg.**matrix_rank**(*M*, *tol=None*)
Return matrix rank of array using SVD method

**Parameters**

- **M** (cupy.ndarray) – Input array. Its *ndim* must be less than or equal to 2.
- **tol** (*None or float*) – Threshold of singular value of *M*. When *tol* is *None*, and *eps* is the epsilon value for datatype of *M*, then *tol* is set to *S.max() * max(M.shape) * eps*, where *S* is the singular value of *M*. It obeys numpy.linalg.matrix_rank().

**Returns** Rank of *M*.

**Return type** *cupy.ndarray*

**See also:**

numpy.linalg.matrix_rank()

### cupy.linalg.slogdet

cupy.linalg.**slogdet**(*a*)
Returns sign and logarithm of the determinant of an array.

It calculates the natural logarithm of the determinant of a given value.

**Parameters** **a** (cupy.ndarray) – The input matrix with dimension (..., N, N).

**Returns** It returns a tuple (sign, logdet). sign represents each sign of the determinant as a real number 0, 1 or -1. 'logdet' represents the natural logarithm of the absolute of the determinant. If the determinant is zero, sign will be 0 and logdet will be -inf. The shapes of both sign and logdet are equal to a.shape[:-2].

**Return type** tuple of *ndarray*

**Warning:** This function calls one or more cuSOLVER routine(s) which may yield invalid results if input conditions are not met. To detect these invalid results, you can set the *linalg* configuration to a value that is not *ignore* in cupyx.errstate() or cupyx.seterr().

**Warning:** To produce the same results as numpy.linalg.slogdet() for singular inputs, set the *linalg* configuration to *raise*.

**See also:**

numpy.linalg.slogdet()

### cupy.trace

cupy.**trace**(*a*, *offset=0*, *axis1=0*, *axis2=1*, *dtype=None*, *out=None*)
Returns the sum along the diagonals of an array.

It computes the sum along the diagonals at axis1 and axis2.

**Parameters**

- **a** (cupy.ndarray) – Array to take trace.
- **offset** (*int*) – Index of diagonals. Zero indicates the main diagonal, a positive value an upper diagonal, and a negative value a lower diagonal.
- **axis1** (*int*) – The first axis along which the trace is taken.
- **axis2** (*int*) – The second axis along which the trace is taken.
- **dtype** – Data type specifier of the output.
- **out** (cupy.ndarray) – Output array.

**Returns** The trace of `a` along axes `(axis1, axis2)`.

**Return type** *cupy.ndarray*

**See also:**

`numpy.trace()`

### Solving linear equations

| | |
|---|---|
| *cupy.linalg.solve* | Solves a linear matrix equation. |
| *cupy.linalg.tensorsolve* | Solves tensor equations denoted by `ax = b`. |
| *cupy.linalg.lstsq* | Return the least-squares solution to a linear matrix equation. |
| *cupy.linalg.inv* | Computes the inverse of a matrix. |
| *cupy.linalg.pinv* | Compute the Moore-Penrose pseudoinverse of a matrix. |
| *cupy.linalg.tensorinv* | Computes the inverse of a tensor. |
| *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` |
| *cupyx.scipy.linalg.solve_triangular* | Solve the equation a x = b for x, assuming a is a triangular matrix. |

### cupy.linalg.solve

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()`.

**See also:**

`numpy.linalg.solve()`

### cupy.linalg.tensorsolve

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

Solves tensor equations denoted by `ax = b`.

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

**Parameters**

- **a** (`cupy.ndarray`) – The tensor with `len(shape) >= 1`
- **b** (`cupy.ndarray`) – The tensor with `len(shape) >= 1`
- **axes** (*`tuple of ints`*) – Axes in `a` to reorder to the right before inversion.

**Returns** The tensor with shape `Q` such that `b.shape + Q == a.shape`.

**Return type** *cupy.ndarray*

**Warning:** This function calls one or more cuSOLVER routine(s) which may yield invalid results if input conditions are not met. To detect these invalid results, you can set the *linalg* configuration to a value that is not *ignore* in `cupyx.errstate()` or `cupyx.seterr()`.

**See also:**

`numpy.linalg.tensorsolve()`

### cupy.linalg.lstsq

`cupy.linalg.`**`lstsq`**(*a*, *b*, *rcond=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, residuals, rank, s)`. Note `x` is the least-squares solution with shape `(N,)` or `(N, K)` depending if `b` was two-dimensional. The sums of `residuals` is the squared Euclidean 2-norm for each column in b - a*x. The `residuals` is an empty array if the rank of a is < N or M <= N, but iff b is 1-dimensional, this is a (1,) shape array, Otherwise the shape is (K,). The `rank` of matrix `a` is an integer. The singular values of `a` are `s`.

**Return type** tuple

**Warning:** This function calls one or more cuSOLVER routine(s) which may yield invalid results if input conditions are not met. To detect these invalid results, you can set the *linalg* configuration to a value that is not *ignore* in `cupyx.errstate()` or `cupyx.seterr()`.

**See also:**

`numpy.linalg.lstsq()`

### cupy.linalg.inv

`cupy.linalg.`**`inv`**(*a*)

Computes the inverse of a matrix.

This function computes matrix `a_inv` from n-dimensional regular matrix `a` such that `dot(a, a_inv) == eye(n)`.

**Parameters** **a** (`cupy.ndarray`) – The regular matrix

**Returns** The inverse of a matrix.

**Return type** *cupy.ndarray*

**Warning:** This function calls one or more cuSOLVER routine(s) which may yield invalid results if input conditions are not met. To detect these invalid results, you can set the *linalg* configuration to a value that is not *ignore* in `cupyx.errstate()` or `cupyx.seterr()`.

**See also:**

`numpy.linalg.inv()`

### cupy.linalg.pinv

`cupy.linalg.`**`pinv`**(*a*, *rcond=1e-15*)

Compute the Moore-Penrose pseudoinverse of a matrix.

It computes a pseudoinverse of a matrix `a`, which is a generalization of the inverse matrix with Singular Value Decomposition (SVD). Note that it automatically removes small singular values for stability.

**Parameters**

- **a** (`cupy.ndarray`) – The matrix with dimension `(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_inv` from tensor `a` such that `tensordot(a_inv, a, ind) == I`, where `I` denotes the identity tensor.

**Parameters**

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

**Returns** The inverse of a tensor whose shape is equivalent to `a.shape[ind:] + a.shape[:ind]`.

**Return type** *cupy.ndarray*

**Warning:** This function calls one or more cuSOLVER routine(s) which may yield invalid results if input conditions are not met. To detect these invalid results, you can set the *linalg* configuration to a value that is not *ignore* in `cupyx.errstate()` or `cupyx.seterr()`.

**See also:**

`numpy.linalg.tensorinv()`

### cupyx.scipy.linalg.lu_factor

`cupyx.scipy.linalg.`**`lu_factor`**(*a*, *overwrite_a=False*, *check_finite=True*)

LU decomposition.

Decompose a given two-dimensional square matrix into `P * L * U`, where `P` is a permutation matrix, `L` lower-triangular with unit diagonal elements, and `U` upper-triangular matrix. 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 <= 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`.

```
>>> 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))
```

```
>>> 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 * x = b`, given the LU factorization of `a`

**Parameters**

- **`lu_and_piv`** (*`tuple`*) – LU factorization of matrix `a` (`(M, M)`) together with pivot indices.
- **`b`** (`cupy.ndarray`) – The matrix with dimension `(M,)` or `(M, N)`.
- **`trans`** (*`{0, 1, 2}`*) – Type of system to solve:

| trans | system |
|---|---|
| 0 | a x = b |
| 1 | a^T x = b |
| 2 | a^H x = b |

- **`overwrite_b`** (*`bool`*) – Allow overwriting data in b (may enhance performance)
- **`check_finite`** (*`bool`*) – Whether to check that the input matrices contain only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs.

**Returns** The matrix with dimension `(M,)` or `(M, N)`.

**Return type** *cupy.ndarray*

**See also:**

`scipy.linalg.lu_solve()`

#### 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', 'C'}*) – Type of system to solve: ======== ========= trans system ======== ========= 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()

### 3.3.9 Logic Functions

#### Truth value testing

| | |
|---|---|
| *cupy.all* | Tests whether all array elements along a given axis evaluate to True. |
| *cupy.any* | Tests whether any array elements along a given axis evaluate to True. |
| *cupy.in1d* | Tests whether each element of a 1-D array is also present in a second array. |
| *cupy.isin* | Calculates element in `test_elements`, broadcasting over `element` only. |

#### cupy.all

cupy.**all**(*a*, *axis=None*, *out=None*, *keepdims=False*)

Tests whether all array elements along a given axis evaluate to True.

**Parameters**

- **a** (cupy.ndarray) – Input array.
- **axis** (*int or tuple of ints*) – Along which axis to compute all. The flattened

array is used by default.

- **out** (cupy.ndarray) – Output array.
- **keepdims** (*bool*) – If True, the axis is remained as an axis of size one.

**Returns** An array reduced of the input array along the axis.

**Return type** *cupy.ndarray*

**See also:**

numpy.all()

### cupy.any

cupy.**any**(*a*, *axis=None*, *out=None*, *keepdims=False*)

Tests whether any array elements along a given axis evaluate to True.

**Parameters**

- **a** (cupy.ndarray) – Input array.
- **axis** (*int or tuple of ints*) – Along which axis to compute all. The flattened array is used by default.
- **out** (cupy.ndarray) – Output array.
- **keepdims** (*bool*) – If True, the axis is remained as an axis of size one.

**Returns** An array reduced of the input array along the axis.

**Return type** *cupy.ndarray*

**See also:**

numpy.any()

### cupy.in1d

cupy.**in1d**(*ar1*, *ar2*, *assume_unique=False*, *invert=False*)

Tests whether each element of a 1-D array is also present in a second array.

Returns a boolean array the same length as ar1 that is True where an element of ar1 is in ar2 and False otherwise.

**Parameters**

- **ar1** (cupy.ndarray) – Input array.
- **ar2** (cupy.ndarray) – The values against which to test each value of ar1.
- **assume_unique** (*bool, optional*) – Ignored
- **invert** (*bool, optional*) – If True, the values in the returned array are inverted (that is, False where an element of ar1 is in ar2 and True otherwise). Default is False.

**Returns** The values ar1[in1d] are in ar2.

**Return type** *cupy.ndarray*, bool

### cupy.isin

cupy.**isin**(*element*, *test_elements*, *assume_unique=False*, *invert=False*)

Calculates element in `test_elements`, broadcasting over `element` only. Returns a boolean array of the same shape as `element` that is `True` where an element of `element` is in `test_elements` and `False` otherwise.

**Parameters**

- **element** (cupy.ndarray) – Input array.
- **test_elements** (cupy.ndarray) – The values against which to test each value of `element`. This argument is flattened if it is an array or array_like.
- **assume_unique** (*bool, optional*) – Ignored
- **invert** (*bool, optional*) – If `True`, the values in the returned array are inverted, as if calculating element not in `test_elements`. Default is `False`.

**Returns** Has the same shape as `element`. The values `element[isin]` are in `test_elements`.

**Return type** *cupy.ndarray*, bool

### Infinities and NaNs

| | |
|---|---|
| *cupy.isfinite* | Tests finiteness elementwise. |
| *cupy.isinf* | Tests if each element is the positive or negative infinity. |
| *cupy.isnan* | Tests if each element is a NaN. |

### Array type testing

| | |
|---|---|
| *cupy.iscomplex* | Returns a bool array, where True if input element is complex. |
| *cupy.iscomplexobj* | Check for a complex type or an array of complex numbers. |
| *cupy.isfortran* | Returns True if the array is Fortran contiguous but *not* C contiguous. |
| *cupy.isreal* | Returns a bool array, where True if input element is real. |
| *cupy.isrealobj* | Return True if x is a not complex type or an array of complex numbers. |
| *cupy.isscalar* | Returns True if the type of num is a scalar type. |

### cupy.iscomplex

cupy.**iscomplex**(*x*)

Returns a bool array, where True if input element is complex.

What is tested is whether the input has a non-zero imaginary part, not if the input type is complex.

**Parameters** **x** (cupy.ndarray) – Input array.

**Returns** Boolean array of the same shape as `x`.

**Return type** *cupy.ndarray*

**See also:**

*isreal()*, *iscomplexobj()*

### Examples

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

## cupy.iscomplexobj

cupy.**iscomplexobj**(*x*)

Check for a complex type or an array of complex numbers.

The type of the input is checked, not the value. Even if the input has an imaginary part equal to zero, *iscomplexobj* evaluates to True.

**Parameters** **x** (cupy.ndarray) – Input array.

**Returns** The return value, True if x is of a complex type or has at least one complex element.

**Return type** bool

**See also:**

*isrealobj()*, *iscomplex()*

### Examples

```
>>> cupy.iscomplexobj(cupy.array([3, 1+0j, True]))
True
>>> cupy.iscomplexobj(cupy.array([3, 1, True]))
False
```

## cupy.isfortran

cupy.**isfortran**(*a*)

Returns True if the array is Fortran contiguous but *not* C contiguous.

If you only want to check if an array is Fortran contiguous use a.flags.f_contiguous instead.

**Parameters** **a** (cupy.ndarray) – Input array.

**Returns** The return value, True if a is Fortran contiguous but not C contiguous.

**Return type** bool

**See also:**

isfortran()

### Examples

cupy.array allows to specify whether the array is written in C-contiguous order (last index varies the fastest), or FORTRAN-contiguous order in memory (first index varies the fastest).

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

```
>>> b = cupy.array([[1, 2, 3], [4, 5, 6]], order='F')
>>> b
array([[1, 2, 3],
       [4, 5, 6]])
>>> cupy.isfortran(b)
True
```

The transpose of a C-ordered array is a FORTRAN-ordered array.

```
>>> a = cupy.array([[1, 2, 3], [4, 5, 6]], order='C')
>>> a
array([[1, 2, 3],
       [4, 5, 6]])
>>> cupy.isfortran(a)
False
>>> b = a.T
>>> b
array([[1, 4],
       [2, 5],
       [3, 6]])
>>> cupy.isfortran(b)
True
```

C-ordered arrays evaluate as False even if they are also FORTRAN-ordered.

```
>>> cupy.isfortran(np.array([1, 2], order='F'))
False
```

## cupy.isreal

cupy.**isreal**(*x*)

Returns a bool array, where True if input element is real.

If element has complex type with zero complex part, the return value for that element is True.

**Parameters** **x** (cupy.ndarray) – Input array.

**Returns** Boolean array of same shape as x.

**Return type** *cupy.ndarray*

**See also:**

*iscomplex()*, *isrealobj()*

#### Examples

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

### cupy.isrealobj

cupy.**isrealobj**(*x*)

Return True if x is a not complex type or an array of complex numbers.

The type of the input is checked, not the value. So even if the input has an imaginary part equal to zero, *isrealobj* evaluates to False if the data type is complex.

**Parameters** **x** (cupy.ndarray) – The input can be of any type and shape.

**Returns** The return value, False if x is of a complex type.

**Return type** bool

**See also:**

*iscomplexobj()*, *isreal()*

#### Examples

```
>>> cupy.isrealobj(cupy.array([3, 1+0j, True]))
False
>>> cupy.isrealobj(cupy.array([3, 1, True]))
True
```

### cupy.isscalar

cupy.**isscalar**(*num*)

Returns True if the type of num is a scalar type.

**See also:**

numpy.isscalar()

### Logic operations

| | |
|---|---|
| *cupy.logical_and* | Computes the logical AND of two arrays. |
| *cupy.logical_or* | Computes the logical OR of two arrays. |
| *cupy.logical_not* | Computes the logical NOT of an array. |
| *cupy.logical_xor* | Computes the logical XOR of two arrays. |

### Comparison

| | |
|---|---|
| *cupy.allclose* | Returns True if two arrays are element-wise equal within a tolerance. |

Continued on next page

Table 45 – continued from previous page

| | |
|---|---|
| *cupy.isclose* | Returns a boolean array where two arrays are equal within a tolerance. |
| *cupy.greater* | Tests elementwise if `x1 > x2`. |
| *cupy.greater_equal* | Tests elementwise if `x1 >= x2`. |
| *cupy.less* | Tests elementwise if `x1 < x2`. |
| *cupy.less_equal* | Tests elementwise if `x1 <= x2`. |
| *cupy.equal* | Tests elementwise if `x1 == x2`. |
| *cupy.not_equal* | Tests elementwise if `x1 != x2`. |

### cupy.allclose

cupy.**allclose**(*a*, *b*, *rtol=1e-05*, *atol=1e-08*, *equal_nan=False*)

Returns True if two arrays are element-wise equal within a tolerance.

Two values in `a` and `b` are considiered 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**

**if `True`, two arrays are element-wise equal within a** tolerance.

**Return type** bool

**See also:**

`numpy.allclose()`

### cupy.isclose

cupy.**isclose**(*a*, *b*, *rtol=1e-05*, *atol=1e-08*, *equal_nan=False*)

Returns a boolean array where two arrays are equal within a tolerance.

Two values in `a` and `b` are considiered 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()`

### 3.3.10 Mathematical Functions

#### Trigonometric functions

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

#### 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

| | |
|---|---|
| *cupy.sinh* | Elementwise hyperbolic sine function. |
| *cupy.cosh* | Elementwise hyperbolic cosine function. |
| *cupy.tanh* | Elementwise hyperbolic tangent function. |
| *cupy.arcsinh* | Elementwise inverse of hyperbolic sine function. |
| *cupy.arccosh* | Elementwise inverse of hyperbolic cosine function. |
| *cupy.arctanh* | Elementwise inverse of hyperbolic tangent function. |

### Rounding

| | |
|---|---|
| *cupy.around* | Rounds to the given number of decimals. |
| *cupy.round_* | |
| *cupy.rint* | Rounds each element of an array to the nearest integer. |
| *cupy.fix* | If given value x is positive, it return floor(x). |
| *cupy.floor* | Rounds each element of an array to its floor integer. |
| *cupy.ceil* | Rounds each element of an array to its ceiling integer. |
| *cupy.trunc* | Rounds each element of an array towards zero. |

### cupy.around

cupy.**around**(*a*, *decimals=0*, *out=None*)

Rounds to the given number of decimals.

**Parameters**

- **a** (cupy.ndarray) – The source array.
- **decimals** (*int*) – umber of decimal places to round to (default: 0). If decimals is negative, it specifies the number of positions to the left of the decimal point.
- **out** (cupy.ndarray) – Output array.

**Returns** Rounded array.

**Return type** *cupy.ndarray*

**See also:**

numpy.around()

### cupy.round_

cupy.**round_**(*a*, *decimals=0*, *out=None*)

### cupy.fix

cupy.**fix = <ufunc 'cupy_fix'>**
If given value x is positive, it return floor(x). Else, it return ceil(x).

**See also:**

numpy.fix()

### Sums, products, differences

| | |
|---|---|
| *cupy.prod* | Returns the product of an array along given axes. |
| *cupy.sum* | Returns the sum of an array along given axes. |
| *cupy.cumprod* | Returns the cumulative product of an array along a given axis. |
| *cupy.cumsum* | Returns the cumulative sum of an array along a given axis. |
| *cupy.nansum* | Returns the sum of an array along given axes treating Not a Numbers (NaNs) as zero. |
| *cupy.nanprod* | Returns the product of an array along given axes treating Not a Numbers (NaNs) as zero. |
| *cupy.diff* | Calculate the n-th discrete difference along the given axis. |

### cupy.prod

cupy.**prod**(*a*, *axis=None*, *dtype=None*, *out=None*, *keepdims=False*)
Returns the product of an array along given axes.

**Parameters**

- **a** (cupy.ndarray) – Array to take product.
- **axis** (*int or sequence of ints*) – Axes along which the product is taken.
- **dtype** – Data type specifier.
- **out** (cupy.ndarray) – Output array.
- **keepdims** (*bool*) – If True, the specified axes are remained as axes of length one.

**Returns** The result array.

**Return type** *cupy.ndarray*

**See also:**

numpy.prod()

### cupy.sum

cupy.**sum**(*a*, *axis=None*, *dtype=None*, *out=None*, *keepdims=False*)
Returns the sum of an array along given axes.

**Parameters**

- **a** (cupy.ndarray) – Array to take sum.

- **axis** (*int or sequence of ints*) – Axes along which the sum is taken.
- **dtype** – Data type specifier.
- **out** (cupy.ndarray) – Output array.
- **keepdims** (*bool*) – If True, the specified axes are remained as axes of length one.

**Returns** The result array.

**Return type** *cupy.ndarray*

**See also:**

numpy.sum()

### cupy.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.

- **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

| | |
|---|---|
| *cupy.exp* | Elementwise exponential function. |
| *cupy.expm1* | Computes `exp(x) - 1` elementwise. |
| *cupy.exp2* | Elementwise exponentiation with base 2. |
| *cupy.log* | Elementwise natural logarithm function. |
| *cupy.log10* | Elementwise common logarithm function. |
| *cupy.log2* | Elementwise binary logarithm function. |
| *cupy.log1p* | Computes `log(1 + x)` elementwise. |
| *cupy.logaddexp* | Computes `log(exp(x1) + exp(x2))` elementwise. |
| *cupy.logaddexp2* | Computes `log2(exp2(x1) + exp2(x2))` elementwise. |

### Other special functions

| | |
|---|---|
| *cupy.i0* | Modified Bessel function of the first kind, order 0. |
| *cupy.sinc* | Elementwise sinc function. |

#### cupy.i0

cupy.**i0 = <ufunc 'cupy_i0'>**
Modified Bessel function of the first kind, order 0.

**See also:**

`numpy.i0()`

#### cupy.sinc

cupy.**sinc = <ufunc 'cupy_sinc'>**
Elementwise sinc function.

**See also:**

`numpy.sinc()`

### Floating point routines

| | |
|---|---|
| *cupy.signbit* | Tests elementwise if the sign bit is set (i.e. |

Continued on next page

Table 52 – continued from previous page

| | |
|---|---|
| *cupy.copysign* | Returns the first argument with the sign bit of the second elementwise. |
| *cupy.frexp* | Decomposes each element to mantissa and two's exponent. |
| *cupy.ldexp* | Computes `x1 * 2 ** x2` elementwise. |
| *cupy.nextafter* | Computes the nearest neighbor float values towards the second argument. |

### 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. |
| *cupy.power* | Computes `x1 ** x2` elementwise. |
| *cupy.subtract* | Subtracts arguments elementwise. |
| *cupy.true_divide* | Elementwise true division (i.e. |
| *cupy.floor_divide* | Elementwise floor division (i.e. |
| *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* | |

### 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.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_conj'>**
Returns the complex conjugate, element-wise.

**See also:**

numpy.conj

### Miscellaneous

| | |
|---|---|
| *cupy.clip* | Clips the values of an array to a given interval. |
| *cupy.sqrt* | Elementwise square root function. |
| *cupy.cbrt* | Elementwise cube root function. |
| *cupy.square* | Elementwise square function. |
| *cupy.absolute* | Elementwise absolute value function. |
| *cupy.sign* | Elementwise sign function. |
| *cupy.maximum* | Takes the maximum of two arrays elementwise. |
| *cupy.minimum* | Takes the minimum of two arrays elementwise. |
| *cupy.fmax* | Takes the maximum of two arrays elementwise. |
| *cupy.fmin* | Takes the minimum of two arrays elementwise. |
| *cupy.nan_to_num* | Elementwise nan_to_num function. |
| *cupy.blackman* | Returns the Blackman window. |
| *cupy.hamming* | Returns the Hamming window. |
| *cupy.hanning* | Returns the Hanning window. |

### cupy.clip

cupy.**clip**(*a*, *a_min=None*, *a_max=None*, *out=None*)
Clips the values of an array to a given interval.

This is equivalent to maximum(minimum(a, a_max), a_min), while this function is more efficient.

**Parameters**

- **a** (cupy.ndarray) – The source array.
- **a_min** (*scalar, cupy.ndarray or None*) – The left side of the interval. When it is None, it is ignored.
- **a_max** (*scalar, cupy.ndarray or None*) – The right side of the interval. When it is None, it is ignored.
- **out** (cupy.ndarray) – Output array.

**Returns** Clipped array.

**Return type** *cupy.ndarray*

**See also:**

numpy.clip()

### cupy.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.blackman

cupy.**blackman**(*M*)
Returns the Blackman window.

The Blackman window is defined as

$$w(n) = 0.42 - 0.5\cos\left(\frac{2\pi n}{M-1}\right) + 0.08\cos\left(\frac{4\pi n}{M-1}\right) \qquad 0 \le n \le M-1$$

**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

$$w(n) = 0.54 - 0.46\cos\left(\frac{2\pi n}{M-1}\right) \qquad 0 \le n \le M-1$$

**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

$$w(n) = 0.5 - 0.5\cos\left(\frac{2\pi n}{M-1}\right) \qquad 0 \le n \le M-1$$

**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()

## 3.3.11 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

```
>>> a = cupy.array([1, 2, 3, 4, 5])
>>> cupy.pad(a, (2, 3), 'constant', constant_values=(4, 6))
array([4, 4, 1, ..., 6, 6, 6])
```

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

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

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

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

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

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

```
>>> cupy.pad(a, (2, 3), 'reflect', reflect_type='odd')
array([-1,  0,  1,  2,  3,  4,  5,  6,  7,  8])
```

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

```
>>> cupy.pad(a, (2, 3), 'symmetric', reflect_type='odd')
array([0, 1, 1, 2, 3, 4, 5, 5, 6, 7])
```

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

```
>>> def pad_with(vector, pad_width, iaxis, kwargs):
...     pad_value = kwargs.get('padder', 10)
...     vector[:pad_width[0]] = pad_value
...     vector[-pad_width[1]:] = pad_value
>>> a = cupy.arange(6)
>>> a = a.reshape((2, 3))
>>> cupy.pad(a, 2, pad_with)
array([[10, 10, 10, 10, 10, 10, 10],
       [10, 10, 10, 10, 10, 10, 10],
       [10, 10,  0,  1,  2, 10, 10],
       [10, 10,  3,  4,  5, 10, 10],
       [10, 10, 10, 10, 10, 10, 10],
       [10, 10, 10, 10, 10, 10, 10]])
>>> cupy.pad(a, 2, pad_with, padder=100)
array([[100, 100, 100, 100, 100, 100, 100],
       [100, 100, 100, 100, 100, 100, 100],
       [100, 100,   0,   1,   2, 100, 100],
       [100, 100,   3,   4,   5, 100, 100],
       [100, 100, 100, 100, 100, 100, 100],
       [100, 100, 100, 100, 100, 100, 100]])
```

### 3.3.12 Random Sampling (`cupy.random`)

Differences between *`cupy.random`* and `numpy.random`:

- CuPy provides Legacy Random Generation API (see also: NumPy 1.16 Reference). The new random generator API (`numpy.random.Generator` class) introduced in NumPy 1.17 has not been implemented yet.
- Most functions under *`cupy.random`* support the `dtype` option, which do not exist in the corresponding NumPy APIs. This option enables generation of float32 values directly without any space overhead.
- CuPy does not guarantee that the same number generator is used across major versions. This means that numbers generated by *`cupy.random`* by new major version may not be the same as the previous one, even if the same seed and distribution are used.

#### Simple random data

| | |
|---|---|
| *`cupy.random.rand`* | Returns an array of uniform random values over the interval `[0, 1)`. |
| *`cupy.random.randn`* | Returns an array of standard normal random values. |

Continued on next page

Table 57 – continued from previous page

| | |
|---|---|
| *cupy.random.randint* | Returns a scalar or an array of integer values over `[low, high)`. |
| *cupy.random.random_integers* | Return a scalar or an array of integer values over `[low, high]` |
| *cupy.random.random_sample* | Returns an array of random values over the interval `[0, 1)`. |
| *cupy.random.random* | Returns an array of random values over the interval `[0, 1)`. |
| *cupy.random.ranf* | Returns an array of random values over the interval `[0, 1)`. |
| *cupy.random.sample* | Returns an array of random values over the interval `[0, 1)`. |
| *cupy.random.choice* | Returns an array of random values from a given 1-D array. |
| *cupy.random.bytes* | Returns random bytes. |

### cupy.random.rand

cupy.random.**rand**(**size*, ***kwarg*)

Returns an array of uniform random values over the interval `[0, 1)`.

Each element of the array is uniformly distributed on the half-open interval `[0, 1)`. All elements are identically and independently distributed (i.i.d.).

**Parameters**

- **size** (*ints*) – The shape of the array.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed. The default is `numpy.float64`.

**Returns** A random array.

**Return type** *cupy.ndarray*

**See also:**

`numpy.random.rand`

**Example**

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

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

### cupy.random.randn

cupy.random.**randn**(**size*, ***kwarg*)

Returns an array of standard normal random values.

Each element of the array is normally distributed with zero mean and unit variance. All elements are identically and independently distributed (i.i.d.).

**Parameters**

- **size** (*ints*) – The shape of the array.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed. The default is `numpy.float64`.

**Returns** An array of standard normal random values.

**Return type** *cupy.ndarray*

**See also:**

`numpy.random.randn`

**Example**

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

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

### cupy.random.randint

cupy.random.**randint**(*low*, *high=None*, *size=None*, *dtype='l'*)

Returns a scalar or an array of integer values over `[low, high)`.

Each element of returned values are independently sampled from uniform distribution over left-close and right-open interval `[low, high)`.

**Parameters**

- **low** (*int*) – If `high` is not `None`, it is the lower bound of the interval. Otherwise, it is the **upper** bound of the interval and lower bound of the interval is set to `0`.
- **high** (*int*) – Upper bound of the interval.
- **size** (*None or int or tuple of ints*) – The shape of returned value.
- **dtype** – Data type specifier.

**Returns** If size is `None`, it is single integer sampled. If size is integer, it is the 1D-array of length `size` element. Otherwise, it is the array whose shape specified by `size`.

**Return type** int or cupy.ndarray of ints

### cupy.random.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*

**See also:**

`numpy.random.random_sample`

### 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* | (experimental) Multivariate normal distribution. |
| *cupy.random.negative_binomial* | Negative binomial distribution. |
| *cupy.random.noncentral_chisquare* | Noncentral chisquare 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.random.standard_t* | Standard Student's t distribution. |
| *cupy.random.triangular* | Triangular distribution. |
| *cupy.random.uniform* | Returns an array of uniformly-distributed samples over an interval. |
| *cupy.random.vonmises* | von Mises distribution. |
| *cupy.random.wald* | Wald distribution. |
| *cupy.random.weibull* | weibull distribution. |
| *cupy.random.zipf* | Zipf distribution. |

### cupy.random.beta

cupy.random.**beta**(*a*, *b*, *size=None*, *dtype=<class 'float'>*)

Beta distribution.

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

$$f(x) = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha,\beta)}.$$

**Parameters**

- **a** (*float*) – Parameter of the beta distribution $\alpha$.
- **b** (*float*) – Parameter of the beta distribution $\beta$.
- **size** (*int or tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

**Returns** Samples drawn from the beta distribution.

**Return type** *cupy.ndarray*

**See also:**

`numpy.random.beta`

### cupy.random.binomial

`cupy.random.`**binomial**(*n*, *p*, *size=None*, *dtype=<class 'int'>*)

Binomial distribution.

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

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

**Parameters**

- **n** (*int*) – Trial number of the binomial distribution.
- **p** (*float*) – Success probability of the binomial distribution.
- **size** (*int or tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.int32` and `numpy.int64` types are allowed.

**Returns** Samples drawn from the binomial distribution.

**Return type** *cupy.ndarray*

**See also:**

`numpy.random.binomial`

### cupy.random.chisquare

`cupy.random.`**chisquare**(*df*, *size=None*, *dtype=<class 'float'>*)

Chi-square distribution.

Returns an array of samples drawn from the chi-square distribution. Its probability density function is defined as

$$f(x) = \frac{(1/2)^{k/2}}{\Gamma(k/2)} x^{k/2-1} e^{-x/2}.$$

**Parameters**

- **df** (*int or array_like of ints*) – Degree of freedom $k$.
- **size** (*int or tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

**Returns** Samples drawn from the chi-square distribution.

**Return type** *cupy.ndarray*

**See also:**

`numpy.random.chisquare`

### cupy.random.dirichlet

`cupy.random.`**dirichlet**(*alpha*, *size=None*, *dtype=<class 'float'>*)

Dirichlet distribution.

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

$$f(x) = \frac{\Gamma(\sum_{i=1}^{K} \alpha_i)}{\prod_{i=1}^{K} \Gamma(\alpha_i)} \prod_{i=1}^{K} x_i^{\alpha_i - 1}.$$

**Parameters**

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

**Returns** Samples drawn from the dirichlet distribution.

**Return type** *cupy.ndarray*

**See also:**

`numpy.random.dirichlet`

### cupy.random.exponential

`cupy.random.`**exponential**(*scale*, *size=None*, *dtype=<class 'float'>*)

Exponential distribution.

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

$$f(x) = \frac{1}{\beta} \exp(-\frac{x}{\beta}).$$

**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

$$f(x) = \frac{1}{B(\frac{d_1}{2}, \frac{d_2}{2})} \left(\frac{d_1}{d_2}\right)^{\frac{d_1}{2}} x^{\frac{d_1}{2}-1} \left(1 + \frac{d_1}{d_2}x\right)^{-\frac{d_1+d_2}{2}}.$$

**Parameters**

- **dfnum** (*float or array_like of floats*) – Parameter of the f distribution $d_1$.
- **dfden** (*float or array_like of floats*) – Parameter of the f distribution $d_2$.
- **size** (*int or tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

**Returns** Samples drawn from the f distribution.

**Return type** *cupy.ndarray*

**See also:**

`numpy.random.f`

### cupy.random.gamma

`cupy.random.`**gamma**(*shape*, *scale=1.0*, *size=None*, *dtype=<class 'float'>*)

Gamma distribution.

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

$$f(x) = \frac{1}{\Gamma(k)\theta^k} x^{k-1} e^{-x/\theta}.$$

**Parameters**

- **shape** (*array*) – Parameter of the gamma distribution $k$.
- **scale** (*array*) – Parameter of the gamma distribution $\theta$
- **size** (*int or tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.

- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

Returns:cupy.ndarray: Samples drawn from the gamma distribution.

**See also:**

`numpy.random.gamma`

### cupy.random.geometric

`cupy.random.`**geometric**(*p*, *size=None*, *dtype=<class 'int'>*)

Geometric distribution.

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

$$f(x) = p(1-p)^{k-1}.$$

**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 `loc` and scale `scale`. Its probability density function is defined as

$$f(x) = \frac{1}{\eta} \exp\left\{-\frac{x-\mu}{\eta}\right\} \exp\left[-\exp\left\{-\frac{x-\mu}{\eta}\right\}\right],$$

where $\mu$ is `loc` and $\eta$ is `scale`.

**Parameters**

- **loc** (*float*) – The location of the mode $\mu$.
- **scale** (*float*) – The scale parameter $\eta$.
- **size** (*int or tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

**Returns** Samples drawn from the Gumbel distribution.

**Return type** *cupy.ndarray*

**See also:**

`numpy.random.gumbel`

### cupy.random.hypergeometric

`cupy.random.`**`hypergeometric`**(*ngood*, *nbad*, *nsample*, *size=None*, *dtype=<class 'int'>*)
hypergeometric distribution.

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

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

**Parameters**

- **ngood** (*int or array_like of ints*) – Parameter of the hypergeometric distribution $n$.
- **nbad** (*int or array_like of ints*) – Parameter of the hypergeometric distribution $m$.
- **nsample** (*int or array_like of ints*) – Parameter of the hypergeometric distribution $N$.
- **size** (*int or tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.int32` and `numpy.int64` types are allowed.

**Returns** Samples drawn from the hypergeometric distribution.

**Return type** *cupy.ndarray*

**See also:**

`numpy.random.hypergeometric`

### cupy.random.laplace

`cupy.random.`**`laplace`**(*loc=0.0*, *scale=1.0*, *size=None*, *dtype=<class 'float'>*)
Laplace distribution.

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

$$f(x) = \frac{1}{2b}\exp\left(-\frac{|x-\mu|}{b}\right).$$

**Parameters**

- **loc** (*float*) – The location of the mode $\mu$.
- **scale** (*float*) – The scale parameter $b$.
- **size** (*int or tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

**Returns** Samples drawn from the laplace distribution.

**Return type** *cupy.ndarray*

**See also:**

`numpy.random.laplace`

### cupy.random.logistic

`cupy.random.`**`logistic`**(*loc=0.0*, *scale=1.0*, *size=None*, *dtype=<class 'float'>*)

Logistic distribution.

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

$$f(x) = \frac{e^{-(x-\mu)/s}}{s(1+e^{-(x-\mu)/s})^2}.$$

**Parameters**

- **loc** (*float*) – The location of the mode $\mu$.
- **scale** (*float*) – The scale parameter $s$.
- **size** (*int or tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

**Returns** Samples drawn from the logistic distribution.

**Return type** *cupy.ndarray*

**See also:**

`numpy.random.logistic`

### cupy.random.lognormal

`cupy.random.`**`lognormal`**(*mean=0.0*, *sigma=1.0*, *size=None*, *dtype=<class 'float'>*)

Returns an array of samples drawn from a log normal distribution.

The samples are natural log of samples drawn from a normal distribution with mean `mean` and deviation `sigma`.

**Parameters**

- **mean** (*float*) – Mean of the normal distribution.
- **sigma** (*float*) – Standard deviation of the normal distribution.
- **size** (*int or tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

**Returns** Samples drawn from the log normal distribution.

**Return type** *cupy.ndarray*

**See also:**

`numpy.random.lognormal`

### cupy.random.logseries

cupy.random.**logseries**(*p*, *size=None*, *dtype=<class 'int'>*)

Log series distribution.

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

$$f(x) = \frac{-p^x}{x \ln(1-p)}.$$

**Parameters**

- **p** (*float*) – Parameter of the log series distribution $p$.
- **size** (*int or tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.int32` and `numpy.int64` types are allowed.

**Returns** Samples drawn from the log series distribution.

**Return type** *cupy.ndarray*

**See also:**

`numpy.random.logseries`

### cupy.random.multinomial

cupy.random.**multinomial**(*n*, *pvals*, *size=None*)

Returns an array from multinomial distribution.

**Parameters**

- **n** (*int*) – Number of trials.
- **pvals** (`cupy.ndarray`) – Probabilities of each of the `p` different outcomes. The sum of these values must be 1.
- **size** (*int or tuple of ints or None*) – Shape of a sample in each trial. For example when `size` is `(a, b)`, shape of returned value is `(a, b, p)` where `p` is `len(pvals)`. If `size` is `None`, it is treated as `()`. So, shape of returned value is `(p,)`.

**Returns** An array drawn from multinomial distribution.

**Return type** *cupy.ndarray*

---

**Note:** It does not support `sum(pvals) < 1` case.

---

**See also:**

`numpy.random.multinomial`

### cupy.random.multivariate_normal

cupy.random.**multivariate_normal**(*mean*, *cov*, *size=None*, *check_valid='ignore'*, *tol=1e-08*, *dtype=<class 'float'>*)

(experimental) Multivariate normal distribution.

Returns an array of samples drawn from the multivariate normal distribution. Its probability density function is defined as

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

**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.
- **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*

**See also:**

`numpy.random.multivariate_normal`

### cupy.random.negative_binomial

`cupy.random.`**negative_binomial**(*n*, *p*, *size=None*, *dtype=<class 'int'>*)

Negative binomial distribution.

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

$$f(x) = \binom{x+n-1}{n-1} p^n (1-p)^x.$$

**Parameters**

- **n** (*int*) – Parameter of the negative binomial distribution $n$.
- **p** (*float*) – Parameter of the negative binomial distribution $p$.
- **size** (*int or tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.int32` and `numpy.int64` types are allowed.

**Returns** Samples drawn from the negative binomial distribution.

**Return type** *cupy.ndarray*

**See also:**

`numpy.random.negative_binomial`

### cupy.random.noncentral_chisquare

cupy.random.**noncentral_chisquare**(*df*, *nonc*, *size=None*, *dtype=<class 'float'>*)
Noncentral chisquare distribution.

Returns an array of samples drawn from the noncentral chisquare distribution. Its probability density function is defined as

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

where $I$ is the modified Bessel function of the first kind.

**Parameters**

- **df** (*float*) – Parameter of the noncentral chisquare distribution $k$.
- **nonc** (*float*) – Parameter of the noncentral chisquare distribution $\lambda$.
- **size** (*int or tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

**Returns** Samples drawn from the noncentral chisquare distribution.

**Return type** *cupy.ndarray*

**See also:**

`numpy.random.noncentral_chisquare`

### cupy.random.noncentral_f

cupy.random.**noncentral_f**(*dfnum*, *dfden*, *nonc*, *size=None*, *dtype=<class 'float'>*)
Noncentral F distribution.

Returns an array of samples drawn from the noncentral F distribution.

Reference: https://en.wikipedia.org/wiki/Noncentral_F-distribution

**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

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

**Parameters**

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

**Returns** Samples drawn from the standard Student's t distribution.

**Return type** *cupy.ndarray*

**See also:**

`numpy.random.standard_t`

### cupy.random.triangular

cupy.random.**triangular**(*left*, *mode*, *right*, *size=None*, *dtype=<class 'float'>*)
Triangular distribution.

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

$$f(x) = \begin{cases} \frac{2(x-l)}{(r-l)(m-l)} & \text{for } l \le x \le m, \\ \frac{2(r-x)}{(r-l)(r-m)} & \text{for } m \le x \le r, \\ 0 & \text{otherwise.} \end{cases}$$

**Parameters**

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

**Returns** Samples drawn from the triangular distribution.

**Return type** *cupy.ndarray*

**See also:**

*cupy.random.RandomState.triangular()* `numpy.random.triangular`

### cupy.random.uniform

cupy.random.**uniform**(*low=0.0*, *high=1.0*, *size=None*, *dtype=<class 'float'>*)

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

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

**Parameters**

- **low** (*float*) – Lower end of the interval.
- **high** (*float*) – Upper end of the interval.
- **size** (*int or tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier.

**Returns** Samples drawn from the uniform distribution.

**Return type** *cupy.ndarray*

**See also:**

`numpy.random.uniform`

### cupy.random.vonmises

cupy.random.**vonmises**(*mu*, *kappa*, *size=None*, *dtype=<class 'float'>*)

von Mises distribution.

Returns an array of samples drawn from the von Mises distribution. Its probability density function is defined as

$$f(x) = \frac{e^{\kappa \cos(x-\mu)}}{2\pi I_0(\kappa)}.$$

**Parameters**

- **mu** (*float*) – Parameter of the von Mises distribution $\mu$.
- **kappa** (*float*) – Parameter of the von Mises distribution $\kappa$.
- **size** (*int or tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

**Returns** Samples drawn from the von Mises distribution.

**Return type** *cupy.ndarray*

**See also:**

`numpy.random.vonmises`

### cupy.random.wald

cupy.random.**wald**(*mean*, *scale*, *size=None*, *dtype=<class 'float'>*)

Wald distribution.

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

$$f(x) = \sqrt{\frac{\lambda}{2\pi x^3}}\, e^{\frac{-\lambda(x-\mu)^2}{2\mu^2 x}}.$$

**Parameters**

- **mean** (*float*) – Parameter of the wald distribution $\mu$.
- **scale** (*float*) – Parameter of the wald distribution $\lambda$.
- **size** (*int or tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

**Returns** Samples drawn from the wald distribution.

**Return type** *cupy.ndarray*

**See also:**

*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

| | |
|---|---|
| *cupy.random.RandomState* | Portable container of a pseudo-random number generator. |
| *cupy.random.seed* | Resets the state of the random number generator with a seed. |
| *cupy.random.get_random_state* | Gets the state of the random number generator for the current device. |
| *cupy.random.set_random_state* | Sets the state of the random number generator for the current device. |

### cupy.random.RandomState

**class** cupy.random.**RandomState**(*seed=None*, *method=100*)

Portable container of a pseudo-random number generator.

An instance of this class holds the state of a random number generator. The state is available only on the device which has been current at the initialization of the instance.

Functions of *cupy.random* use global instances of this class. Different instances are used for different devices. The global state for the current device can be obtained by the *cupy.random.get_random_state()* function.

**Parameters**

- **seed** (*None or int*) – Seed of the random number generator. See the *seed()* method for detail.
- **method** (*int*) – Method of the random number generator. Following values are available:

```
cupy.cuda.curand.CURAND_RNG_PSEUDO_DEFAULT
cupy.cuda.curand.CURAND_RNG_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.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*, *dtype=<class 'float'>*)
(experimental) Returns an array of samples drawn from the multivariate normal distribution.

**See also:**

*cupy.random.multivariate_normal()* for full documentation, `numpy.random.RandomState.multivariate_normal`

**negative_binomial**(*n*, *p*, *size=None*, *dtype=<class 'int'>*)
Returns an array of samples drawn from the negative binomial distribution.

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

**See also:**

*cupy.random.negative_binomial()* for full documentation, `numpy.random.RandomState.negative_binomial`

**noncentral_chisquare**(*df*, *nonc*, *size=None*, *dtype=<class 'float'>*)
Returns an array of samples drawn from the noncentral chi-square distribution.

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

**See also:**

*cupy.random.noncentral_chisquare()* for full documentation, `numpy.random.RandomState.noncentral_chisquare`

**noncentral_f**(*dfnum*, *dfden*, *nonc*, *size=None*, *dtype=<class 'float'>*)
Returns an array of samples drawn from the noncentral F distribution.

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

**See also:**

*cupy.random.noncentral_f()* for full documentation, `numpy.random.RandomState.noncentral_f`

**normal**(*loc=0.0*, *scale=1.0*, *size=None*, *dtype=<class 'float'>*)
Returns an array of normally distributed samples.

**See also:**

*cupy.random.normal()* for full documentation, `numpy.random.RandomState.normal`

**pareto**(*a*, *size=None*, *dtype=<class 'float'>*)
Returns an array of samples drawn from the pareto II distribution.

**See also:**

`cupy.random.pareto_kernel()` for full documentation, `numpy.random.RandomState.pareto`

**permutation**(*a*)
Returns a permuted range or a permutation of an array.

**poisson**(*lam=1.0*, *size=None*, *dtype=<class 'int'>*)
Returns an array of samples drawn from the poisson distribution.

**See also:**

*cupy.random.poisson()* for full documentation, `numpy.random.RandomState.poisson`

**power**(*a*, *size=None*, *dtype=<class 'float'>*)
Returns an array of samples drawn from the power distribution.

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

**See also:**

*cupy.random.power()* for full documentation, numpy.random.RandomState.power

**rand**(**size*, ***kwarg*)
Returns uniform random values over the interval [0, 1).

**See also:**

*cupy.random.rand()* for full documentation, numpy.random.RandomState.rand

**randint**(*low*, *high=None*, *size=None*, *dtype='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 document, 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.

---

### 3.3.13 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.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*)

Perform an indirect sort using an array of keys.

**Parameters** **keys** (cupy.ndarray) – (k, N) array containing k (N,)-shaped arrays. The k different "rows" to be sorted. The last row is the primary sort key.

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

**Return type** *cupy.ndarray*

---

**Note:** For its implementation reason, cupy.lexsort currently supports only keys with their rank of one or two and does not support axis parameter that numpy.lexsort supports.

---

**See also:**

numpy.lexsort()

### cupy.argsort

cupy.**argsort**(*a*, *axis=-1*)

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

**Parameters**

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

**Returns** Array of indices that sort a.

**Return type** *cupy.ndarray*

---

**Note:** For its implementation reason, cupy.argsort does not support kind and order parameters.

---

**See also:**

numpy.argsort()

### cupy.msort

cupy.**msort**(*a*)

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

**Parameters** **a** (cupy.ndarray) – Array to be sorted.

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

**Return type** *cupy.ndarray*

**See also:**

numpy.msort()

### cupy.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

| | |
|---|---|
| *cupy.argmax* | Returns the indices of the maximum along an axis. |
| *cupy.nanargmax* | Return the indices of the maximum values in the specified axis ignoring NaNs. |
| *cupy.argmin* | Returns the indices of the minimum along an axis. |
| *cupy.nanargmin* | Return the indices of the minimum values in the specified axis ignoring NaNs. |
| *cupy.nonzero* | Return the indices of the elements that are non-zero. |
| *cupy.flatnonzero* | Return indices that are non-zero in the flattened version of a. |
| *cupy.where* | Return elements, either from x or y, depending on condition. |

#### cupy.argmax

cupy.**argmax**(*a*, *axis=None*, *dtype=None*, *out=None*, *keepdims=False*)

Returns the indices of the maximum along an axis.

**Parameters**

- **a** (cupy.ndarray) – Array to take argmax.
- **axis** (*int*) – Along which axis to find the maximum. a is flattened by default.
- **dtype** – Data type specifier.
- **out** (cupy.ndarray) – Output array.
- **keepdims** (*bool*) – If True, the axis axis is preserved as an axis of length one.

**Returns** The indices of the maximum of a along an axis.

**Return type** *cupy.ndarray*

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

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

**See also:**

numpy.argmax()

#### cupy.nanargmax

cupy.**nanargmax**(*a*, *axis=None*, *dtype=None*, *out=None*, *keepdims=False*)

Return the indices of the maximum values in the specified axis ignoring NaNs. For all-NaN slice -1 is returned. Subclass cannot be passed yet, subok=True still unsupported

**Parameters**

- **a** (cupy.ndarray) – Array to take nanargmax.
- **axis** (*int*) – Along which axis to find the maximum. a is flattened by default.

**Returns**

**The indices of the maximum of `a`** along an axis ignoring NaN values.

**Return type** *cupy.ndarray*

---

**Note:** For performance reasons, `cupy.nanargmax` returns `out of range values` for all-NaN slice whereas `numpy.nanargmax` raises `ValueError`

---

**See also:**

`numpy.nanargmax()`

### cupy.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()`

## Counting

| | |
|---|---|
| *cupy.count_nonzero* | Counts the number of non-zero values in the array. |

### cupy.count_nonzero

`cupy.`**`count_nonzero`**`(`*a*, *axis=None*`)`

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

---

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

---

**Parameters**

- **a** (`cupy.ndarray`) – The array for which to count non-zeros.
- **axis** (*int or tuple, optional*) – Axis or tuple of axes along which to count non-zeros. Default is None, meaning that non-zeros will be counted along a flattened version of `a`

**Returns**

**Number of non-zero values in the array** along a given axis. Otherwise, the total number of non-zero values in the array is returned.

**Return type** cupy.ndarray of int

### 3.3.14 Statistical Functions

#### Order statistics

| | |
|---|---|
| *cupy.amin* | Returns the minimum of an array or the minimum along an axis. |
| *cupy.amax* | Returns the maximum of an array or the maximum along an axis. |
| *cupy.nanmin* | Returns the minimum of an array along an axis ignoring NaN. |
| *cupy.nanmax* | Returns the maximum of an array along an axis ignoring NaN. |
| *cupy.percentile* | Computes the q-th percentile of the data along the specified axis. |

#### cupy.amin

cupy.**amin**(*a*, *axis=None*, *out=None*, *keepdims=False*, *dtype=None*)

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.
- **dtype** – Data type specifier.

**Returns** The minimum of a, along the axis if specified.

**Return type** *cupy.ndarray*

**See also:**

numpy.amin()

#### cupy.amax

cupy.**amax**(*a*, *axis=None*, *out=None*, *keepdims=False*, *dtype=None*)

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.
- **dtype** – Data type specifier.

**Returns** The maximum of a, along the axis if specified.

**Return type** *cupy.ndarray*

**See also:**

numpy.amax()

### cupy.nanmin

cupy.**nanmin**(*a*, *axis=None*, *out=None*, *keepdims=False*)

Returns the minimum of an array along an axis ignoring NaN.

When there is a slice whose elements are all NaN, a RuntimeWarning is raised and NaN is returned.

**Parameters**

- **a** (cupy.ndarray) – Array to take the minimum.
- **axis** (*int*) – Along which axis to take the minimum. The flattened array is used by default.
- **out** (cupy.ndarray) – Output array.
- **keepdims** (*bool*) – If True, the axis is remained as an axis of size one.

**Returns** The minimum of a, along the axis if specified.

**Return type** *cupy.ndarray*

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

**See also:**

numpy.nanmin()

### cupy.nanmax

cupy.**nanmax**(*a*, *axis=None*, *out=None*, *keepdims=False*)

Returns the maximum of an array along an axis ignoring NaN.

When there is a slice whose elements are all NaN, a RuntimeWarning is raised and NaN is returned.

**Parameters**

- **a** (cupy.ndarray) – Array to take the maximum.
- **axis** (*int*) – Along which axis to take the maximum. The flattened array is used by default.

- **out** (cupy.ndarray) – Output array.
- **keepdims** (*bool*) – If True, the axis is remained as an axis of size one.

**Returns** The maximum of a, along the axis if specified.

**Return type** *cupy.ndarray*

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

**See also:**

numpy.nanmax()

### cupy.percentile

cupy.**percentile**(*a*, *q*, *axis=None*, *out=None*, *interpolation='linear'*, *keepdims=False*)

Computes the q-th percentile of the data along the specified axis.

**Parameters**

- **a** (cupy.ndarray) – Array for which to compute percentiles.
- **q** (*float, tuple of floats or* cupy.ndarray) – Percentiles to compute in the range between 0 and 100 inclusive.
- **axis** (*int or tuple of ints*) – Along which axis or axes to compute the percentiles. The flattened array is used by default.
- **out** (cupy.ndarray) – Output array.
- **interpolation** (*str*) – Interpolation method when a quantile lies between two data points. linear interpolation is used by default. Supported interpolations are''lower'', higher, midpoint, nearest and linear.
- **keepdims** (*bool*) – If True, the axis is remained as an axis of size one.

**Returns** The percentiles of a, along the axis if specified.

**Return type** *cupy.ndarray*

**See also:**

numpy.percentile()

### Means and variances

| | |
|---|---|
| *cupy.average* | Returns the weighted average along an axis. |
| *cupy.mean* | Returns the arithmetic mean along an axis. |
| *cupy.var* | Returns the variance along an axis. |
| *cupy.std* | Returns the standard deviation along an axis. |
| *cupy.nanmean* | Returns the arithmetic mean along an axis ignoring NaN values. |
| *cupy.nanvar* | Returns the variance along an axis ignoring NaN values. |
| *cupy.nanstd* | Returns the standard deviation along an axis ignoring NaN values. |

### 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

| | |
|---|---|
| *cupy.histogram* | Computes the histogram of a set of data. |
| *cupy.bincount* | Count number of occurrences of each value in array of non-negative ints. |

### cupy.histogram

cupy.**histogram**(*x*, *bins=10*)
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.

**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.max(x) + 1, minlength).

**Return type** *cupy.ndarray*

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

**See also:**

numpy.bincount()

### Correlations

| | |
|---|---|
| *cupy.corrcoef* | Returns the Pearson product-moment correlation coefficients of an array. |
| *cupy.cov* | Returns the covariance matrix of an array. |

#### cupy.corrcoef

cupy.**corrcoef**(*a*, *y=None*, *rowvar=True*, *bias=None*, *ddof=None*)

Returns the Pearson product-moment correlation coefficients of an array.

**Parameters**

- **a** (cupy.ndarray) – Array to compute the Pearson product-moment correlation coefficients.
- **y** (cupy.ndarray) – An additional set of variables and observations.
- **rowvar** (*bool*) – If True, then each row represents a variable, with observations in the columns. Otherwise, the relationship is transposed.
- **bias** (*None*) – Has no effect, do not use.
- **ddof** (*None*) – Has no effect, do not use.

**Returns** The Pearson product-moment correlation coefficients of the input array.

**Return type** *cupy.ndarray*

**See also:**

numpy.corrcoef()

#### cupy.cov

cupy.**cov**(*a*, *y=None*, *rowvar=True*, *bias=False*, *ddof=None*)

Returns the covariance matrix of an array.

This function currently does not support fweights and aweights options.

**Parameters**

- **a** (cupy.ndarray) – Array to compute covariance matrix.
- **y** (cupy.ndarray) – An additional set of variables and observations.
- **rowvar** (*bool*) – If True, then each row represents a variable, with observations in the columns. Otherwise, the relationship is transposed.
- **bias** (*bool*) – If False, normalization is by (N - 1), where N is the number of observations given (unbiased estimate). If True, then normalization is by N.
- **ddof** (*int*) – If not None the default value implied by bias is overridden. Note that ddof=1 will return the unbiased estimate and ddof=0 will return the simple average.

**Returns** The covariance matrix of the input array.

**Return type** *cupy.ndarray*

**See also:**

numpy.cov()

### 3.3.15 CuPy-specific Functions

CuPy-specific functions are placed under cupyx namespace.

| | |
|---|---|
| *cupyx.rsqrt* | Returns the reciprocal square root. |
| *cupyx.scatter_add* | Adds given values to specified elements of an array. |
| *cupyx.scatter_max* | Stores a maximum value of elements specified by indices to an array. |
| *cupyx.scatter_min* | Stores a minimum value of elements specified by indices to an array. |

### cupyx.rsqrt

cupyx.**rsqrt = <ufunc 'cupy_rsqrt'>**
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

```
>>> 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

```
>>> 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

```
>>> 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.

## 3.4 SciPy-compatible Routines

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

### 3.4.1 Discrete Fourier transforms (`scipy.fft`)

#### Fast Fourier Transforms

| | |
|---|---|
| *cupyx.scipy.fft.fft* | Compute the one-dimensional FFT. |
| *cupyx.scipy.fft.ifft* | Compute the one-dimensional inverse FFT. |
| *cupyx.scipy.fft.fft2* | Compute the two-dimensional FFT. |
| *cupyx.scipy.fft.ifft2* | Compute the two-dimensional inverse FFT. |
| *cupyx.scipy.fft.fftn* | Compute the N-dimensional FFT. |
| *cupyx.scipy.fft.ifftn* | Compute the N-dimensional inverse FFT. |
| *cupyx.scipy.fft.rfft* | Compute the one-dimensional FFT for real input. |
| *cupyx.scipy.fft.irfft* | Compute the one-dimensional inverse FFT for real input. |
| *cupyx.scipy.fft.rfft2* | Compute the two-dimensional FFT for real input. |
| *cupyx.scipy.fft.irfft2* | Compute the two-dimensional inverse FFT for real input. |
| *cupyx.scipy.fft.rfftn* | Compute the N-dimensional FFT for real input. |
| *cupyx.scipy.fft.irfftn* | Compute the N-dimensional inverse FFT for real input. |

Continued on next page

Table 69 – continued from previous page

| | |
|---|---|
| *cupyx.scipy.fft.hfft* | Compute the FFT of a signal that has Hermitian symmetry. |
| *cupyx.scipy.fft.ihfft* | Compute the FFT of a signal that has Hermitian symmetry. |

### cupyx.scipy.fft.fft

cupyx.scipy.fft.**fft**(*x*, *n=None*, *axis=-1*, *norm=None*, *overwrite_x=False*)

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.

**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

cupyx.scipy.fft.**ifft**(*x*, *n=None*, *axis=-1*, *norm=None*, *overwrite_x=False*)

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.

**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

cupyx.scipy.fft.**fft2**(*x*, *s=None*, *axes=(-2, -1)*, *norm=None*, *overwrite_x=False*)
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.

**Returns** The transformed array which shape is specified by s and type will convert to complex if that of the input is another.

**Return type** *cupy.ndarray*

**See also:**

scipy.fft.fft2()

### cupyx.scipy.fft.ifft2

cupyx.scipy.fft.**ifft2**(*x*, *s=None*, *axes=(-2, -1)*, *norm=None*, *overwrite_x=False*)
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.

**Returns** The transformed array which shape is specified by s and type will convert to complex if that of the input is another.

**Return type** *cupy.ndarray*

**See also:**

scipy.fft.ifft2()

### cupyx.scipy.fft.fftn

cupyx.scipy.fft.**fftn**(*x*, *s=None*, *axes=None*, *norm=None*, *overwrite_x=False*)
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.

**Returns** The transformed array which shape is specified by `s` and type will convert to complex if that of the input is another.

**Return type** *cupy.ndarray*

**See also:**

`scipy.fft.fftn()`

### cupyx.scipy.fft.ifftn

`cupyx.scipy.fft.`**`ifftn`**(*x*, *s=None*, *axes=None*, *norm=None*, *overwrite_x=False*)

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.

**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*)

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.

**Returns** The transformed array.

**Return type** *cupy.ndarray*

**See also:**

scipy.fft.rfft()

### cupyx.scipy.fft.irfft

cupyx.scipy.fft.**irfft**(*x*, *n=None*, *axis=-1*, *norm=None*, *overwrite_x=False*)

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.

**Returns** The transformed array.

**Return type** *cupy.ndarray*

**Warning:** The input array may be modified in CUDA 10.1 and above, even when *overwrite_x is False*.

**See also:**

scipy.fft.irfft()

### cupyx.scipy.fft.rfft2

cupyx.scipy.fft.**rfft2**(*x*, *s=None*, *axes=(-2*, *-1)*, *norm=None*, *overwrite_x=False*)

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.

**Returns** The transformed array which shape is specified by s and type will convert to complex if the input is other. The length of the last axis transformed will be s[-1]//2+1.

**Return type** *cupy.ndarray*

**See also:**

`scipy.fft.rfft2()`

### cupyx.scipy.fft.irfft2

`cupyx.scipy.fft.`**`irfft2`**(*x*, *s=None*, *axes=(-2, -1)*, *norm=None*, *overwrite_x=False*)

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.

**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*

**Warning:** The input array may be modified in CUDA 10.1 and above, even when *overwrite_x is False*.

**See also:**

`scipy.fft.irfft2()`

### cupyx.scipy.fft.rfftn

`cupyx.scipy.fft.`**`rfftn`**(*x*, *s=None*, *axes=None*, *norm=None*, *overwrite_x=False*)

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.

**Returns** The transformed array which shape is specified by `s` and type will convert to complex if the input is other. The length of the last axis transformed will be `s[-1]//2+1`.

**Return type** *cupy.ndarray*

**See also:**

`scipy.fft.rfftn()`

### cupyx.scipy.fft.irfftn

cupyx.scipy.fft.**irfftn**(*x*, *s=None*, *axes=None*, *norm=None*, *overwrite_x=False*)

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.

**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*

**Warning:** The input array may be modified in CUDA 10.1 and above, even when *overwrite_x is False*.

**See also:**

scipy.fft.irfftn()

### cupyx.scipy.fft.hfft

cupyx.scipy.fft.**hfft**(*x*, *n=None*, *axis=-1*, *norm=None*, *overwrite_x=False*)

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.

**Returns** The transformed array which shape is specified by n and type will convert to complex if the input is other. If n is not given, the length of the transformed axis is 2*(m-1) where *m* is the length of the transformed axis of the input.

**Return type** *cupy.ndarray*

**See also:**

scipy.fft.hfft()

#### cupyx.scipy.fft.ihfft

cupyx.scipy.fft.**ihfft**(*x*, *n=None*, *axis=-1*, *norm=None*, *overwrite_x=False*)

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.

**Returns** The transformed array which shape is specified by n and type will convert to complex if the input is other. The length of the transformed axis is n//2+1.

**Return type** *cupy.ndarray*

**See also:**

scipy.fft.ihfft()

#### Code compatibility features

1. The boolean switch cupy.fft.config.enable_nd_planning also affects the FFT functions in this module, see *FFT Functions*. Moreover, as with other FFT modules in CuPy, FFT functions in this module can take advantage of an existing cuFFT plan (returned by *cupyx.scipy.fftpack.get_fft_plan()*) when used as a context manager.
2. 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. This is not an in-place FFT, the user should always use the return value from the functions, e.g. x = cupyx.scipy.fft.fft(x, ..., overwrite_x=True, ...).
3. 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.

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

### 3.4.2 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

| | |
|---|---|
| *cupyx.scipy.fftpack.fft* | Compute the one-dimensional FFT. |
| *cupyx.scipy.fftpack.ifft* | Compute the one-dimensional inverse FFT. |
| *cupyx.scipy.fftpack.fft2* | Compute the two-dimensional FFT. |
| *cupyx.scipy.fftpack.ifft2* | Compute the two-dimensional inverse FFT. |
| *cupyx.scipy.fftpack.fftn* | Compute the N-dimensional FFT. |
| *cupyx.scipy.fftpack.ifftn* | Compute the N-dimensional inverse FFT. |
| *cupyx.scipy.fftpack.rfft* | Compute the one-dimensional FFT for real input. |
| *cupyx.scipy.fftpack.irfft* | Compute the one-dimensional inverse FFT for real input. |
| *cupyx.scipy.fftpack.get_fft_plan* | Generate a CUDA FFT plan for transforming up to three axes. |

### cupyx.scipy.fftpack.fft

cupyx.scipy.fftpack.**fft**(*x*, *n=None*, *axis=-1*, *overwrite_x=False*, *plan=None*)

Compute the one-dimensional FFT.

**Parameters**

- **x** (cupy.ndarray) – Array to be transformed.
- **n** (*None or int*) – Length of the transformed axis of the output. If `n` is not given, the length of the input along the axis specified by `axis` is used.
- **axis** (*int*) – Axis over which to compute the FFT.
- **overwrite_x** (*bool*) – If True, the contents of `x` can be destroyed.
- **plan** (*cupy.cuda.cufft.Plan1d*) – a cuFFT plan for transforming `x` over `axis`, which can be obtained using:

```
plan = cupyx.scipy.fftpack.get_fft_plan(x, axis)
```

Note that *plan* is defaulted to None, meaning CuPy will use an auto-generated plan behind the scene.

**Returns** The transformed array which shape is specified by `n` and type will convert to complex if that of the input is another.

**Return type** *cupy.ndarray*

**Note:** The argument *plan* is currently experimental and the interface may be changed in the future version.

**See also:**

scipy.fftpack.fft()

### cupyx.scipy.fftpack.ifft

cupyx.scipy.fftpack.**ifft**(*x*, *n=None*, *axis=-1*, *overwrite_x=False*, *plan=None*)

Compute the one-dimensional inverse FFT.

**Parameters**

- **x** (cupy.ndarray) – Array to be transformed.

- **n** (*None or int*) – Length of the transformed axis of the output. If `n` is not given, the length of the input along the axis specified by `axis` is used.
- **axis** (*int*) – Axis over which to compute the FFT.
- **overwrite_x** (*bool*) – If True, the contents of `x` can be destroyed.
- **plan** (*cupy.cuda.cufft.Plan1d*) – a cuFFT plan for transforming `x` over `axis`, which can be obtained using:

```
plan = cupyx.scipy.fftpack.get_fft_plan(x, axis)
```

Note that *plan* is defaulted to None, meaning CuPy will use an auto-generated plan behind the scene.

**Returns** The transformed array which shape is specified by `n` and type will convert to complex if that of the input is another.

**Return type** *cupy.ndarray*

---

**Note:** The argument *plan* is currently experimental and the interface may be changed in the future version.

---

**See also:**

`scipy.fftpack.ifft()`

### cupyx.scipy.fftpack.fft2

cupyx.scipy.fftpack.**fft2**(*x*, *shape=None*, *axes=(-2, -1)*, *overwrite_x=False*, *plan=None*)

Compute the two-dimensional FFT.

**Parameters**

- **x** (`cupy.ndarray`) – Array to be transformed.
- **shape** (*None or tuple of ints*) – Shape of the transformed axes of the output. If `shape` is not given, the lengths of the input along the axes specified by `axes` are used.
- **axes** (*tuple of ints*) – Axes over which to compute the FFT.
- **overwrite_x** (*bool*) – If True, the contents of `x` can be destroyed.
- **plan** (*cupy.cuda.cufft.PlanNd*) – a cuFFT plan for transforming `x` over `axes`, which can be obtained using:

```
plan = cupyx.scipy.fftpack.get_fft_plan(x, axes)
```

Note that *plan* is defaulted to None, meaning CuPy will either use an auto-generated plan behind the scene if cupy.fft.config. enable_nd_planning = True, or use no cuFFT plan if it is set to False.

**Returns** The transformed array which shape is specified by `shape` and type will convert to complex if that of the input is another.

**Return type** *cupy.ndarray*

**See also:**

`scipy.fftpack.fft2()`

**Note:** The argument *plan* is currently experimental and the interface may be changed in the future version.

### cupyx.scipy.fftpack.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*) – 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.
- **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*) – a cuFFT plan for transforming x over axes, which can be obtained using:

```
plan = cupyx.scipy.fftpack.get_fft_plan(x, axes)
```

Note that *plan* is defaulted to None, meaning CuPy will either use an auto-generated plan behind the scene if cupy.fft.config. enable_nd_planning = True, or use no cuFFT plan if it is set to False.

**Returns** The transformed array which shape is specified by `shape` and type will convert to complex if that of the input is another.

**Return type** *cupy.ndarray*

**See also:**

`scipy.fftpack.fftn()`

---

**Note:** The argument *plan* is currently experimental and the interface may be changed in the future version.

---

### cupyx.scipy.fftpack.ifftn

`cupyx.scipy.fftpack.`**`ifftn`**(*x*, *shape=None*, *axes=None*, *overwrite_x=False*, *plan=None*)

Compute the N-dimensional inverse FFT.

**Parameters**

- **`x`** (`cupy.ndarray`) – Array to be transformed.
- **`shape`** (*`None or tuple of ints`*) – Shape of the transformed axes of the output. If `shape` is not given, the lengths of the input along the axes specified by `axes` are used.
- **`axes`** (*`tuple of ints`*) – Axes over which to compute the FFT.
- **`overwrite_x`** (*`bool`*) – If True, the contents of `x` can be destroyed.
- **`plan`** (*`cupy.cuda.cufft.PlanNd`*) – a cuFFT plan for transforming `x` over `axes`, which can be obtained using:

```
plan = cupyx.scipy.fftpack.get_fft_plan(x, axes)
```

Note that *plan* is defaulted to None, meaning CuPy will either use an auto-generated plan behind the scene if cupy.fft.config. enable_nd_planning = True, or use no cuFFT plan if it is set to False.

**Returns** The transformed array which shape is specified by `shape` and type will convert to complex if that of the input is another.

**Return type** *cupy.ndarray*

**See also:**

`scipy.fftpack.ifftn()`

---

**Note:** The argument *plan* is currently experimental and the interface may be changed in the future version.

---

### cupyx.scipy.fftpack.rfft

cupyx.scipy.fftpack.**rfft**(*x*, *n=None*, *axis=-1*, *overwrite_x=False*)

Compute the one-dimensional FFT for real input.

The returned real array contains

```
[y(0),Re(y(1)),Im(y(1)),...,Re(y(n/2))]  # if n is even
[y(0),Re(y(1)),Im(y(1)),...,Re(y(n/2)),Im(y(n/2))]  # if n is odd
```

**Parameters**

- **x** (cupy.ndarray) – Array to be transformed.
- **n** (*None or int*) – Length of the transformed axis of the output. If n is not given, the length of the input along the axis specified by axis is used.
- **axis** (*int*) – Axis over which to compute the FFT.
- **overwrite_x** (*bool*) – If True, the contents of x can be destroyed.

**Returns** The transformed array.

**Return type** *cupy.ndarray*

**See also:**

scipy.fftpack.rfft()

### cupyx.scipy.fftpack.irfft

cupyx.scipy.fftpack.**irfft**(*x*, *n=None*, *axis=-1*, *overwrite_x=False*)

Compute the one-dimensional inverse FFT for real input.

**Parameters**

- **x** (cupy.ndarray) – Array to be transformed.
- **n** (*None or int*) – Length of the transformed axis of the output. If n is not given, the length of the input along the axis specified by axis is used.
- **axis** (*int*) – Axis over which to compute the FFT.
- **overwrite_x** (*bool*) – If True, the contents of x can be destroyed.

**Returns** The transformed array.

**Return type** *cupy.ndarray*

**Warning:** The input array may be modified in CUDA 10.1 and above, even when *overwrite_x is False*.

**See also:**

scipy.fftpack.irfft()

### cupyx.scipy.fftpack.get_fft_plan

cupyx.scipy.fftpack.**get_fft_plan**(*a*, *shape=None*, *axes=None*, *value_type='C2C'*)

Generate a CUDA FFT plan for transforming up to three axes.

**Parameters**

- **a** (cupy.ndarray) – Array to be transform, assumed to be either C- or F- contiguous.
- **shape** (*None or tuple of ints*) – Shape of the transformed axes of the output. If shape is not given, the lengths of the input along the axes specified by axes are used.
- **axes** (*None or int or tuple of int*) – The axes of the array to transform. If *None*, it is assumed that all axes are transformed.

  Currently, for performing N-D transform these must be a set of up to three adjacent axes, and must include either the first or the last axis of the array.
- **value_type** (*'C2C'*) – The FFT type to perform. Currently only complex-to-complex transforms are supported.

**Returns** a cuFFT plan for either 1D transform (cupy.cuda.cufft.Plan1d) or N-D transform (cupy.cuda.cufft.PlanNd).

---

**Note:** The returned plan can not only be passed as one of the arguments of the functions in cupyx.scipy.fftpack, but also be used as a context manager for both cupy.fft and cupyx.scipy.fftpack functions:

```
x = cupy.random.random(16).reshape(4, 4).astype(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. The get_fft_plan function has no counterpart in scipy.fftpack. It returns a cuFFT plan that can be passed to the FFT functions in this module (using the argument plan) to accelarate the computation. The argument plan is currently experimental and the interface may be changed in the future version.
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 destroyed and replaced by the output. 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, ...)`.

### 3.4.3 Multi-dimensional image processing

CuPy provides multi-dimensional image processing functions. It supports a subset of `scipy.ndimage` interface.

#### Interpolation

| | |
|---|---|
| *cupyx.scipy.ndimage.affine_transform* | Apply an affine transformation. |
| *cupyx.scipy.ndimage.convolve* | Multi-dimensional convolution. |
| *cupyx.scipy.ndimage.correlate* | Multi-dimensional correlate. |
| *cupyx.scipy.ndimage.map_coordinates* | Map the input array to new coordinates by interpolation. |
| *cupyx.scipy.ndimage.rotate* | Rotate an array. |
| *cupyx.scipy.ndimage.shift* | Shift an array. |
| *cupyx.scipy.ndimage.zoom* | Zoom an array. |

#### 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, ..., 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.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()`

### cupyx.scipy.ndimage.correlate

cupyx.scipy.ndimage.**correlate**(*input*, *weights*, *output=None*, *mode='reflect'*, *cval=0.0*, *origin=0*)

Multi-dimensional correlate.

The array is correlated with the given kernel.

**Parameters**

- **input** (cupy.ndarray) – The input array.
- **weights** (cupy.ndarray) – Array of weights, same number of dimensions as input
- **output** (cupy.ndarray, *dtype or None*) – The array in which to place the output.
- **mode** (*str*) – The array borders are handled according to the given mode (`'reflect'`, `'constant'`, `'nearest'`, `'mirror'`, `'wrap'`). Default is `'reflect'`.
- **cval** (*scalar*) – Value to fill past edges of input if mode is `constant`. Default is `0.0`.
- **origin** (*scalar or tuple of scalar*) – The origin parameter controls the placement of the filter, relative to the center of the current element of the input. Default of 0 is equivalent to `(0,)*input.ndim`.

**Returns** The result of correlate.

**Return type** *cupy.ndarray*

**See also:**

`scipy.ndimage.correlate()`

### 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()`

### 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

cupyx.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**

- **input** (cupy.ndarray) – The input array.
- **shift** (*float or sequence*) – The shift along the axes. If a float, shift is the same for each axis. If a sequence, shift should contain one value for each axis.
- **output** (cupy.ndarray *or dtype*) – The array in which to place the output, or the dtype of the returned array.
- **order** (*int*) – The order of the spline interpolation. 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()

### cupyx.scipy.ndimage.zoom

cupyx.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**

- **input** (cupy.ndarray) – The input array.
- **zoom** (*float or sequence*) – The zoom factor along the axes. If a float, zoom is the same for each axis. If a sequence, zoom should contain one value for each axis.
- **output** (cupy.ndarray *or dtype*) – The array in which to place the output, or the dtype of the returned array.
- **order** (*int*) – The order of the spline interpolation. 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()`

#### OpenCV mode

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

### 3.4.4 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

| | |
|---|---|
| *cupyx.scipy.sparse.coo_matrix* | COOrdinate format sparse matrix. |
| *cupyx.scipy.sparse.csc_matrix* | Compressed Sparse Column matrix. |
| *cupyx.scipy.sparse.csr_matrix* | Compressed Sparse Row matrix. |
| *cupyx.scipy.sparse.dia_matrix* | Sparse matrix with DIAgonal storage. |
| *cupyx.scipy.sparse.spmatrix* | Base class of all sparse matrixes. |

#### cupyx.scipy.sparse.coo_matrix

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

COOrdinate format sparse matrix.

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-dimenaional *cupy.ndarray*.

**Parameters**

- **arg1** – Arguments for the initializer.
- **shape** (*tuple*) – Shape of a matrix. Its length must be two.
- **dtype** – Data type. It must be an argument of `numpy.dtype`.
- **copy** (*bool*) – If `True`, copies of given data are always used.

**See also:**

`scipy.sparse.coo_matrix`

### Methods

**__len__**()

**__iter__**()

**arcsin**()
Elementwise arcsin.

**arcsinh**()
Elementwise arcsinh.

**arctan**()
Elementwise arctan.

**arctanh**()
Elementwise arctanh.

**asformat**(*format*)
Return this matrix in a given sparse format.

**Parameters** **format** (*str or None*) – Format you need.

**asfptype**()
Upcasts matrix to a floating point format.

When the matrix has floating point type, the method returns itself. Otherwise it makes a copy with floating point type and the same format.

**Returns** A matrix with float type.

**Return type** *cupyx.scipy.sparse.spmatrix*

**astype**(*t*)
Casts the array to given data type.

**Parameters** **dtype** – Type specifier.

**Returns** A copy of the array with a given type.

**ceil**()
Elementwise ceil.

**conj**(*copy=True*)
Element-wise complex conjugation.

If the matrix is of non-complex data type and *copy* is False, this method does nothing and the data is not copied.

**Parameters** **copy** (*bool*) – If True, the result is guaranteed to not share data with self.

**Returns** The element-wise complex conjugate.

**Return type** *cupyx.scipy.sparse.spmatrix*

**conjugate**(*copy=True*)
Element-wise complex conjugation.

If the matrix is of non-complex data type and *copy* is False, this method does nothing and the data is not copied.

**Parameters** **copy** (*bool*) – If True, the result is guaranteed to not share data with self.

**Returns** The element-wise complex conjugate.

**Return type** *cupyx.scipy.sparse.spmatrix*

**copy**()
Returns a copy of this matrix.

No data/indices will be shared between the returned value and current matrix.

**count_nonzero**()
Returns number of non-zero entries.

---

**Note:** This method counts the actual number of non-zero entories, which does not include explicit zero entries. Instead `nnz` returns the number of entries including explicit zeros.

---

**Returns** Number of non-zero entries.

**deg2rad**()
Elementwise deg2rad.

**diagonal**(*k=0*)
Returns the k-th diagonal of the matrix.

**Parameters**

- **k** (*int, optional*) – Which diagonal to get, corresponding to elements
- **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 entories in place.

**expm1**()
Elementwise expm1.

**floor**()
Elementwise floor.

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

**Parameters** **stream** (cupy.cuda.Stream) – CUDA stream object. If it is given, the copy runs asynchronously. Otherwise, the copy is synchronous.

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

**Return type** *scipy.sparse.coo_matrix*

**getH**()

**get_shape**()
Returns the shape of the matrix.

**Returns** Shape of the matrix.

**Return type** tuple

**getformat**()

**getmaxprint**()

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

**log1p**()
Elementwise log1p.

**maximum**(*other*)

**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.

**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 comuted. If it is `None`, it computes the sum of all the elements. Select from `{None, 0, 1, -2, -1}`.
- **dtype** – The type of returned matrix. If it is not specified, type of the array is used.
- **out** (`cupy.ndarray`) – Output matrix.

**Returns** Summed array.

**Return type** *cupy.ndarray*

**See also:**

`scipy.sparse.spmatrix.sum()`

**sum_duplicates**()
Eliminate duplicate matrix entries by adding them together.

**Warning:** When sorting the indices, CuPy follows the convention of cuSPARSE, which is different from that of SciPy. Therefore, the order of the output indices may differ:

```
>>> #     1 0 0
>>> # A = 1 1 0
>>> #     1 1 1
>>> data = cupy.array([1, 1, 1, 1, 1, 1], 'f')
>>> row = cupy.array([0, 1, 1, 2, 2, 2], 'i')
>>> col = cupy.array([0, 0, 1, 0, 1, 2], 'i')
>>> A = cupyx.scipy.sparse.coo_matrix((data, (row, col)),
...                                   shape=(3, 3))
>>> a = A.get()
>>> A.sum_duplicates()
>>> a.sum_duplicates()  # a is scipy.sparse.coo_matrix
>>> A.row
array([0, 1, 1, 2, 2, 2], dtype=int32)
>>> a.row
array([0, 1, 2, 1, 2, 2], dtype=int32)
>>> A.col
array([0, 0, 1, 0, 1, 2], dtype=int32)
>>> a.col
array([0, 0, 0, 1, 1, 2], dtype=int32)
```

**Warning:** Calling this function might synchronize the device.

**See also:**

`scipy.sparse.coo_matrix.sum_duplicates()`

**tan**()
Elementwise tan.

**tanh**()
Elementwise tanh.

**toarray**(*order=None*, *out=None*)
Returns a dense matrix representing the same value.

**Parameters**

- **order** (*str*) – Not supported.
- **out** – Not supported.

**Returns** Dense array representing the same value.

**Return type** *cupy.ndarray*

**See also:**

`scipy.sparse.coo_matrix.toarray()`

**tobsr**(*blocksize=None*, *copy=False*)
Convert this matrix to Block Sparse Row format.

**tocoo**(*copy=False*)
Converts the matrix to 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*

**tocsc**(*copy=False*)
Converts the matrix to Compressed Sparse Column format.

**Parameters** **copy** (*bool*) – If `False`, it shares data arrays as much as possible. Actually this option is ignored because all arrays in a matrix cannot be shared in coo to csc conversion.

**Returns** Converted matrix.

**Return type** *cupyx.scipy.sparse.csc_matrix*

**tocsr**(*copy=False*)
Converts the matrix to Compressed Sparse Row format.

**Parameters** **copy** (*bool*) – If `False`, it shares data arrays as much as possible. Actually this option is ignored because all arrays in a matrix cannot be shared in coo to csr conversion.

**Returns** Converted matrix.

**Return type** *cupyx.scipy.sparse.csr_matrix*

**todense**(*order=None*, *out=None*)
Return a dense matrix representation of this matrix.

**todia**(*copy=False*)
Convert this matrix to sparse DIAgonal format.

**todok**(*copy=False*)
Convert this matrix to Dictionary Of Keys format.

**tolil**(*copy=False*)
Convert this matrix to LInked List format.

**transpose**(*axes=None*, *copy=False*)
Returns a transpose matrix.

**Parameters**

- **axes** – This option is not supported.
- **copy** (*bool*) – If `True`, a returned matrix shares no data. Otherwise, it shared data arrays as much as possible.

**Returns** Transpose matrix.

**Return type** *cupyx.scipy.sparse.spmatrix*

**trunc**()
Elementwise trunc.

**__eq__**(*other*)
Return self==value.

**__ne__**(*other*)
Return self!=value.

**__lt__**(*other*)
Return self<value.

**__le__**(*other*)
Return self<=value.

**__gt__**(*other*)
Return self>value.

**__ge__**(*other*)
Return self>=value.

**__nonzero__**()

**__bool__**()

### Attributes

**A**
Dense ndarray representation of this matrix.

This property is equivalent to *toarray()* method.

**H**

**T**

**device**
CUDA device on which this array resides.

**dtype**
Data type of the matrix.

**format = 'coo'**

**has_canonical_format**

**ndim**

**nnz**

**shape**

**size**

### cupyx.scipy.sparse.csc_matrix

**class** cupyx.scipy.sparse.**csc_matrix**(*arg1*, *shape=None*, *dtype=None*, *copy=False*)

Compressed Sparse Column matrix.

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-dimenaional *cupy.ndarray*.

**Parameters**

- **arg1** – Arguments for the initializer.
- **shape** (*tuple*) – Shape of a matrix. Its length must be two.
- **dtype** – Data type. It must be an argument of `numpy.dtype`.
- **copy** (*bool*) – If `True`, copies of given arrays are always used.

**See also:**

`scipy.sparse.csc_matrix`

#### Methods

**__getitem__**(*slices*)

**__len__**()

**__iter__**()

**arcsin**()
Elementwise arcsin.

**arcsinh**()
Elementwise arcsinh.

**arctan**()
Elementwise arctan.

**arctanh**()
Elementwise arctanh.

**asformat**(*format*)
Return this matrix in a given sparse format.

**Parameters** **format** (*str or None*) – Format you need.

**asfptype**()
Upcasts matrix to a floating point format.

When the matrix has floating point type, the method returns itself. Otherwise it makes a copy with floating point type and the same format.

**Returns** A matrix with float type.

**Return type** *cupyx.scipy.sparse.spmatrix*

**astype**(*t*)
Casts the array to given data type.

**Parameters** **dtype** – Type specifier.

**Returns** A copy of the array with a given type.

**ceil**()
Elementwise ceil.

**conj**(*copy=True*)
Element-wise complex conjugation.

If the matrix is of non-complex data type and *copy* is False, this method does nothing and the data is not copied.

**Parameters** **copy** (*bool*) – If True, the result is guaranteed to not share data with self.

**Returns** The element-wise complex conjugate.

**Return type** *cupyx.scipy.sparse.spmatrix*

**conjugate**(*copy=True*)
Element-wise complex conjugation.

If the matrix is of non-complex data type and *copy* is False, this method does nothing and the data is not copied.

**Parameters** **copy** (*bool*) – If True, the result is guaranteed to not share data with self.

**Returns** The element-wise complex conjugate.

**Return type** *cupyx.scipy.sparse.spmatrix*

**copy**()
Returns a copy of this matrix.

No data/indices will be shared between the returned value and current matrix.

**count_nonzero**()
Returns number of non-zero entries.

---

**Note:** This method counts the actual number of non-zero entories, which does not include explicit zero entries. Instead `nnz` returns the number of entries including explicit zeros.

---

**Returns** Number of non-zero entries.

**deg2rad**()
Elementwise deg2rad.

**diagonal**(*k=0*)
Returns the k-th diagonal of the matrix.

**Parameters**

- **k** (*int, optional*) – Which diagonal to get, corresponding to elements
- **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 entories in place.

**expm1**()
Elementwise expm1.

**floor**()
Elementwise floor.

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

> **Warning:** You need to install SciPy to use this method.

**Parameters** **stream** (cupy.cuda.Stream) – CUDA stream object. If it is given, the copy runs asynchronously. Otherwise, the copy is synchronous.

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

**Return type** *scipy.sparse.csc_matrix*

**getH**()

**get_shape**()
Returns the shape of the matrix.

**Returns** Shape of the matrix.

**Return type** tuple

**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*)

**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 the matrix in place.

**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 comuted. 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**()

**tan**()
Elementwise tan.

**tanh**()
Elementwise tanh.

**toarray**(*order=None*, *out=None*)
Returns a dense matrix representing the same value.

**Parameters**

- **order** (*{'C', 'F', None}*) – Whether to store data in C (row-major) order or F (column-major) order. Default is C-order.
- **out** – Not supported.

**Returns** Dense array representing the same matrix.

**Return type** *cupy.ndarray*

**See also:**

`scipy.sparse.csc_matrix.toarray()`

**tobsr**(*blocksize=None*, *copy=False*)
Convert this matrix to Block Sparse Row format.

**tocoo**(*copy=False*)
Converts the matrix to 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*

**tocsc**(*copy=None*)
Converts the matrix to Compressed Sparse Column format.

**Parameters** **copy** (*bool*) – If `False`, the method returns itself. Otherwise it makes a copy of the matrix.

**Returns** Converted matrix.

**Return type** *cupyx.scipy.sparse.csc_matrix*

**tocsr**(*copy=False*)
Converts the matrix to Compressed Sparse Row format.

**Parameters** **copy** (*bool*) – If `False`, it shares data arrays as much as possible. Actually this option is ignored because all arrays in a matrix cannot be shared in csr to csc conversion.

**Returns** Converted matrix.

**Return type** *cupyx.scipy.sparse.csr_matrix*

**todense**(*order=None*, *out=None*)
Return a dense matrix representation of this matrix.

**todia**(*copy=False*)
Convert this matrix to sparse DIAgonal format.

**todok**(*copy=False*)
Convert this matrix to Dictionary Of Keys format.

**tolil**(*copy=False*)
Convert this matrix to LInked List format.

**transpose**(*axes=None*, *copy=False*)
Returns a transpose matrix.

**Parameters**

- **axes** – This option is not supported.
- **copy** (*bool*) – If `True`, a returned matrix shares no data. Otherwise, it shared data arrays as much as possible.

**Returns** Transpose matrix.

**Return type** *cupyx.scipy.sparse.spmatrix*

**trunc**()
Elementwise trunc.

**__eq__**(*other*)
Return self==value.

**__ne__**(*other*)
Return self!=value.

**__lt__**(*other*)
Return self<value.

**__le__**(*other*)
Return self<=value.

**__gt__**(*other*)
Return self>value.

**__ge__**(*other*)
Return self>=value.

**__nonzero__**()

**__bool__**()

#### Attributes

**A**
Dense ndarray representation of this matrix.

This property is equivalent to *toarray()* method.

**H**

**T**

**device**
CUDA device on which this array resides.

**dtype**
Data type of the matrix.

**format = 'csc'**

**has_canonical_format**

**ndim**

**nnz**

**shape**

**size**

### cupyx.scipy.sparse.csr_matrix

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

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-dimenaional *cupy.ndarray*.

**Parameters**

- **arg1** – Arguments for the initializer.
- **shape** (*tuple*) – Shape of a matrix. Its length must be two.
- **dtype** – Data type. It must be an argument of `numpy.dtype`.
- **copy** (*bool*) – If `True`, copies of given arrays are always used.

**See also:**

`scipy.sparse.csr_matrix`

### Methods

**__getitem__**(*slices*)

**__len__**()

**__iter__**()

**arcsin**()
Elementwise arcsin.

**arcsinh**()
Elementwise arcsinh.

**arctan**()
Elementwise arctan.

**arctanh**()
Elementwise arctanh.

**asformat**(*format*)
Return this matrix in a given sparse format.

**Parameters** **format** (*str or None*) – Format you need.

**asfptype**()
Upcasts matrix to a floating point format.

When the matrix has floating point type, the method returns itself. Otherwise it makes a copy with floating point type and the same format.

**Returns** A matrix with float type.

**Return type** *cupyx.scipy.sparse.spmatrix*

**astype**(*t*)
Casts the array to given data type.

**Parameters** **dtype** – Type specifier.

**Returns** A copy of the array with a given type.

**ceil**()
Elementwise ceil.

**conj**(*copy=True*)
Element-wise complex conjugation.

If the matrix is of non-complex data type and *copy* is False, this method does nothing and the data is not copied.

**Parameters** **copy** (*bool*) – If True, the result is guaranteed to not share data with self.

**Returns** The element-wise complex conjugate.

**Return type** *cupyx.scipy.sparse.spmatrix*

**conjugate**(*copy=True*)
Element-wise complex conjugation.

If the matrix is of non-complex data type and *copy* is False, this method does nothing and the data is not copied.

**Parameters** **copy** (*bool*) – If True, the result is guaranteed to not share data with self.

**Returns** The element-wise complex conjugate.

**Return type** *cupyx.scipy.sparse.spmatrix*

**copy**()
Returns a copy of this matrix.

No data/indices will be shared between the returned value and current matrix.

**count_nonzero**()
Returns number of non-zero entries.

---

**Note:** This method counts the actual number of non-zero entories, which does not include explicit zero entries. Instead `nnz` returns the number of entries including explicit zeros.

---

**Returns** Number of non-zero entries.

**deg2rad**()
Elementwise deg2rad.

**diagonal**(*k=0*)
Returns the k-th diagonal of the matrix.

**Parameters**

- **k** (*int, optional*) – Which diagonal to get, corresponding to elements
- **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 entories in place.

**expm1**()
Elementwise expm1.

**floor**()
Elementwise floor.

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

**Parameters** **stream** (cupy.cuda.Stream) – CUDA stream object. If it is given, the copy runs asynchronously. Otherwise, the copy is synchronous.

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

**Return type** *scipy.sparse.csr_matrix*

**getH**()

**get_shape**()
Returns the shape of the matrix.

**Returns** Shape of the matrix.

**Return type** tuple

**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*)

**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 the matrix in place.

**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 comuted. 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**()

**tan**()
Elementwise tan.

**tanh**()
Elementwise tanh.

**toarray**(*order=None*, *out=None*)
Returns a dense matrix representing the same value.

**Parameters**

- **order** (*{'C', 'F', None}*) – Whether to store data in C (row-major) order or F (column-major) order. Default is C-order.
- **out** – Not supported.

**Returns** Dense array representing the same matrix.

**Return type** *cupy.ndarray*

**See also:**

`scipy.sparse.csr_matrix.toarray()`

**tobsr**(*blocksize=None*, *copy=False*)
Convert this matrix to Block Sparse Row format.

**tocoo**(*copy=False*)
Converts the matrix to 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*

**tocsc**(*copy=False*)
Converts the matrix to Compressed Sparse Column format.

**Parameters** **copy** (*bool*) – If `False`, it shares data arrays as much as possible. Actually this option is ignored because all arrays in a matrix cannot be shared in csr to csc conversion.

**Returns** Converted matrix.

**Return type** *cupyx.scipy.sparse.csc_matrix*

**tocsr**(*copy=False*)
Converts the matrix to Compressed Sparse Row format.

**Parameters** **copy** (*bool*) – If `False`, the method returns itself. Otherwise it makes a copy of the matrix.

**Returns** Converted matrix.

**Return type** *cupyx.scipy.sparse.csr_matrix*

**todense**(*order=None*, *out=None*)
Return a dense matrix representation of this matrix.

**todia**(*copy=False*)
Convert this matrix to sparse DIAgonal format.

**todok**(*copy=False*)
Convert this matrix to Dictionary Of Keys format.

**tolil**(*copy=False*)
Convert this matrix to LInked List format.

**transpose**(*axes=None*, *copy=False*)
Returns a transpose matrix.

**Parameters**

- **axes** – This option is not supported.
- **copy** (*bool*) – If `True`, a returned matrix shares no data. Otherwise, it shared data arrays as much as possible.

**Returns** Transpose matrix.

**Return type** *cupyx.scipy.sparse.spmatrix*

**trunc**()
Elementwise trunc.

**__eq__**(*other*)
Return self==value.

**__ne__**(*other*)
Return self!=value.

**__lt__**(*other*)
Return self<value.

**__le__**(*other*)
Return self<=value.

**__gt__**(*other*)
Return self>value.

**__ge__**(*other*)
Return self>=value.

**__nonzero__**()

**__bool__**()

### Attributes

**A**
Dense ndarray representation of this matrix.

This property is equivalent to *toarray()* method.

**H**

**T**

**device**
CUDA device on which this array resides.

**dtype**
Data type of the matrix.

**format = 'csr'**

**has_canonical_format**

**ndim**

**nnz**

**shape**

**size**

## cupyx.scipy.sparse.dia_matrix

**class** cupyx.scipy.sparse.**dia_matrix**(*arg1*, *shape=None*, *dtype=None*, *copy=False*)
Sparse matrix with DIAgonal storage.

Now it has only one initializer format below:

`dia_matrix((data, offsets))`

**Parameters**

- **arg1** – Arguments for the initializer.
- **shape** (*tuple*) – Shape of a matrix. Its length must be two.
- **dtype** – Data type. It must be an argument of `numpy.dtype`.
- **copy** (*bool*) – If `True`, copies of given arrays are always used.

**See also:**

`scipy.sparse.dia_matrix`

### Methods

**__len__**()

**__iter__**()

**arcsin**()
Elementwise arcsin.

**arcsinh**()
Elementwise arcsinh.

**arctan**()
Elementwise arctan.

**arctanh**()
Elementwise arctanh.

**asformat**(*format*)
Return this matrix in a given sparse format.

**Parameters** **format** (*str or None*) – Format you need.

**asfptype**()
Upcasts matrix to a floating point format.

When the matrix has floating point type, the method returns itself. Otherwise it makes a copy with floating point type and the same format.

**Returns** A matrix with float type.

**Return type** *cupyx.scipy.sparse.spmatrix*

**astype**(*t*)
Casts the array to given data type.

**Parameters** **dtype** – Type specifier.

**Returns** A copy of the array with a given type.

**ceil**()
Elementwise ceil.

**conj**(*copy=True*)
Element-wise complex conjugation.

If the matrix is of non-complex data type and *copy* is False, this method does nothing and the data is not copied.

**Parameters** **copy** (*bool*) – If True, the result is guaranteed to not share data with self.

**Returns** The element-wise complex conjugate.

**Return type** *cupyx.scipy.sparse.spmatrix*

**conjugate**(*copy=True*)
Element-wise complex conjugation.

If the matrix is of non-complex data type and *copy* is False, this method does nothing and the data is not copied.

**Parameters** **copy** (*bool*) – If True, the result is guaranteed to not share data with self.

**Returns** The element-wise complex conjugate.

**Return type** *cupyx.scipy.sparse.spmatrix*

**copy**()
Returns a copy of this matrix.

No data/indices will be shared between the returned value and current matrix.

**count_nonzero**()
Returns number of non-zero entries.

---

**Note:** This method counts the actual number of non-zero entories, which does not include explicit zero entries. Instead `nnz` returns the number of entries including explicit zeros.

---

**Returns** Number of non-zero entries.

**deg2rad**()
Elementwise deg2rad.

**diagonal**(*k=0*)
Returns the k-th diagonal of the matrix.

**Parameters**

- **k** (*int, optional*) – Which diagonal to get, corresponding to elements
- **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*)

**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.

**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 comuted. If it is `None`, it computes the sum of all the elements. Select from `{None, 0, 1, -2, -1}`.
- **dtype** – The type of returned matrix. If it is not specified, type of the array is used.
- **out** (`cupy.ndarray`) – Output matrix.

**Returns** Summed array.

**Return type** *cupy.ndarray*

**See also:**

`scipy.sparse.spmatrix.sum()`

**tan**()
Elementwise tan.

**tanh**()
Elementwise tanh.

**toarray**(*order=None*, *out=None*)
Returns a dense matrix representing the same value.

**tobsr**(*blocksize=None*, *copy=False*)
Convert this matrix to Block Sparse Row format.

**tocoo**(*copy=False*)
Convert this matrix to COOrdinate format.

**tocsc**(*copy=False*)
Converts the matrix to Compressed Sparse Column format.

**Parameters** **copy** (*bool*) – If `False`, it shares data arrays as much as possible. Actually this option is ignored because all arrays in a matrix cannot be shared in dia to csc conversion.

**Returns** Converted matrix.

**Return type** *cupyx.scipy.sparse.csc_matrix*

**tocsr**(*copy=False*)
Converts the matrix to Compressed Sparse Row format.

**Parameters** **copy** (*bool*) – If `False`, it shares data arrays as much as possible. Actually this option is ignored because all arrays in a matrix cannot be shared in dia to csr conversion.

**Returns** Converted matrix.

**Return type** *cupyx.scipy.sparse.csc_matrix*

**todense**(*order=None*, *out=None*)
Return a dense matrix representation of this matrix.

**todia**(*copy=False*)
Convert this matrix to sparse DIAgonal format.

**todok**(*copy=False*)
Convert this matrix to Dictionary Of Keys format.

**tolil**(*copy=False*)
Convert this matrix to LInked List format.

**transpose**(*axes=None*, *copy=False*)
Reverses the dimensions of the sparse matrix.

**trunc**()
Elementwise trunc.

**__eq__**(*other*)
Return self==value.

**__ne__**(*other*)
Return self!=value.

**__lt__**(*other*)
Return self<value.

**__le__**(*other*)
Return self<=value.

**__gt__**(*other*)
Return self>value.

**__ge__**(*other*)
Return self>=value.

**__nonzero__**()

**__bool__**()

#### Attributes

**A**
Dense ndarray representation of this matrix.

This property is equivalent to *toarray()* method.

**H**

**T**

**device**
CUDA device on which this array resides.

**dtype**
Data type of the matrix.

**format = 'dia'**

**ndim**

**nnz**

**shape**

**size**

### cupyx.scipy.sparse.spmatrix

**class** cupyx.scipy.sparse.**spmatrix**(*maxprint=50*)
Base class of all sparse matrixes.

See scipy.sparse.spmatrix

#### Methods

**__len__**()

**__iter__**()

**asformat**(*format*)
Return this matrix in a given sparse format.

**Parameters** **format** (*str or None*) – Format you need.

**asfptype**()
Upcasts matrix to a floating point format.

When the matrix has floating point type, the method returns itself. Otherwise it makes a copy with floating point type and the same format.

**Returns** A matrix with float type.

**Return type** *cupyx.scipy.sparse.spmatrix*

**astype**(*t*)
Casts the array to given data type.

**Parameters** **t** – Type specifier.

**Returns** A copy of the array with the given type and the same format.

**Return type** *cupyx.scipy.sparse.spmatrix*

**conj**(*copy=True*)
Element-wise complex conjugation.

If the matrix is of non-complex data type and *copy* is False, this method does nothing and the data is not copied.

**Parameters** **copy** (*bool*) – If True, the result is guaranteed to not share data with self.

**Returns** The element-wise complex conjugate.

**Return type** *cupyx.scipy.sparse.spmatrix*

**conjugate**(*copy=True*)
Element-wise complex conjugation.

If the matrix is of non-complex data type and *copy* is False, this method does nothing and the data is not copied.

**Parameters** **copy** (*bool*) – If True, the result is guaranteed to not share data with self.

**Returns** The element-wise complex conjugate.

**Return type** *cupyx.scipy.sparse.spmatrix*

**copy**()
Returns a copy of this matrix.

No data/indices will be shared between the returned value and current matrix.

**count_nonzero**()
Number of non-zero entries, equivalent to

**diagonal**(*k=0*)
Returns the k-th diagonal of the matrix.

**Parameters**

- **k** (*int, optional*) – Which diagonal to get, corresponding to elements
- **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*)

**minimum**(*other*)

**multiply**(*other*)
Point-wise multiplication by another matrix

**power**(*n*, *dtype=None*)

**reshape**(*shape*, *order='C'*)
Gives a new shape to a sparse matrix without changing its data.

**set_shape**(*shape*)

**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 comuted. If it is `None`, it computes the sum of all the elements. Select from `{None, 0, 1, -2, -1}`.
- **dtype** – The type of returned matrix. If it is not specified, type of the array is used.
- **out** (`cupy.ndarray`) – Output matrix.

**Returns** Summed array.

**Return type** *cupy.ndarray*

**See also:**

`scipy.sparse.spmatrix.sum()`

**toarray**(*order=None*, *out=None*)
Return a dense ndarray representation of this matrix.

**tobsr**(*blocksize=None*, *copy=False*)
Convert this matrix to Block Sparse Row format.

**tocoo**(*copy=False*)
Convert this matrix to COOrdinate format.

**tocsc**(*copy=False*)
Convert this matrix to Compressed Sparse Column format.

**tocsr**(*copy=False*)
Convert this matrix to Compressed Sparse Row format.

**todense**(*order=None*, *out=None*)
Return a dense matrix representation of this matrix.

**todia**(*copy=False*)
Convert this matrix to sparse DIAgonal format.

**todok**(*copy=False*)
Convert this matrix to Dictionary Of Keys format.

**tolil**(*copy=False*)
Convert this matrix to LInked List format.

**transpose**(*axes=None*, *copy=False*)
Reverses the dimensions of the sparse matrix.

**__eq__**(*other*)
Return self==value.

**__ne__**(*other*)
Return self!=value.

**__lt__**(*other*)
Return self<value.

**__le__**(*other*)
Return self<=value.

**__gt__**(*other*)
Return self>value.

**__ge__**(*other*)
Return self>=value.

**__nonzero__**()

**__bool__**()

### Attributes

**A**
Dense ndarray representation of this matrix.

This property is equivalent to *toarray()* method.

**H**

**T**

**device**
CUDA device on which this array resides.

**ndim**

**nnz**

**shape**

**size**

## Functions

### Building sparse matrices

| | |
|---|---|
| *cupyx.scipy.sparse.diags* | Construct a sparse matrix from diagonals. |
| *cupyx.scipy.sparse.eye* | Creates a sparse matrix with ones on diagonal. |

Continued on next page

Table 73 – continued from previous page

| | |
|---|---|
| *cupyx.scipy.sparse.identity* | Creates an identity matrix in sparse format. |
| *cupyx.scipy.sparse.spdiags* | Creates a sparse matrix from diagonals. |
| *cupyx.scipy.sparse.rand* | Generates a random sparse matrix. |
| *cupyx.scipy.sparse.random* | Generates a random sparse matrix. |

### cupyx.scipy.sparse.diags

cupyx.scipy.sparse.**diags**(*diagonals*, *offsets=0*, *shape=None*, *format=None*, *dtype=None*)

Construct a sparse matrix from diagonals.

**Parameters**

- **diagonals** (*sequence of array_like*) – Sequence of arrays containing the matrix diagonals, corresponding to *offsets*.
- **offsets** (*sequence of int or an int*) –

  **Diagonals to set:**

  - k = 0 the main diagonal (default)
  - k > 0 the k-th upper diagonal
  - k < 0 the k-th lower diagonal
- **shape** (*tuple of int*) – Shape of the result. If omitted, a square matrix large enough to contain the diagonals is returned.
- **format** (*{"dia", "csr", "csc", "lil", ..}*) – Matrix format of the result. By default (format=None) an appropriate sparse matrix format is returned. This choice is subject to change.
- **dtype** (*dtype*) – Data type of the matrix.

**Returns** Generated matrix.

**Return type** *cupyx.scipy.sparse.spmatrix*

#### Notes

This function differs from *spdiags* in the way it handles off-diagonals.

The result from *diags* is the sparse equivalent of:

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

Repeated diagonal offsets are disallowed.

### cupyx.scipy.sparse.eye

cupyx.scipy.sparse.**eye**(*m*, *n=None*, *k=0*, *dtype='d'*, *format=None*)

Creates a sparse matrix with ones on diagonal.

**Parameters**

- **m** (*int*) – Number of rows.

- **n** (*int or None*) – Number of columns. If it is None, it makes a square matrix.
- **k** (*int*) – Diagonal to place ones on.
- **dtype** – Type of a matrix to create.
- **format** (*str or None*) – Format of the result, e.g. format="csr".

**Returns** Created sparse matrix.

**Return type** *cupyx.scipy.sparse.spmatrix*

**See also:**

scipy.sparse.eye()

### cupyx.scipy.sparse.identity

cupyx.scipy.sparse.**identity**(*n*, *dtype='d'*, *format=None*)

Creates an identity matrix in sparse format.

**Note:** Currently it only supports csr, csc and coo formats.

**Parameters**

- **n** (*int*) – Number of rows and columns.
- **dtype** – Type of a matrix to create.
- **format** (*str or None*) – Format of the result, e.g. format="csr".

**Returns** Created identity matrix.

**Return type** *cupyx.scipy.sparse.spmatrix*

**See also:**

scipy.sparse.identity()

### cupyx.scipy.sparse.spdiags

cupyx.scipy.sparse.**spdiags**(*data*, *diags*, *m*, *n*, *format=None*)

Creates a sparse matrix from diagonals.

**Parameters**

- **data** (cupy.ndarray) – Matrix diagonals stored row-wise.
- **diags** (cupy.ndarray) – Diagonals to set.
- **m** (*int*) – Number of rows.
- **n** (*int*) – Number of cols.
- **format** (*str or None*) – Sparse format, e.g. format="csr".

**Returns** Created sparse matrix.

**Return type** *cupyx.scipy.sparse.spmatrix*

**See also:**

`scipy.sparse.spdiags()`

### cupyx.scipy.sparse.rand

`cupyx.scipy.sparse.`**`rand`**(*m*, *n*, *density=0.01*, *format='coo'*, *dtype=None*, *random_state=None*)

Generates a random sparse matrix.

See *cupyx.scipy.sparse.random()* for detail.

**Parameters**

- **m** (*int*) – Number of rows.
- **n** (*int*) – Number of cols.
- **density** (*float*) – Ratio of non-zero entries.
- **format** (*str*) – Matrix format.
- **dtype** (*dtype*) – Type of the returned matrix values.
- **random_state** (`cupy.random.RandomState` *or* *int*) – State of random number generator. If an integer is given, the method makes a new state for random number generator and uses it. If it is not given, the default state is used. This state is used to generate random indexes for nonzero entries.

**Returns** Generated matrix.

**Return type** *cupyx.scipy.sparse.spmatrix*

**See also:**

`scipy.sparse.rand()`

**See also:**

*cupyx.scipy.sparse.random()*

### cupyx.scipy.sparse.random

`cupyx.scipy.sparse.`**`random`**(*m*, *n*, *density=0.01*, *format='coo'*, *dtype=None*, *random_state=None*, *data_rvs=None*)

Generates a random sparse matrix.

This function generates a random sparse matrix. First it selects non-zero elements with given density `density` from `(m, n)` elements. So the number of non-zero elements `k` is `k = m * n * density`. Value of each element is selected with `data_rvs` function.

**Parameters**

- **m** (*int*) – Number of rows.
- **n** (*int*) – Number of cols.
- **density** (*float*) – Ratio of non-zero entries.
- **format** (*str*) – Matrix format.
- **dtype** (*dtype*) – Type of the returned matrix values.

- **random_state** (*cupy.random.RandomState or int*) – State of random number generator. If an integer is given, the method makes a new state for random number generator and uses it. If it is not given, the default state is used. This state is used to generate random indexes for nonzero entries.
- **data_rvs** (*callable*) – A function to generate data for a random matrix. If it is not given, *random_state.rand* is used.

**Returns** Generated matrix.

**Return type** *cupyx.scipy.sparse.spmatrix*

**See also:**

`scipy.sparse.random()`

### Identifying sparse matrices

| | |
|---|---|
| *cupyx.scipy.sparse.issparse* | Checks if a given matrix is a sparse matrix. |
| *cupyx.scipy.sparse.isspmatrix* | Checks if a given matrix is a sparse matrix. |
| *cupyx.scipy.sparse.isspmatrix_csc* | Checks if a given matrix is of CSC format. |
| *cupyx.scipy.sparse.isspmatrix_csr* | Checks if a given matrix is of CSR format. |
| *cupyx.scipy.sparse.isspmatrix_coo* | Checks if a given matrix is of COO format. |
| *cupyx.scipy.sparse.isspmatrix_dia* | Checks if a given matrix is of DIA format. |

### cupyx.scipy.sparse.issparse

cupyx.scipy.sparse.**issparse**(*x*)

Checks if a given matrix is a sparse matrix.

**Returns**

**Returns if x is *cupyx.scipy.sparse.spmatrix* that is** a base class of all sparse matrix classes.

**Return type** bool

### cupyx.scipy.sparse.isspmatrix

cupyx.scipy.sparse.**isspmatrix**(*x*)

Checks if a given matrix is a sparse matrix.

**Returns**

**Returns if x is *cupyx.scipy.sparse.spmatrix* that is** a base class of all sparse matrix classes.

**Return type** bool

### cupyx.scipy.sparse.isspmatrix_csc

cupyx.scipy.sparse.**isspmatrix_csc**(*x*)

Checks if a given matrix is of CSC format.

**Returns** Returns if x is *cupyx.scipy.sparse.csc_matrix*.

**Return type** bool

### cupyx.scipy.sparse.isspmatrix_csr

cupyx.scipy.sparse.**isspmatrix_csr**(*x*)
Checks if a given matrix is of CSR format.

**Returns** Returns if x is *cupyx.scipy.sparse.csr_matrix*.

**Return type** bool

### cupyx.scipy.sparse.isspmatrix_coo

cupyx.scipy.sparse.**isspmatrix_coo**(*x*)
Checks if a given matrix is of COO format.

**Returns** Returns if x is *cupyx.scipy.sparse.coo_matrix*.

**Return type** bool

### cupyx.scipy.sparse.isspmatrix_dia

cupyx.scipy.sparse.**isspmatrix_dia**(*x*)
Checks if a given matrix is of DIA format.

**Returns** Returns if x is *cupyx.scipy.sparse.dia_matrix*.

**Return type** bool

## Linear Algebra

| | |
|---|---|
| *cupyx.scipy.sparse.linalg.lsqr* | Solves linear system with QR decomposition. |

### cupyx.scipy.sparse.linalg.lsqr

cupyx.scipy.sparse.linalg.**lsqr**(*A*, *b*)
Solves linear system with QR decomposition.

Find the solution to a large, sparse, linear system of equations. The function solves `Ax = b`. Given two-dimensional matrix `A` is decomposed into `Q * R`.

**Parameters**

- **A** (*cupy.ndarray or cupyx.scipy.sparse.csr_matrix*) – The input matrix with dimension `(N, N)`
- **b** (*cupy.ndarray*) – Right-hand side vector.

**Returns** Its length must be ten. It has same type elements as SciPy. Only the first element, the solution vector `x`, is available and other elements are expressed as `None` because the implementation of cuSOLVER is different from the one of SciPy. You can easily calculate the fourth element by `norm(b - Ax)` and the ninth element by `norm(x)`.

**Return type** tuple

**See also:**

`scipy.sparse.linalg.lsqr()`

### 3.4.5 Special Functions

#### Bessel Functions

| | |
|---|---|
| *cupyx.scipy.special.j0* | Bessel function of the first kind of order 0. |
| *cupyx.scipy.special.j1* | Bessel function of the first kind of order 1. |
| *cupyx.scipy.special.y0* | Bessel function of the second kind of order 0. |
| *cupyx.scipy.special.y1* | Bessel function of the second kind of order 1. |
| *cupyx.scipy.special.i0* | Modified Bessel function of order 0. |
| *cupyx.scipy.special.i1* | Modified Bessel function of order 1. |

##### 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()

### Gamma and Related Functions

| | |
|---|---|
| *cupyx.scipy.special.gamma* | Gamma function. |
| *cupyx.scipy.special.gammaln* | Logarithm of the absolute value of the Gamma function. |
| *cupyx.scipy.special.polygamma* | Polygamma function n. |
| *cupyx.scipy.special.digamma* | The digamma function. |

### cupyx.scipy.special.gamma

cupyx.scipy.special.**gamma = <ufunc 'cupyx_scipy_gamma'>**
Gamma function.

**Parameters** **z** (cupy.ndarray) – The input of gamma function.

**Returns** Computed value of gamma function.

**Return type** *cupy.ndarray*

**See also:**

scipy.special.gamma

### cupyx.scipy.special.gammaln

cupyx.scipy.special.**gammaln = <ufunc 'cupyx_scipy_gammaln'>**
Logarithm of the absolute value of the Gamma function.

**Parameters**

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

**Returns** Values of gammaln at x.

**Return type** *cupy.ndarray*

**See also:**

scipy.special.gammaln

### cupyx.scipy.special.polygamma

cupyx.scipy.special.**polygamma**(*n*, *x*)

Polygamma function n.

**Parameters**

- **n** (cupy.ndarray) – The order of the derivative of *psi*.
- **x** (cupy.ndarray) – Where to evaluate the polygamma function.

**Returns** The result.

**Return type** *cupy.ndarray*

**See also:**

scipy.special.polygamma

### cupyx.scipy.special.digamma

cupyx.scipy.special.**digamma = <ufunc 'cupyx_scipy_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

| | |
|---|---|
| *cupyx.scipy.special.ndtr* | Cumulative distribution function of normal distribution. |

### 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

| | |
|---|---|
| *cupyx.scipy.special.erf* | Error function. |
| *cupyx.scipy.special.erfc* | Complementary error function. |
| *cupyx.scipy.special.erfcx* | Scaled complementary error function. |

Continued on next page

Table 79 – continued from previous page

| | |
|---|---|
| *cupyx.scipy.special.erfinv* | Inverse function of error function. |
| *cupyx.scipy.special.erfcinv* | Inverse function of complementary error function. |

### 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()

### cupyx.scipy.special.erfcinv

cupyx.scipy.special.**erfcinv = <ufunc 'cupyx_scipy_erfcinv'>**
Inverse function of complementary error function.

**See also:**

scipy.special.erfcinv()

### Other Special Functions

| | |
|---|---|
| *cupyx.scipy.special.zeta* | Hurwitz zeta function. |

#### cupyx.scipy.special.zeta

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

Hurwitz zeta function.

**Parameters**

- **x** (cupy.ndarray) – Input data, must be real.
- **q** (cupy.ndarray) – Input data, must be real.

**Returns** Values of zeta(x, q).

**Return type** *cupy.ndarray*

**See also:**

scipy.special.zeta

## 3.5 NumPy-CuPy Generic Code Support

| | |
|---|---|
| *cupy.get_array_module* | Returns the array module for arguments. |
| *cupyx.scipy.get_array_module* | Returns the array module for arguments. |

## 3.6 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.

### 3.6.1 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:

```
import cupy
import numpy
```

(continues on next page)

(continued from previous page)

```
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
```

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

### 3.6.2 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).

```
# 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%"
```

(continues on next page)

(continued from previous page)

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

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

```
import cupy

mempool = cupy.get_default_memory_pool()

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

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

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

### 3.6.3 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.

```
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)
```

## 3.7 Low-Level CUDA Support

### 3.7.1 Device management

| | |
|---|---|
| *cupy.cuda.Device* | Object that represents a CUDA device. |

#### cupy.cuda.Device

**class** cupy.cuda.**Device**(*device=None*)

Object that represents a CUDA device.

This class provides some basic manipulations on CUDA devices.

It supports the context protocol. For example, the following code is an example of temporarily switching the

current device:

```
with Device(0):
    do_something_on_device_0()
```

After the *with* statement gets done, the current device is reset to the original one.

**Parameters** **device** (*int or cupy.cuda.Device*) – Index of the device to manipulate. Be careful that the device ID (a.k.a. GPU ID) is zero origin. If it is a Device object, then its ID is used. The current device is selected by default.

**Variables** ***id*** (*int*) – ID of this device.

### Methods

**__enter__**(*self*)

**__exit__**(*self*, **args*)

**from_pci_bus_id**(*type cls*, *pci_bus_id*)
Returns a new device instance based on a PCI Bus ID

**Parameters** **pci_bus_id** (*str*) – The string for a device in the following format [domain]:[bus]:[device].[function] where domain, bus, device, and function are all hexadecimal values.

**Returns** An instance of the Device class that has the PCI Bus ID as given by the argument pci_bus_id.

**Return type** device (*Device*)

**synchronize**(*self*)
Synchronizes the current thread to the device.

**use**(*self*)
Makes this device current.

If you want to switch a device temporarily, use the *with* statement.

### Attributes

**attributes**
A dictionary of device attributes.

**Returns** Dictionary of attribute values with the names as keys. The string *cudaDevAttr* has been trimmed from the names. For example, the attribute corresponding to the enumerated value *cudaDevAttrMaxThreadsPerBlock* will have key *MaxThreadsPerBlock*.

**Return type** attributes (dict)

**compute_capability**
Compute capability of this device.

The capability is represented by a string containing the major index and the minor index. For example, compute capability 3.5 is represented by the string '35'.

**cublas_handle**
The cuBLAS handle for this device.

The same handle is used for the same device even if the Device instance itself is different.

**cusolver_handle**
The cuSOLVER handle for this device.

The same handle is used for the same device even if the Device instance itself is different.

**cusolver_sp_handle**
The cuSOLVER Sphandle for this device.

The same handle is used for the same device even if the Device instance itself is different.

**cusparse_handle**
The cuSPARSE handle for this device.

The same handle is used for the same device even if the Device instance itself is different.

**id**
'int'

**Type** id

**mem_info**
The device memory info.

**Returns** The amount of free memory, in bytes. total: The total amount of memory, in bytes.

**Return type** free

**pci_bus_id**
A string of the PCI Bus ID

**Returns** Returned identifier string for the device in the following format [domain]:[bus]:[device].[function] where domain, bus, device, and function are all hexadecimal values.

**Return type** pci_bus_id (str)

### 3.7.2 Memory management

| | |
|---|---|
| *cupy.get_default_memory_pool* | Returns CuPy default memory pool for GPU memory. |
| *cupy.get_default_pinned_memory_pool* | Returns CuPy default memory pool for pinned memory. |
| *cupy.cuda.Memory* | Memory allocation on a CUDA device. |
| *cupy.cuda.UnownedMemory* | CUDA memory that is not owned by CuPy. |
| *cupy.cuda.PinnedMemory* | Pinned memory allocation on host. |
| *cupy.cuda.MemoryPointer* | Pointer to a point on a device memory. |
| *cupy.cuda.PinnedMemoryPointer* | Pointer of a pinned memory. |
| *cupy.cuda.alloc* | Calls the current allocator. |
| *cupy.cuda.alloc_pinned_memory* | Calls the current allocator. |
| *cupy.cuda.get_allocator* | Returns the current allocator for GPU memory. |
| *cupy.cuda.set_allocator* | Sets the current allocator for GPU memory. |
| *cupy.cuda.using_allocator* | Sets a thread-local allocator for GPU memory inside |
| *cupy.cuda.set_pinned_memory_allocator* | Sets the current allocator for the pinned memory. |
| *cupy.cuda.MemoryPool* | Memory pool for all GPU devices on the host. |
| *cupy.cuda.PinnedMemoryPool* | Memory pool for pinned memory on the host. |

#### 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.cuda.PinnedMemory

**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).

---

**Note:** This wraps an internal version of this function to provide a *contextmanager* as *contextmanger* decoration doesn't behave well in Cython.

---

### 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=_malloc*)
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

**malloc** (*self*, *size_t size*) → MemoryPointer
Allocates the memory, from the pool if possible.

This method can be used as a CuPy memory allocator. The simplest way to use a memory pool as the default allocator is the following code:

```
set_allocator(MemoryPool().malloc)
```

Also, the way to use a memory pool of Managed memory (Unified memory) as the default allocator is the following code:

```
set_allocator(MemoryPool(malloc_managed).malloc)
```

**Parameters** **size** (*int*) – Size of the memory buffer to allocate in bytes.

**Returns** Pointer to the allocated buffer.

**Return type** *MemoryPointer*

**n_free_blocks**(*self*) → size_t
Counts the total number of free blocks.

**Returns** The total number of free blocks.

**Return type** int

**set_limit**(*self*, *size=None*, *fraction=None*)
Sets the upper limit of memory allocation of the current device.

When *fraction* is specified, its value will become a fraction of the amount of GPU memory that is available for allocation. For example, if you have a GPU with 2 GiB memory, you can either use `set_limit(fraction=0.5)` or `set_limit(size=1024**3)` to limit the memory size to 1 GiB.

`size` and `fraction` cannot be specified at one time. If both of them are **not** specified or `0` is specified, the limit will be disabled.

---

**Note:** You can also set the limit by using `CUPY_GPU_MEMORY_LIMIT` environment variable. See *Environment variables* for the details. The limit set by this method supersedes the value specified in the environment variable.

Also note that this method only changes the limit for the current device, whereas the environment variable sets the default limit for all devices.

---

**Parameters**

- **size** (*int*) – Limit size in bytes.
- **fraction** (*float*) – Fraction in the range of `[0, 1]`.

**total_bytes**(*self*) → size_t
Gets the total number of bytes acquired in the pool.

**Returns** The total number of bytes acquired in the pool.

**Return type** int

**used_bytes**(*self*) → size_t
Gets the total number of bytes used.

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

**Return type** int

### 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

### 3.7.3 Memory hook

| | |
|---|---|
| *cupy.cuda.MemoryHook* | Base class of hooks for Memory allocations. |
| *cupy.cuda.memory_hooks.DebugPrintHook* | Memory hook that prints debug information. |
| *cupy.cuda.memory_hooks.LineProfileHook* | Code line CuPy memory profiler. |

#### cupy.cuda.MemoryHook

**class** cupy.cuda.**MemoryHook**
Base class of hooks for Memory allocations.

*MemoryHook* is an callback object. Registered memory hooks are invoked before and after memory is allocated from GPU device, and memory is retrieved from memory pool, and memory is released to memory pool.

Memory hooks that derive *MemoryHook* are required to implement six methods: *alloc_preprocess()*, *alloc_postprocess()*, *malloc_preprocess()*, *malloc_postprocess()*, *free_preprocess()*, and *free_postprocess()*, By default, these methods do nothing.

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

Likewise, *malloc_preprocess()* (resp. *malloc_postprocess()*) of all memory hooks registered are called before (resp. after) memory is retrieved from memory pool.

Below is a pseudo code to descirbe how malloc and hooks work. Please note that *alloc_preprocess()* and *alloc_postprocess()* are not invoked if a cached free chunk is found:

```
def malloc(size):
    Call malloc_preprocess of all memory hooks
    Try to find a cached free chunk from memory pool
    if chunk is not found:
        Call alloc_preprocess for all memory hooks
        Invoke actual memory allocation to get a new chunk
        Call alloc_postprocess for all memory hooks
    Call malloc_postprocess for all memory hooks
```

Moreover, *free_preprocess()* (resp. *free_postprocess()*) of all memory hooks registered are called before (resp. after) memory is released to memory pool.

Below is a pseudo code to descirbe how free and hooks work:

```
def free(ptr):
    Call free_preprocess of all memory hooks
    Push a memory chunk of a given pointer back to memory pool
    Call free_postprocess for all memory hooks
```

To register a memory hook, use `with` statement. Memory hooks are registered to all method calls within `with` statement and are unregistered at the end of `with` statement.

---

**Note:** CuPy stores the dictionary of registered function hooks as a thread local object. So, memory hooks registered can be different depending on threads.

---

### Methods

**__enter__**(*self*)

**__exit__**(*self*, *_)

### 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

**name = 'MemoryHook'**

### cupy.cuda.memory_hooks.DebugPrintHook

**class** cupy.cuda.memory_hooks.**DebugPrintHook**(*file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>, flush=True*)

Memory hook that prints debug information.

This memory hook outputs the debug information of input arguments of `malloc` and `free` methods involved in the hooked functions at postprocessing time (that is, just after each method is called).

#### Example

The basic usage is to use it with `with` statement.

Code example:

```
>>> import cupy
>>> from cupy.cuda import memory_hooks
>>>
>>> cupy.cuda.set_allocator(cupy.cuda.MemoryPool().malloc)
>>> with memory_hooks.DebugPrintHook():
...     x = cupy.array([1, 2, 3])
...     del x
```

Output example:

```
{"hook":"alloc","device_id":0,"mem_size":512,"mem_ptr":150496608256}
{"hook":"malloc","device_id":0,"size":24,"mem_size":512,"mem_ptr":150496608256,
↪"pmem_id":"0x7f39200c5278"}
{"hook":"free","device_id":0,"mem_size":512,"mem_ptr":150496608256,"pmem_id":
↪"0x7f39200c5278"}
```

where the output format is JSONL (JSON Lines) and `hook` is the name of hook point, and `device_id` is the CUDA Device ID, and `size` is the requested memory size to allocate, and `mem_size` is the rounded memory size to be allocated, and `mem_ptr` is the memory pointer, and `pmem_id` is the pooled memory object ID.

**Variables**

- **`file`** – Output file_like object that redirect to.
- **`flush`** – If `True`, this hook forcibly flushes the text stream at the end of print. The default is `True`.

### Methods

**`__enter__`**(*self*)

**`__exit__`**(*self*, *_)

**`alloc_postprocess`**(*self*, ***kwargs*)
Callback function invoked after allocating memory from GPU device.

**Keyword Arguments**

- **`device_id`** (*int*) – CUDA device ID
- **`mem_size`** (*int*) – Rounded memory bytesize allocated
- **`mem_ptr`** (*int*) – Obtained memory pointer. 0 if an error occurred in allocation.

**`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

**name = 'DebugPrintHook'**

### cupy.cuda.memory_hooks.LineProfileHook

**class** cupy.cuda.memory_hooks.**LineProfileHook**(*max_depth=0*)
Code line CuPy memory profiler.

This profiler shows line-by-line GPU memory consumption using traceback module. But, note that it can trace only CPython level, no Cython level. ref. https://github.com/cython/cython/issues/1755

#### Example

Code example:

```
from cupy.cuda import memory_hooks
hook = memory_hooks.LineProfileHook()
with hook:
    # some CuPy codes
hook.print_report()
```

Output example:

```
_root (4.00KB, 4.00KB)
  lib/python3.6/unittest/__main__.py:18:<module> (4.00KB, 4.00KB)
    lib/python3.6/unittest/main.py:255:runTests (4.00KB, 4.00KB)
      tests/cupy_tests/test.py:37:test (1.00KB, 1.00KB)
```

(continues on next page)

(continued from previous page)

```
    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'**

### 3.7.4 Streams and events

| | |
|---|---|
| *cupy.cuda.Stream* | CUDA stream. |
| *cupy.cuda.get_current_stream* | Gets current CUDA stream. |
| *cupy.cuda.Event* | CUDA event, a synchronization point of CUDA streams. |
| *cupy.cuda.get_elapsed_time* | Gets the elapsed time between two events. |

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

**Parameters**

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

**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.get_current_stream

`cupy.cuda.`**get_current_stream**`()`
Gets current CUDA stream.

**Returns** The current CUDA stream.

**Return type** *cupy.cuda.Stream*

### cupy.cuda.Event

**class** cupy.cuda.**Event**
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

## 3.7.5 Texture memory

| | |
|---|---|
| *cupy.cuda.texture.ChannelFormatDescriptor* | A class that holds the channel format description. |
| *cupy.cuda.texture.CUDAarray* | Allocate a CUDA array (*cudaArray_t*) that can be used as texture memory. |
| *cupy.cuda.texture.ResourceDescriptor* | A class that holds the resource description. |
| *cupy.cuda.texture.TextureDescriptor* | A class that holds the texture description. |
| *cupy.cuda.texture.TextureObject* | A class that holds a texture object. |
| *cupy.cuda.texture.TextureReference* | A class that holds a texture reference. |

### cupy.cuda.texture.ChannelFormatDescriptor

**class** cupy.cuda.texture.**ChannelFormatDescriptor**(*int x*, *int y*, *int z*, *int w*, *int f*)

A class that holds the channel format description. Equivalent to `cudaChannelFormatDesc`.

**Parameters**

- **x** (*int*) – the number of bits for the x channel.
- **y** (*int*) – the number of bits for the y channel.
- **z** (*int*) – the number of bits for the z channel.
- **w** (*int*) – the number of bits for the w channel.
- **f** (*int*) – the channel format. Use one of the values in `cudaChannelFormat*`, such as `cupy.cuda.runtime.cudaChannelFormatKindFloat`.

**See also:**

cudaCreateChannelDesc()

#### Methods

**get_channel_format**(*self*)

Returns a dict containing the input.

#### Attributes

**ptr**

### cupy.cuda.texture.CUDAarray

**class** cupy.cuda.texture.**CUDAarray**(*ChannelFormatDescriptor desc*, *size_t width*, *size_t height=0*, *size_t depth=0*, *unsigned int flags=0*)

Allocate a CUDA array (*cudaArray_t*) that can be used as texture memory. Depending on the input, either 1D, 2D, or 3D CUDA array is returned.

**Parameters**

- **desc** (*ChannelFormatDescriptor*) – an instance of *ChannelFormatDescriptor*.
- **width** (*int*) – the width (in elements) of the array.
- **height** (*int, optional*) – the height (in elements) of the array.
- **depth** (*int, optional*) – the depth (in elements) of the array.

- **flags** (*int, optional*) – the flag for extensions. Use one of the values in `cudaArray*`, such as `cupy.cuda.runtime.cudaArrayDefault`.

**Warning:** The memory allocation of *CUDAarray* is done outside of CuPy's memory management (enabled by default) due to CUDA's limitation. Users of *CUDAarray* should be cautious about any out-of-memory possibilities.

**See also:**

cudaMalloc3DArray()

### Methods

**copy_from**(*self*, *in_arr*, *stream=None*)
Copy data from device or host array to CUDA array.

**Parameters**

- **in_arr** (*cupy.ndarray or numpy.ndarray*) –
- **stream** (*cupy.cuda.Stream*) – if not `None`, an asynchronous copy is performed.

---

**Note:** For CUDA arrays with different dimensions, the requirements for the shape of the input array are given as follows:

- 1D: `(nch * width,)`
- 2D: `(height, nch * width)`
- 3D: `(depth, height, nch * width)`

where `nch` is the number of channels specified in *desc*.

---

**copy_to**(*self*, *out_arr*, *stream=None*)
Copy data from CUDA array to device or host array.

**Parameters**

- **out_arr** (*cupy.ndarray or numpy.ndarray*) –
- **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**

**height**

**ndim**

**ptr**

**width**

### cupy.cuda.texture.ResourceDescriptor

**class** cupy.cuda.texture.**ResourceDescriptor**(*int restype*, *CUDAarray cuArr=None*, *ndarray arr=None*, *ChannelFormatDescriptor chDesc=None*, *size_t sizeInBytes=0*, *size_t width=0*, *size_t height=0*, *size_t pitchInBytes=0*)

A class that holds the resource description. Equivalent to `cudaResourceDesc`.

**Parameters**

- **restype** (*int*) – the resource type. Use one of the values in `cudaResourceType*`, such as `cupy.cuda.runtime.cudaResourceTypeArray`.
- **cuArr** (`CUDAarray`, *optional*) – An instance of *CUDAarray*, required if `restype` is set to `cupy.cuda.runtime.cudaResourceTypeArray`.
- **arr** (`cupy.ndarray`, *optional*) – An instance of *ndarray*, required if `restype` is set to `cupy.cuda.runtime.cudaResourceTypeLinear` or `cupy.cuda.runtime.cudaResourceTypePitch2D`.
- **chDesc** (`ChannelFormatDescriptor`, *optional*) – an instance of *ChannelFormatDescriptor*, required if `restype` is set to `cupy.cuda.runtime.cudaResourceTypeLinear` or `cupy.cuda.runtime.cudaResourceTypePitch2D`.
- **sizeInBytes** (*int*, *optional*) – total bytes in the linear memory, required if `restype` is set to `cupy.cuda.runtime.cudaResourceTypeLinear`.
- **width** (*int*, *optional*) – the width (in elements) of the 2D array, required if `restype` is set to `cupy.cuda.runtime.cudaResourceTypePitch2D`.
- **height** (*int*, *optional*) – the height (in elements) of the 2D array, required if `restype` is set to `cupy.cuda.runtime.cudaResourceTypePitch2D`.
- **pitchInBytes** (*int*, *optional*) – the number of bytes per pitch-aligned row, required if `restype` is set to `cupy.cuda.runtime.cudaResourceTypePitch2D`.

---

**Note:** A texture backed by *mipmap* arrays is currently not supported in CuPy.

---

**See also:**

cudaCreateTextureObject()

### Methods

**get_resource_desc**(*self*)
Returns a dict containing the input.

### Attributes

**arr**

**chDesc**

**cuArr**

**ptr**

## cupy.cuda.texture.TextureDescriptor

**class** cupy.cuda.texture.**TextureDescriptor**(*addressModes=None*, *int filterMode=0*, *int readMode=0*, *sRGB=None*, *borderColors=None*, *normalizedCoords=None*, *maxAnisotropy=None*)

A class that holds the texture description. Equivalent to `cudaTextureDesc`.

**Parameters**

- **addressModes** (*tuple or list*) – an iterable with length up to 3, each element is one of the values in `cudaAddressMode*`, such as `cupy.cuda.runtime.cudaAddressModeWrap`.
- **filterMode** (*int*) – the filter mode. Use one of the values in `cudaFilterMode*`, such as `cupy.cuda.runtime.cudaFilterModePoint`.
- **readMode** (*int*) – the read mode. Use one of the values in `cudaReadMode*`, such as `cupy.cuda.runtime.cudaReadModeElementType`.
- **normalizedCoords** (*int*) – whether coordinates are normalized or not.
- **sRGB** (*int, optional*) –
- **borderColors** (*tuple or list, optional*) – an iterable with length up to 4.
- **maxAnisotropy** (*int, optional*) –

**Note:** A texture backed by *mipmap* arrays is currently not supported in CuPy.

**See also:**

cudaCreateTextureObject()

### Methods

**get_texture_desc**(*self*)
Returns a dict containing the input.

#### Attributes

**ptr**

### cupy.cuda.texture.TextureObject

**class** cupy.cuda.texture.**TextureObject**(*ResourceDescriptor ResDesc*, *TextureDescriptor TexDesc*)

A class that holds a texture object. Equivalent to `cudaTextureObject_t`. The returned *TextureObject* instance can be passed as a argument when launching *RawKernel*.

**Parameters**

- **ResDesc** (ResourceDescriptor) – an intance of the resource descriptor.
- **TexDesc** (TextureDescriptor) – an instance of the texture descriptor.

**See also:**

cudaCreateTextureObject()

#### Methods

#### Attributes

**ResDesc**

**TexDesc**

**ptr**

### cupy.cuda.texture.TextureReference

**class** cupy.cuda.texture.**TextureReference**(*intptr_t texref*, *ResourceDescriptor ResDesc*, *TextureDescriptor TexDesc*)

A class that holds a texture reference. Equivalent to `CUtexref` (the driver API is used under the hood).

**Parameters**

- **texref** (*intptr_t*) – a handle to the texture reference declared in the CUDA source code. This can be obtained by calling *get_texref()*.
- **ResDesc** (ResourceDescriptor) – an intance of the resource descriptor.
- **TexDesc** (TextureDescriptor) – an instance of the texture descriptor.

**Warning:** As of CUDA Toolkit v10.1, the Texture Reference API (in both driver and runtime) is marked as deprecated. To help transition to the new Texture Object API, this class mimics the usage of *TextureObject*. Users who have legacy CUDA codes that use texture references should consider migration to the new API.

This CuPy interface is subject to removal once the offcial NVIDIA support is dropped in the future.

**See also:**

*TextureObject*, cudaCreateTextureObject()

#### Methods

#### Attributes

**ResDesc**

**TexDesc**

**texref**

### 3.7.6 Profiler

| | |
|---|---|
| *cupy.cuda.profile* | Enable CUDA profiling during with statement. |
| *cupy.cuda.profiler.initialize* | Initialize the CUDA profiler. |
| *cupy.cuda.profiler.start* | Enable profiling. |
| *cupy.cuda.profiler.stop* | Disable profiling. |
| *cupy.cuda.nvtx.Mark* | Marks an instantaneous event (marker) in the application. |
| *cupy.cuda.nvtx.MarkC* | Marks an instantaneous event (marker) in the application. |
| *cupy.cuda.nvtx.RangePush* | Starts a nested range. |
| *cupy.cuda.nvtx.RangePushC* | Starts a nested range. |
| *cupy.cuda.nvtx.RangePop* | Ends a nested range. |

#### cupy.cuda.profile

cupy.cuda.**profile**()
Enable CUDA profiling during with statement.

This function enables profiling on entering a with statement, and disables profiling on leaving the statement.

```
>>> with cupy.cuda.profile():
...     # do something you want to measure
...     pass
```

#### 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.

Markes 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.

Markes are used to describe events at a specific time during execution of the application.

**Parameters**

- **message** (*str*) – Name of a marker.
- **color** (*uint32*) – Color code for a marker.

### cupy.cuda.nvtx.RangePush

cupy.cuda.nvtx.**RangePush**(*message*, *int id_color=-1*)
Starts a nested range.

Ranges are used to describe events over a time span during execution of the application. The duration of a range is defined by the corresponding pair of `RangePush*()` to `RangePop()` calls.

**Parameters**

- **message** (*str*) – Name of a range.
- **id_color** (*int*) – ID of color for a range.

#### cupy.cuda.nvtx.RangePushC

cupy.cuda.nvtx.**RangePushC**(*message*, *uint32_t color=0*)
Starts a nested range.

Ranges are used to describe events over a time span during execution of the application. The duration of a range is defined by the corresponding pair of `RangePush*()` to `RangePop()` calls.

**Parameters**

- **message** (*str*) – Name of a range.
- **color** (*uint32*) – ARGB color for a range.

#### cupy.cuda.nvtx.RangePop

cupy.cuda.nvtx.**RangePop**()
Ends a nested range.

Ranges are used to describe events over a time span during execution of the application. The duration of a range is defined by the corresponding pair of `RangePush*()` to `RangePop()` calls.

### 3.7.7 NCCL

| | |
|---|---|
| *cupy.cuda.nccl.NcclCommunicator* | Initialize an NCCL communicator for one device controlled by one process. |
| *cupy.cuda.nccl.get_build_version* | |
| *cupy.cuda.nccl.get_version* | Returns the runtime version of NCCL. |
| *cupy.cuda.nccl.get_unique_id* | |
| *cupy.cuda.nccl.groupStart* | Start a group of NCCL calls. |
| *cupy.cuda.nccl.groupEnd* | End a group of NCCL calls. |

#### cupy.cuda.nccl.NcclCommunicator

**class** cupy.cuda.nccl.**NcclCommunicator**(*int ndev*, *tuple commId*, *int rank*)
Initialize an NCCL communicator for one device controlled by one process.

**Parameters**

- **ndev** (*int*) – Total number of GPUs to be used.
- **commId** (*tuple*) – The unique ID returned by *get_unique_id()*.
- **rank** (*int*) – The rank of the GPU managed by the current process.

**Returns** An `NcclCommunicator` instance.

**Return type** *NcclCommunicator*

**Note:** This method is for creating an NCCL communicator in a multi-process environment, typically managed by MPI or `multiprocessing`. For controlling multiple devices by one process, use *initAll()* instead.

**See also:**

ncclCommInitRank

### Methods

**abort**(*self*)

**allGather**(*self*, *intptr_t sendbuf*, *intptr_t recvbuf*, *size_t count*, *int datatype*, *intptr_t stream*)

**allReduce**(*self*, *intptr_t sendbuf*, *intptr_t recvbuf*, *size_t count*, *int datatype*, *int op*, *intptr_t stream*)

**bcast**(*self*, *intptr_t buff*, *int count*, *int datatype*, *int root*, *intptr_t stream*)

**broadcast**(*self*, *intptr_t sendbuff*, *intptr_t recvbuff*, *int count*, *int datatype*, *int root*, *intptr_t stream*)

**check_async_error**(*self*)

**destroy**(*self*)

**device_id**(*self*)

**static initAll**(*devices*)

Initialize NCCL communicators for multiple devices in a single process.

**Parameters** **devices** (*int or list of int*) – The number of GPUs or a list of GPUs to be used. For the former case, the first `devices` GPUs will be used.

**Returns** A list of `NcclCommunicator` instances.

**Return type** list

---

**Note:** This method is for creating a group of NCCL communicators, each controlling one device, in a single process like this:

```
from cupy.cuda import nccl
# Use 3 GPUs: #0, #2, and #3
comms = nccl.NcclCommunicator.initAll([0, 2, 3])
assert len(comms) == 3
```

In a multi-process setup, use the default initializer instead.

---

**See also:**

ncclCommInitAll

**rank_id**(*self*)

**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*)

**size**(*self*)

## cupy.cuda.nccl.get_build_version

cupy.cuda.nccl.**get_build_version**()

## cupy.cuda.nccl.get_version

cupy.cuda.nccl.**get_version**()

Returns the runtime version of NCCL.

This function will return 0 when built with NCCL version earlier than 2.3.4, which does not support `ncclGetVersion` API.

#### cupy.cuda.nccl.get_unique_id

`cupy.cuda.nccl.`**`get_unique_id`**`()`

#### cupy.cuda.nccl.groupStart

`cupy.cuda.nccl.`**`groupStart`**`()`
Start a group of NCCL calls. Must be paired with *groupEnd()*.

---

**Note:** This method is only 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()
```

---

**See also:**

ncclGroupStart

#### cupy.cuda.nccl.groupEnd

`cupy.cuda.nccl.`**`groupEnd`**`()`
End a group of NCCL calls. Must be paired with *groupStart()*.

---

**Note:** This method is only 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()
```

---

**See also:**

ncclGroupEnd

### 3.7.8 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.

| |
|---|
| *cupy.cuda.runtime.driverGetVersion* |
| *cupy.cuda.runtime.runtimeGetVersion* |
| *cupy.cuda.runtime.getDevice* |
| *cupy.cuda.runtime.deviceGetAttribute* |
| *cupy.cuda.runtime.deviceGetByPCIBusId* |
| *cupy.cuda.runtime.deviceGetPCIBusId* |
| *cupy.cuda.runtime.getDeviceCount* |
| *cupy.cuda.runtime.setDevice* |
| *cupy.cuda.runtime.deviceSynchronize* |
| *cupy.cuda.runtime.deviceCanAccessPeer* |
| *cupy.cuda.runtime.deviceEnablePeerAccess* |
| *cupy.cuda.runtime.malloc* |
| *cupy.cuda.runtime.mallocManaged* |
| *cupy.cuda.runtime.malloc3DArray* |
| *cupy.cuda.runtime.mallocArray* |
| *cupy.cuda.runtime.hostAlloc* |
| *cupy.cuda.runtime.hostRegister* |
| *cupy.cuda.runtime.hostUnregister* |
| *cupy.cuda.runtime.free* |
| *cupy.cuda.runtime.freeHost* |
| *cupy.cuda.runtime.freeArray* |
| *cupy.cuda.runtime.memGetInfo* |
| *cupy.cuda.runtime.memcpy* |
| *cupy.cuda.runtime.memcpyAsync* |
| *cupy.cuda.runtime.memcpyPeer* |
| *cupy.cuda.runtime.memcpyPeerAsync* |
| *cupy.cuda.runtime.memcpy2D* |
| *cupy.cuda.runtime.memcpy2DAsync* |
| *cupy.cuda.runtime.memcpy2DFromArray* |
| *cupy.cuda.runtime.memcpy2DFromArrayAsync* |
| *cupy.cuda.runtime.memcpy2DToArray* |
| *cupy.cuda.runtime.memcpy2DToArrayAsync* |
| *cupy.cuda.runtime.memcpy3D* |
| *cupy.cuda.runtime.memcpy3DAsync* |
| *cupy.cuda.runtime.memset* |
| *cupy.cuda.runtime.memsetAsync* |
| *cupy.cuda.runtime.memPrefetchAsync* |
| *cupy.cuda.runtime.memAdvise* |
| *cupy.cuda.runtime.pointerGetAttributes* |
| *cupy.cuda.runtime.streamCreate* |
| *cupy.cuda.runtime.streamCreateWithFlags* |
| *cupy.cuda.runtime.streamDestroy* |
| *cupy.cuda.runtime.streamSynchronize* |
| *cupy.cuda.runtime.streamAddCallback* |

Continued on next page

Table 89 – continued from previous page

| |
|---|
| *cupy.cuda.runtime.streamQuery* |
| *cupy.cuda.runtime.streamWaitEvent* |
| *cupy.cuda.runtime.eventCreate* |
| *cupy.cuda.runtime.eventCreateWithFlags* |
| *cupy.cuda.runtime.eventDestroy* |
| *cupy.cuda.runtime.eventElapsedTime* |
| *cupy.cuda.runtime.eventQuery* |
| *cupy.cuda.runtime.eventRecord* |
| *cupy.cuda.runtime.eventSynchronize* |

### cupy.cuda.runtime.driverGetVersion

cupy.cuda.runtime.**driverGetVersion**() → int

### cupy.cuda.runtime.runtimeGetVersion

cupy.cuda.runtime.**runtimeGetVersion**() → int

### cupy.cuda.runtime.getDevice

cupy.cuda.runtime.**getDevice**() → int

### cupy.cuda.runtime.deviceGetAttribute

cupy.cuda.runtime.**deviceGetAttribute**(*int attrib*, *int device*) → int

### cupy.cuda.runtime.deviceGetByPCIBusId

cupy.cuda.runtime.**deviceGetByPCIBusId**(*unicode pci_bus_id*) → int

### cupy.cuda.runtime.deviceGetPCIBusId

cupy.cuda.runtime.**deviceGetPCIBusId**(*int device*) → unicode

### cupy.cuda.runtime.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.malloc

cupy.cuda.runtime.**malloc**(*size_t size*) → intptr_t

### cupy.cuda.runtime.mallocManaged

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

### cupy.cuda.runtime.malloc3DArray

cupy.cuda.runtime.**malloc3DArray**(*intptr_t descPtr*, *size_t width*, *size_t height*, *size_t depth*, *unsigned int flags=0*) → intptr_t

### cupy.cuda.runtime.mallocArray

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

### cupy.cuda.runtime.hostAlloc

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

### cupy.cuda.runtime.hostRegister

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

### cupy.cuda.runtime.hostUnregister

cupy.cuda.runtime.**hostUnregister**(*intptr_t ptr*)

### cupy.cuda.runtime.free

cupy.cuda.runtime.**free**(*intptr_t ptr*)

### cupy.cuda.runtime.freeHost

cupy.cuda.runtime.**freeHost**(*intptr_t ptr*)

### cupy.cuda.runtime.freeArray

cupy.cuda.runtime.**freeArray**(*intptr_t ptr*)

### cupy.cuda.runtime.memGetInfo

cupy.cuda.runtime.**memGetInfo**()

### cupy.cuda.runtime.memcpy

cupy.cuda.runtime.**memcpy**(*intptr_t dst*, *intptr_t src*, *size_t size*, *int kind*)

### cupy.cuda.runtime.memcpyAsync

cupy.cuda.runtime.**memcpyAsync**(*intptr_t dst*, *intptr_t src*, *size_t size*, *int kind*, *intptr_t stream*)

### cupy.cuda.runtime.memcpyPeer

cupy.cuda.runtime.**memcpyPeer**(*intptr_t dst*, *int dstDevice*, *intptr_t src*, *int srcDevice*, *size_t size*)

### cupy.cuda.runtime.memcpyPeerAsync

cupy.cuda.runtime.**memcpyPeerAsync**(*intptr_t dst*, *int dstDevice*, *intptr_t src*, *int srcDevice*, *size_t size*, *intptr_t stream*)

### cupy.cuda.runtime.memcpy2D

cupy.cuda.runtime.**memcpy2D**(*intptr_t dst*, *size_t dpitch*, *intptr_t src*, *size_t spitch*, *size_t width*, *size_t height*, *MemoryKind kind*)

### cupy.cuda.runtime.memcpy2DAsync

cupy.cuda.runtime.**memcpy2DAsync**(*intptr_t dst*, *size_t dpitch*, *intptr_t src*, *size_t spitch*, *size_t width*, *size_t height*, *MemoryKind kind*, *intptr_t stream*)

### cupy.cuda.runtime.memcpy2DFromArray

cupy.cuda.runtime.**memcpy2DFromArray**(*intptr_t dst*, *size_t dpitch*, *intptr_t src*, *size_t wOffset*, *size_t hOffset*, *size_t width*, *size_t height*, *int kind*)

### cupy.cuda.runtime.memcpy2DFromArrayAsync

cupy.cuda.runtime.**memcpy2DFromArrayAsync**(*intptr_t dst*, *size_t dpitch*, *intptr_t src*, *size_t wOffset*, *size_t hOffset*, *size_t width*, *size_t height*, *int kind*, *intptr_t stream*)

### cupy.cuda.runtime.memcpy2DToArray

cupy.cuda.runtime.**memcpy2DToArray**(*intptr_t dst*, *size_t wOffset*, *size_t hOffset*, *intptr_t src*, *size_t spitch*, *size_t width*, *size_t height*, *int kind*)

### cupy.cuda.runtime.memcpy2DToArrayAsync

cupy.cuda.runtime.**memcpy2DToArrayAsync**(*intptr_t dst*, *size_t wOffset*, *size_t hOffset*, *intptr_t src*, *size_t spitch*, *size_t width*, *size_t height*, *int kind*, *intptr_t stream*)

### cupy.cuda.runtime.memcpy3D

cupy.cuda.runtime.**memcpy3D**(*intptr_t Memcpy3DParmsPtr*)

### cupy.cuda.runtime.memcpy3DAsync

cupy.cuda.runtime.**memcpy3DAsync**(*intptr_t Memcpy3DParmsPtr*, *intptr_t stream*)

### cupy.cuda.runtime.memset

cupy.cuda.runtime.**memset**(*intptr_t ptr*, *int value*, *size_t size*)

### cupy.cuda.runtime.memsetAsync

cupy.cuda.runtime.**memsetAsync**(*intptr_t ptr*, *int value*, *size_t size*, *intptr_t stream*)

### cupy.cuda.runtime.memPrefetchAsync

cupy.cuda.runtime.**memPrefetchAsync**(*intptr_t devPtr*, *size_t count*, *int dstDevice*, *intptr_t stream*)

### cupy.cuda.runtime.memAdvise

cupy.cuda.runtime.**memAdvise**(*intptr_t devPtr*, *size_t count*, *int advice*, *int device*)

### cupy.cuda.runtime.pointerGetAttributes

cupy.cuda.runtime.**pointerGetAttributes**(*intptr_t ptr*) → PointerAttributes

### cupy.cuda.runtime.streamCreate

cupy.cuda.runtime.**streamCreate**() → intptr_t

### cupy.cuda.runtime.streamCreateWithFlags

cupy.cuda.runtime.**streamCreateWithFlags**(*unsigned int flags*) → intptr_t

### cupy.cuda.runtime.streamDestroy

cupy.cuda.runtime.**streamDestroy**(*intptr_t stream*)

### cupy.cuda.runtime.streamSynchronize

cupy.cuda.runtime.**streamSynchronize**(*intptr_t stream*)

### cupy.cuda.runtime.streamAddCallback

cupy.cuda.runtime.**streamAddCallback**(*intptr_t stream*, *callback*, *intptr_t arg*, *unsigned int flags=0*)

### cupy.cuda.runtime.streamQuery

cupy.cuda.runtime.**streamQuery**(*intptr_t stream*)

### cupy.cuda.runtime.streamWaitEvent

cupy.cuda.runtime.**streamWaitEvent**(*intptr_t stream*, *intptr_t event*, *unsigned int flags=0*)

### cupy.cuda.runtime.eventCreate

cupy.cuda.runtime.**eventCreate**() → intptr_t

### cupy.cuda.runtime.eventCreateWithFlags

cupy.cuda.runtime.**eventCreateWithFlags**(*unsigned int flags*) → intptr_t

### cupy.cuda.runtime.eventDestroy

cupy.cuda.runtime.**eventDestroy**(*intptr_t event*)

### cupy.cuda.runtime.eventElapsedTime

cupy.cuda.runtime.**eventElapsedTime**(*intptr_t start*, *intptr_t end*) → float

### cupy.cuda.runtime.eventQuery

cupy.cuda.runtime.**eventQuery**(*intptr_t event*)

### cupy.cuda.runtime.eventRecord

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

#### cupy.cuda.runtime.eventSynchronize

cupy.cuda.runtime.**eventSynchronize**(*intptr_t event*)

## 3.8 Kernel binary memoization

| | |
|---|---|
| *cupy.memoize* | Makes a function memoizing the result for each argument and device. |
| *cupy.clear_memo* | Clears the memoized results for all functions decorated by memoize. |

### 3.8.1 cupy.memoize

cupy.**memoize**(*bool for_each_device=False*)

Makes a function memoizing the result for each argument and device.

This decorator provides automatic memoization of the function result.

**Parameters** **for_each_device** (*bool*) – If `True`, it memoizes the results for each device. Otherwise, it memoizes the results only based on the arguments.

### 3.8.2 cupy.clear_memo

cupy.**clear_memo**()

Clears the memoized results for all functions decorated by memoize.

## 3.9 Custom kernels

| | |
|---|---|
| *cupy.ElementwiseKernel* | User-defined elementwise kernel. |
| *cupy.ReductionKernel* | User-defined reduction kernel. |
| *cupy.RawKernel* | User-defined custom kernel. |
| *cupy.RawModule* | User-defined custom module. |
| *cupy.fuse* | Decorator that fuses a function. |

### 3.9.1 cupy.ElementwiseKernel

**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.

**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**

**kwargs**

**name**

**nargs**

**nin**

**no_return**

**nout**

**operation**

**out_params**

**params**

**preamble**

**reduce_dims**

**return_tuple**

### 3.9.2 cupy.ReductionKernel

**class** cupy.**ReductionKernel**

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__**(*self*, **args*, ***kwargs*)

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.

### 3.9.3 cupy.RawKernel

**class** `cupy.`**RawKernel**(*code*, *name*, *options=()*, *backend=u'nvrtc'*, *translate_cucomplex=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`.

#### 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.

#### Attributes

**attributes**

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

```
kernel = RawKernel(...)  # arguments omitted
kernel.max_dynamic_shared_size_bytes = ...
kernel.preferred_shared_memory_carveout = ...
```

Note that the two attributes shown in the above example are the only two currently settable in CUDA.

Any attribute not existing in the present CUDA toolkit version will have the value -1.

**Returns** A dictionary containing the kernel's attributes.

**Return type** dict

**backend**

**binary_version**
The binary architecture version that was used during compilation, in the format: 10*major + minor.

**cache_mode_ca**
Indicates whether option "-Xptxas –dlcm=ca" was set during compilation.

**code**

**const_size_bytes**
The size in bytes of constant memory used by the function.

**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.

### 3.9.4 cupy.RawModule

**class** cupy.**RawModule**(*code=None*, ***, *path=None*, *options=()*, *backend=u'nvrtc'*, *translate_cucomplex=False*)
User-defined custom module.

This class can be used to either compile raw CUDA sources or load CUDA modules (*.cubin). 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 initializing a new instance of this class, and the kernels can be retrieved by calling *get_function()*, which will return an instance of *RawKernel*. (Same as in *RawKernel*, the generated binary is also cached.)

For the latter case, an existing CUDA binary (*.cubin) can be loaded by providing its path, and kernels therein can be retrieved similarly.

**Parameters**

- **code** (*str*) – CUDA source code. Mutually exclusive with `path`.
- **path** (*str*) – Path to cubin/ptx. Mutually exclusive with `code`.
- **options** (*tuple of str*) – Compiler options passed to the backend (NVRTC or NVCC). For details, see https://docs.nvidia.com/cuda/nvrtc/index.html#group__options 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`.

---

**Note:** Each kernel in `RawModule` possesses independent function attributes.

---

### Methods

**get_function**(*self*, *name*)
Retrieve a CUDA kernel by its name from the module.

**Parameters** **name** (*str*) – Name of the kernel function.

**Returns** An `RawKernel` instance.

**Return type** *RawKernel*

**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**

**cubin_path**

**options**

### 3.9.5 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

```
>>> @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])
```

## 3.10 Interoperability

CuPy can also be used in conjunction with other frameworks.

### 3.10.1 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.

```
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.

### 3.10.2 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:

```
import cupy
from numba import cuda

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

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

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

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

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

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

```
import numpy
import numba
import cupy

x = numpy.arange(10)  # type: numpy.ndarray
x_numba = numba.cuda.to_device(x)  # type: numba.cuda.cudadrv.devicearray.
↪DeviceNDArray
x_cupy = cupy.asarray(x_numba)  # type: cupy.ndarray
```

### 3.10.3 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 folowing is a simple example code borrowed from mpi4py Tutorial:

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

import cupy
from mpi4py import MPI

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

(continues on next page)

(continued from previous page)

```
# 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.

### 3.10.4 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()*).

| | |
|---|---|
| *cupy.fromDlpack* | Zero-copy conversion from a DLPack tensor to a *ndarray*. |

#### cupy.fromDlpack

cupy.**fromDlpack**(*dltensor*) → ndarray

Zero-copy conversion from a DLPack tensor to a *ndarray*.

DLPack is a open in memory tensor structure proposed in this repository: dmlc/dlpack.

This function takes a `PyCapsule` object which contains a pointer to a DLPack tensor as input, and returns a *ndarray*. This function does not copy the data in the DLPack tensor but both DLPack tensor and *ndarray* have pointers which are pointing to the same memory region for the data.

**Parameters** **dltensor** (`PyCapsule`) – Input DLPack tensor which is encapsulated in a `PyCapsule` object.

**Returns** A CuPy ndarray.

**Return type** array (*ndarray*)

**See also:**

*cupy.ndarray.toDlpack()* is a method for zero-copy conversion from a *ndarray* to a DLPack tensor (which is encapsulated in a `PyCapsule` object).

**Example**

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

Here is a simple example:

```
import cupy

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

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

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

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

```
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())
```

## 3.11 Testing Modules

CuPy offers testing utilities to support unit testing. They are under namespace *cupy.testing*.

### 3.11.1 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*.

| | |
|---|---|
| *cupy.testing.assert_allclose* | Raises an AssertionError if objects are not equal up to desired tolerance. |
| *cupy.testing.assert_array_almost_equal* | Raises an AssertionError if objects are not equal up to desired precision. |
| *cupy.testing.assert_array_almost_equal_nulp* | Compare two arrays relatively to their spacing. |
| *cupy.testing.assert_array_max_ulp* | Check that all items of arrays differ in at most N Units in the Last Place. |
| *cupy.testing.assert_array_equal* | Raises an AssertionError if two array_like objects are not equal. |
| *cupy.testing.assert_array_list_equal* | Compares lists of arrays pairwise with `assert_array_equal`. |

Continued on next page

Table 93 – continued from previous page

| | |
|---|---|
| *cupy.testing.assert_array_less* | Raises an AssertionError if array_like objects are not ordered by less than. |

### 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.

**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()`

### 3.11.2 NumPy-CuPy Consistency Check

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

| | |
|---|---|
| *cupy.testing.numpy_cupy_allclose* | Decorator that checks NumPy results and CuPy ones are close. |
| *cupy.testing.numpy_cupy_array_almost_equal* | Decorator that checks NumPy results and CuPy ones are almost equal. |
| *cupy.testing.numpy_cupy_array_almost_equal_nulp* | Decorator that checks results of NumPy and CuPy are equal w.r.t. |
| *cupy.testing.numpy_cupy_array_max_ulp* | Decorator that checks results of NumPy and CuPy ones are equal w.r.t. |
| *cupy.testing.numpy_cupy_array_equal* | Decorator that checks NumPy results and CuPy ones are equal. |
| *cupy.testing.numpy_cupy_array_list_equal* | Decorator that checks the resulting lists of NumPy and CuPy's one are equal. |
| *cupy.testing.numpy_cupy_array_less* | Decorator that checks the CuPy result is less than NumPy result. |
| *cupy.testing.numpy_cupy_raises* | Decorator that checks the NumPy and CuPy throw same errors. |

#### 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.

```
>>> import unittest
>>> from cupy import testing
>>> @testing.gpu
... class TestFoo(unittest.TestCase):
...
...     @testing.numpy_cupy_allclose()
...     def test_foo(self, xp):
...         # ...
...         # Prepare data with xp
...         # ...
...
...         xp_result = xp.zeros(10)
...         return xp_result
```

**See also:**

*cupy.testing.assert_allclose()*

### cupy.testing.numpy_cupy_array_almost_equal

cupy.testing.**numpy_cupy_array_almost_equal**(*decimal=6, err_msg='', verbose=True, name='xp', type_check=True, accept_error=False, sp_name=None, scipy_name=None*)

Decorator that checks NumPy results and CuPy ones are almost equal.

**Parameters**

- **decimal** (*int*) – Desired precision.
- **err_msg** (*str*) – The error message to be printed in case of failure.
- **verbose** (*bool*) – If `True`, the conflicting values are appended to the error message.
- **name** (*str*) – Argument name whose value is either `numpy` or `cupy` module.

- **type_check** (*bool*) – If `True`, consistency of dtype is also checked.
- **accept_error** (*bool, Exception or tuple of Exception*) – Specify acceptable errors. When both NumPy test and CuPy test raises the same type of errors, and the type of the errors is specified with this argument, the errors are ignored and not raised. If it is `True` all error types are acceptable. If it is `False` no error is acceptable.
- **sp_name** (*str or None*) – Argument name whose value is either `scipy.sparse` or `cupyx.scipy.sparse` module. If `None`, no argument is given for the modules.
- **scipy_name** (*str or None*) – Argument name whose value is either `scipy` or `cupyx.scipy` module. If `None`, no argument is given for the modules.

Decorated test fixture is required to return the same arrays in the sense of *cupy.testing.assert_array_almost_equal()* (except the type of array module) even if `xp` is `numpy` or `cupy`.

**See also:**

*cupy.testing.assert_array_almost_equal()*

### cupy.testing.numpy_cupy_array_almost_equal_nulp

cupy.testing.**numpy_cupy_array_almost_equal_nulp**(*nulp=1*, *name='xp'*, *type_check=True*, *accept_error=False*, *sp_name=None*, *scipy_name=None*)

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

**Parameters**

- **nulp** (*int*) – The maximum number of unit in the last place for tolerance.
- **name** (*str*) – Argument name whose value is either `numpy` or `cupy` module.
- **type_check** (*bool*) – If `True`, consistency of dtype is also checked.
- **accept_error** (*bool, Exception or tuple of Exception*) – Specify acceptable errors. When both NumPy test and CuPy test raises the same type of errors, and the type of the errors is specified with this argument, the errors are ignored and not raised. If it is `True`, all error types are acceptable. If it is `False`, no error is acceptable.
- **sp_name** (*str or None*) – Argument name whose value is either `scipy.sparse` or `cupyx.scipy.sparse` module. If `None`, no argument is given for the modules.
- **scipy_name** (*str or None*) – Argument name whose value is either `scipy` or `cupyx.scipy` module. If `None`, no argument is given for the modules.

Decorated test fixture is required to return the same arrays in the sense of *cupy.testing.assert_array_almost_equal_nulp()* (except the type of array module) even if `xp` is `numpy` or `cupy`.

**See also:**

*cupy.testing.assert_array_almost_equal_nulp()*

### cupy.testing.numpy_cupy_array_max_ulp

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

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

**Parameters**

- **maxulp** (*int*) – The maximum number of units in the last place that elements of resulting two arrays can differ.
- **dtype** (*numpy.dtype*) – Data-type to convert the resulting two array to if given.
- **name** (*str*) – Argument name whose value is either `numpy` or `cupy` module.
- **type_check** (*bool*) – If `True`, consistency of dtype is also checked.
- **accept_error** (*bool, Exception or tuple of Exception*) – Specify acceptable errors. When both NumPy test and CuPy test raises the same type of errors, and the type of the errors is specified with this argument, the errors are ignored and not raised. If it is `True` all error types are acceptable. If it is `False` no error is acceptable.
- **sp_name** (*str or None*) – Argument name whose value is either `scipy.sparse` or `cupyx.scipy.sparse` module. If `None`, no argument is given for the modules.
- **scipy_name** (*str or None*) – Argument name whose value is either `scipy` or `cupyx.scipy` module. If `None`, no argument is given for the modules.

Decorated test fixture is required to return the same arrays in the sense of *assert_array_max_ulp()* (except the type of array module) even if `xp` is `numpy` or `cupy`.

**See also:**

*cupy.testing.assert_array_max_ulp()*

### cupy.testing.numpy_cupy_array_equal

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

Decorator that checks NumPy results and CuPy ones are equal.

**Parameters**

- **err_msg** (*str*) – The error message to be printed in case of failure.
- **verbose** (*bool*) – If `True`, the conflicting values are appended to the error message.
- **name** (*str*) – Argument name whose value is either `numpy` or `cupy` module.
- **type_check** (*bool*) – If `True`, consistency of dtype is also checked.
- **accept_error** (*bool, Exception or tuple of Exception*) – Specify acceptable errors. When both NumPy test and CuPy test raises the same type of errors, and the type of the errors is specified with this argument, the errors are ignored and not raised. If it is `True` all error types are acceptable. If it is `False` no error is acceptable.
- **sp_name** (*str or None*) – Argument name whose value is either `scipy.sparse` or `cupyx.scipy.sparse` module. If `None`, no argument is given for the modules.
- **scipy_name** (*str or None*) – Argument name whose value is either `scipy` or `cupyx.scipy` module. If `None`, no argument is given for the modules.
- **strides_check** (*bool*) – If `True`, consistency of strides is also checked.

Decorated test fixture is required to return the same arrays in the sense of *numpy_cupy_array_equal()* (except the type of array module) even if `xp` is `numpy` or `cupy`.

**See also:**

*cupy.testing.assert_array_equal()*

### cupy.testing.numpy_cupy_array_list_equal

cupy.testing.**numpy_cupy_array_list_equal**(*err_msg=''*, *verbose=True*, *name='xp'*, *sp_name=None*, *scipy_name=None*)

Decorator that checks the resulting lists of NumPy and CuPy's one are equal.

**Parameters**

- **err_msg** (*str*) – The error message to be printed in case of failure.
- **verbose** (*bool*) – If `True`, the conflicting values are appended to the error message.
- **name** (*str*) – Argument name whose value is either `numpy` or `cupy` module.
- **sp_name** (*str or None*) – Argument name whose value is either `scipy.sparse` or `cupyx.scipy.sparse` module. If `None`, no argument is given for the modules.
- **scipy_name** (*str or None*) – Argument name whose value is either `scipy` or `cupyx.scipy` module. If `None`, no argument is given for the modules.

Decorated test fixture is required to return the same list of arrays (except the type of array module) even if `xp` is `numpy` or `cupy`.

**See also:**

*cupy.testing.assert_array_list_equal()*

### cupy.testing.numpy_cupy_array_less

cupy.testing.**numpy_cupy_array_less**(*err_msg=''*, *verbose=True*, *name='xp'*, *type_check=True*, *accept_error=False*, *sp_name=None*, *scipy_name=None*)

Decorator that checks the CuPy result is less than NumPy result.

**Parameters**

- **err_msg** (*str*) – The error message to be printed in case of failure.
- **verbose** (*bool*) – If `True`, the conflicting values are appended to the error message.
- **name** (*str*) – Argument name whose value is either `numpy` or `cupy` module.
- **type_check** (*bool*) – If `True`, consistency of dtype is also checked.
- **accept_error** (*bool, Exception or tuple of Exception*) – Specify acceptable errors. When both NumPy test and CuPy test raises the same type of errors, and the type of the errors is specified with this argument, the errors are ignored and not raised. If it is `True` all error types are acceptable. If it is `False` no error is acceptable.
- **sp_name** (*str or None*) – Argument name whose value is either `scipy.sparse` or `cupyx.scipy.sparse` module. If `None`, no argument is given for the modules.
- **scipy_name** (*str or None*) – Argument name whose value is either `scipy` or `cupyx.scipy` module. If `None`, no argument is given for the modules.

Decorated test fixture is required to return the smaller array when `xp` is `cupy` than the one when `xp` is `numpy`.

**See also:**

*cupy.testing.assert_array_less()*

#### cupy.testing.numpy_cupy_raises

cupy.testing.**numpy_cupy_raises**(*name='xp'*, *sp_name=None*, *scipy_name=None*, *accept_error=<class 'Exception'>*)

Decorator that checks the NumPy and CuPy throw same errors.

**Parameters**

- **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.
- **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.

Decorated test fixture is required throw same errors even if `xp` is `numpy` or `cupy`.

### 3.11.3 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* | Decorator for parameterized dtype test. |
| *cupy.testing.for_all_dtypes* | Decorator that checks the fixture with all dtypes. |
| *cupy.testing.for_float_dtypes* | Decorator that checks the fixture with float dtypes. |
| *cupy.testing.for_signed_dtypes* | Decorator that checks the fixture with signed dtypes. |
| *cupy.testing.for_unsigned_dtypes* | Decorator that checks the fixture with unsinged dtypes. |
| *cupy.testing.for_int_dtypes* | Decorator that checks the fixture with integer and optionally bool dtypes. |
| *cupy.testing.for_complex_dtypes* | Decorator that checks the fixture with complex dtypes. |
| *cupy.testing.for_dtypes_combination* | Decorator that checks the fixture with a product set of dtypes. |
| *cupy.testing.for_all_dtypes_combination* | Decorator that checks the fixture with a product set of all dtypes. |
| *cupy.testing.for_signed_dtypes_combination* | Decorator for parameterized test w.r.t. |
| *cupy.testing.for_unsigned_dtypes_combination* | Decorator for parameterized test w.r.t. |
| *cupy.testing.for_int_dtypes_combination* | Decorator for parameterized test w.r.t. |

#### cupy.testing.for_dtypes

cupy.testing.**for_dtypes**(*dtypes*, *name='dtype'*)

Decorator for parameterized dtype test.

**Parameters**

- **dtypes** (*list of dtypes*) – dtypes to be tested.
- **name** (*str*) – Argument name to which specified dtypes are passed.

This decorator adds a keyword argument specified by `name` to the test fixture. Then, it runs the fixtures in

parallel by passing the each element of `dtypes` to the named argument.

### cupy.testing.for_all_dtypes

`cupy.testing.`**`for_all_dtypes`**(*name='dtype'*, *no_float16=False*, *no_bool=False*, *no_complex=False*)

Decorator that checks the fixture with all dtypes.

**Parameters**

- **name** (*str*) – Argument name to which specified dtypes are passed.
- **no_float16** (*bool*) – If `True`, `numpy.float16` is omitted from candidate dtypes.
- **no_bool** (*bool*) – If `True`, `numpy.bool_` is omitted from candidate dtypes.
- **no_complex** (*bool*) – If `True`, `numpy.complex64` and `numpy.complex128` are omitted from candidate dtypes.

dtypes to be tested: `numpy.complex64` (optional), `numpy.complex128` (optional), `numpy.float16` (optional), `numpy.float32`, `numpy.float64`, `numpy.dtype('b')`, `numpy.dtype('h')`, `numpy.dtype('i')`, `numpy.dtype('l')`, `numpy.dtype('q')`, `numpy.dtype('B')`, `numpy.dtype('H')`, `numpy.dtype('I')`, `numpy.dtype('L')`, `numpy.dtype('Q')`, and `numpy.bool_` (optional).

The usage is as follows. This test fixture checks if `cPickle` successfully reconstructs *cupy.ndarray* for various dtypes. `dtype` is an argument inserted by the decorator.

```
>>> import unittest
>>> from cupy import testing
>>> @testing.gpu
... class TestNpz(unittest.TestCase):
...
...     @testing.for_all_dtypes()
...     def test_pickle(self, dtype):
...         a = testing.shaped_arange((2, 3, 4), dtype=dtype)
...         s = six.moves.cPickle.dumps(a)
...         b = six.moves.cPickle.loads(s)
...         testing.assert_array_equal(a, b)
```

Typically, we use this decorator in combination with decorators that check consistency between NumPy and CuPy like *cupy.testing.numpy_cupy_allclose()*. The following is such an example.

```
>>> import unittest
>>> from cupy import testing
>>> @testing.gpu
... class TestMean(unittest.TestCase):
...
...     @testing.for_all_dtypes()
...     @testing.numpy_cupy_allclose()
...     def test_mean_all(self, xp, dtype):
...         a = testing.shaped_arange((2, 3), xp, dtype)
...         return a.mean()
```

**See also:**

*cupy.testing.for_dtypes()*

### cupy.testing.for_float_dtypes

cupy.testing.**for_float_dtypes**(*name='dtype'*, *no_float16=False*)
Decorator that checks the fixture with float dtypes.

**Parameters**

- **name** (*str*) – Argument name to which specified dtypes are passed.
- **no_float16** (*bool*) – If `True`, `numpy.float16` is omitted from candidate dtypes.

dtypes to be tested are `numpy.float16` (optional), `numpy.float32`, and `numpy.float64`.

**See also:**

*cupy.testing.for_dtypes()*, *cupy.testing.for_all_dtypes()*

### cupy.testing.for_signed_dtypes

cupy.testing.**for_signed_dtypes**(*name='dtype'*)
Decorator that checks the fixture with signed dtypes.

**Parameters** **name** (*str*) – Argument name to which specified dtypes are passed.

dtypes to be tested are `numpy.dtype('b')`, `numpy.dtype('h')`, `numpy.dtype('i')`, `numpy.dtype('l')`, and `numpy.dtype('q')`.

**See also:**

*cupy.testing.for_dtypes()*, *cupy.testing.for_all_dtypes()*

### cupy.testing.for_unsigned_dtypes

cupy.testing.**for_unsigned_dtypes**(*name='dtype'*)
Decorator that checks the fixture with unsinged 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()*

### 3.11.4 Parameterized order Test

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

| | |
|---|---|
| *cupy.testing.for_orders* | Decorator to parameterize tests with order. |
| *cupy.testing.for_CF_orders* | Decorator that checks the fixture with orders 'C' and 'F'. |

#### cupy.testing.for_orders

cupy.testing.**for_orders**(*orders*, *name='order'*)

Decorator to parameterize tests with order.

**Parameters**

- **orders** (*list of order*) – orders to be tested.
- **name** (*str*) – Argument name to which the specified order is passed.

This decorator adds a keyword argument specified by `name` to the test fixtures. Then, the fixtures run by passing each element of `orders` to the named argument.

#### cupy.testing.for_CF_orders

cupy.testing.**for_CF_orders**(*name='order'*)

Decorator that checks the fixture with orders 'C' and 'F'.

**Parameters** **name** (*str*) – Argument name to which the specified order is passed.

**See also:**

*cupy.testing.for_all_dtypes()*

## 3.12 Profiling

### 3.12.1 time range

| | |
|---|---|
| *cupy.prof.TimeRangeDecorator* | Decorator to mark function calls with range in NVIDIA profiler |
| *cupy.prof.time_range* | A context manager to describe the enclosed block as a nested range |

#### cupy.prof.TimeRangeDecorator

**class** cupy.prof.**TimeRangeDecorator**(*message=None*, *color_id=None*, *argb_color=None*, *sync=False*)

Decorator to mark function calls with range in NVIDIA profiler

Decorated function calls are marked as ranges in NVIDIA profiler timeline.

```
>>> from cupy import prof
>>> @cupy.prof.TimeRangeDecorator()
```

(continues on next page)

(continued from previous page)

```
... def function_to_profile():
...     pass
```

**Parameters**

- **message** (*str*) – Name of a range, default use `func.__name__`.
- **color_id** – range color ID
- **argb_color** – range color in ARGB (e.g. 0xFF00FF00 for green)
- **sync** (*bool*) – If `True`, waits for completion of all outstanding processing on GPU before calling *cupy.cuda.nvtx.RangePush()* or *cupy.cuda.nvtx.RangePop()*

**See also:**

*cupy.cuda.nvtx.RangePush() cupy.cuda.nvtx.RangePop()*

### Methods

**__call__**(*func*)
Call self as a function.

**__enter__**()

**__exit__**(*exc_type*, *exc_value*, *traceback*)

## cupy.prof.time_range

cupy.prof.**time_range**(*message*, *color_id=None*, *argb_color=None*, *sync=False*)
A context manager to describe the enclosed block as a nested range

```
>>> from cupy import prof
>>> with cupy.prof.time_range('some range in green', color_id=0):
...     # do something you want to measure
...     pass
```

**Parameters**

- **message** – Name of a range.
- **color_id** – range color ID
- **argb_color** – range color in ARGB (e.g. 0xFF00FF00 for green)
- **sync** (*bool*) – If `True`, waits for completion of all outstanding processing on GPU before calling *cupy.cuda.nvtx.RangePush()* or *cupy.cuda.nvtx.RangePop()*

**See also:**

*cupy.cuda.nvtx.RangePush() cupy.cuda.nvtx.RangePop()*

## 3.13 Environment variables

Here are the environment variables CuPy uses.

| | |
|---|---|
| CUDA_PATH | Path to the directory containing CUDA. The parent of the directory containing nvcc is used as default. When nvcc is not found, /usr/local/cuda is used. See *Working with Custom CUDA Installation* for details. |
| CUPY_CACHE_DIR | Path to the directory to store kernel cache. ${HOME}/.cupy/kernel_cache is used by default. See *Overview* for details. |
| CUPY_CACHE_SAVE_CUDA_SOURCE | 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. |
| CUPY_DUMP_CUDA_SOURCE_ON_ERROR | If set to 1, when CUDA kernel compilation fails, CuPy dumps CUDA kernel code to standard error. It is disabled by default. |
| CUPY_CUDA_COMPILE_WITH_DEBUG | If set to 1, CUDA kernel will be compiled with debug information (--device-debug and --generate-line-info). It is disabled by default. |
| CUPY_GPU_MEMORY_LIMIT | The amount of memory that can be allocated for each device. The value can be specified in absolute bytes or fraction (e.g., "90%") of the total memory of each GPU. See *Memory Management* for details. 0 (unlimited) is used by default. |
| CUPY_SEED | Set the seed for random number generators. For historical reasons CHAINER_SEED is used if CUPY_SEED is unspecified. |
| CUPY_EXPERIMENTAL_SLICE_COPY | If set to 1, the following syntax is enabled: cupy_ndarray[:] = numpy_ndarray. |

Moreover, as in any CUDA programs, all of the CUDA environment variables listed in the CUDA Toolkit Documentation will also be honored.

### 3.13.1 For installation

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

| | |
|---|---|
| CUDA_PATH | See the description above. |
| CUTENSOR_PATH | Path to the cuTENSOR root directory that contains lib and include directories. (experimental) |
| NVCC | Define the compiler to use when compiling CUDA files. |
| CUPY_PYTHON_350_FORCE | Enforce CuPy to be installed against Python 3.5.0 (not recommended). |
| CUPY_INSTALL_USE_HIP | For building the ROCm support, see *Install CuPy from Source* for further detail. |
| CUPY_NVCC_GENERATE_CODE | To build CuPy for a particular CUDA architecture. For example, CUPY_NVCC_GENERATE_CODE=compute_60,sm_60. When this is not set, the default is to support all architectures. |

## 3.14 Difference between CuPy and NumPy

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

### 3.14.1 Cast behavior from float to integer

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

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

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

### 3.14.2 Random methods support dtype argument

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

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

### 3.14.3 Out-of-bounds indices

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

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

### 3.14.4 Duplicate values in indices

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

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

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

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

### 3.14.5 Zero-dimensional array

#### Reduction methods

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

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

#### Type promotion

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

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

### 3.14.6 Data types

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

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

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

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

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

### 3.14.8 Random seed arrays are hashed to scalars

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

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

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

## 3.15 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.

### 3.15.1 NumPy / CuPy APIs

#### Module-Level

| NumPy | CuPy |
|---|---|
| numpy.abs | cupy.abs |
| numpy.absolute | *cupy.absolute* |
| numpy.add | *cupy.add* |
| numpy.add_docstring | - |
| numpy.add_newdoc | - |
| numpy.add_newdoc_ufunc | - |
| numpy.alen | - |
| numpy.all | *cupy.all* |
| numpy.allclose | *cupy.allclose* |
| numpy.alltrue | - |
| numpy.amax | *cupy.amax* |
| numpy.amin | *cupy.amin* |
| numpy.angle | *cupy.angle* |
| numpy.any | *cupy.any* |
| numpy.append | - |
| numpy.apply_along_axis | - |
| numpy.apply_over_axes | - |
| numpy.arange | *cupy.arange* |
| numpy.arccos | *cupy.arccos* |
| numpy.arccosh | *cupy.arccosh* |
| numpy.arcsin | *cupy.arcsin* |

Continued on next page

Table 98 – continued from previous page

| NumPy | CuPy |
|---|---|
| numpy.arcsinh | *cupy.arcsinh* |
| numpy.arctan | *cupy.arctan* |
| numpy.arctan2 | *cupy.arctan2* |
| numpy.arctanh | *cupy.arctanh* |
| numpy.argmax | *cupy.argmax* |
| numpy.argmin | *cupy.argmin* |
| numpy.argpartition | *cupy.argpartition* |
| numpy.argsort | *cupy.argsort* |
| numpy.argwhere | - |
| numpy.around | *cupy.around* |
| numpy.array | *cupy.array* |
| numpy.array2string | - |
| numpy.array_equal | - |
| numpy.array_equiv | - |
| numpy.array_repr | *cupy.array_repr* |
| numpy.array_split | *cupy.array_split* |
| numpy.array_str | *cupy.array_str* |
| numpy.asanyarray | *cupy.asanyarray* |
| numpy.asarray | *cupy.asarray* |
| numpy.asarray_chkfinite | - |
| numpy.ascontiguousarray | *cupy.ascontiguousarray* |
| numpy.asfarray | - |
| numpy.asfortranarray | *cupy.asfortranarray* |
| numpy.asmatrix | - |
| numpy.asscalar | - |
| numpy.atleast_1d | *cupy.atleast_1d* |
| numpy.atleast_2d | *cupy.atleast_2d* |
| numpy.atleast_3d | *cupy.atleast_3d* |
| numpy.average | *cupy.average* |
| numpy.bartlett | - |
| numpy.base_repr | *cupy.base_repr* |
| numpy.binary_repr | *cupy.binary_repr* |
| numpy.bincount | *cupy.bincount* |
| numpy.bitwise_and | *cupy.bitwise_and* |
| numpy.bitwise_not | - |
| numpy.bitwise_or | *cupy.bitwise_or* |
| numpy.bitwise_xor | *cupy.bitwise_xor* |
| numpy.blackman | *cupy.blackman* |
| numpy.block | - |
| numpy.bmat | - |
| numpy.broadcast_arrays | *cupy.broadcast_arrays* |
| numpy.broadcast_to | *cupy.broadcast_to* |
| numpy.busday_count | - |
| numpy.busday_offset | - |
| numpy.byte_bounds | - |
| numpy.can_cast | *cupy.can_cast* |
| numpy.cbrt | *cupy.cbrt* |
| numpy.ceil | *cupy.ceil* |
| numpy.choose | *cupy.choose* |

Continued on next page

Table 98 – continued from previous page

| NumPy | CuPy |
|---|---|
| numpy.clip | *cupy.clip* |
| numpy.column_stack | *cupy.column_stack* |
| numpy.common_type | *cupy.common_type* |
| numpy.compare_chararrays | - |
| numpy.compress | - |
| numpy.concatenate | *cupy.concatenate* |
| numpy.conj | *cupy.conj* |
| numpy.conjugate | - |
| numpy.convolve | - |
| numpy.copy | *cupy.copy* |
| numpy.copysign | *cupy.copysign* |
| numpy.copyto | *cupy.copyto* |
| numpy.corrcoef | *cupy.corrcoef* |
| numpy.correlate | - |
| numpy.cos | *cupy.cos* |
| numpy.cosh | *cupy.cosh* |
| numpy.count_nonzero | *cupy.count_nonzero* |
| numpy.cov | *cupy.cov* |
| numpy.cross | *cupy.cross* |
| numpy.cumprod | *cupy.cumprod* |
| numpy.cumproduct | - |
| numpy.cumsum | *cupy.cumsum* |
| numpy.datetime_as_string | - |
| numpy.datetime_data | - |
| numpy.deg2rad | *cupy.deg2rad* |
| numpy.degrees | *cupy.degrees* |
| numpy.delete | - |
| numpy.deprecate | - |
| numpy.deprecate_with_doc | - |
| numpy.diag | *cupy.diag* |
| numpy.diag_indices | - |
| numpy.diag_indices_from | - |
| numpy.diagflat | *cupy.diagflat* |
| numpy.diagonal | *cupy.diagonal* |
| numpy.diff | *cupy.diff* |
| numpy.digitize | - |
| numpy.disp | - |
| numpy.divide | *cupy.divide* |
| numpy.divmod | *cupy.divmod* |
| numpy.dot | *cupy.dot* |
| numpy.dsplit | *cupy.dsplit* |
| numpy.dstack | *cupy.dstack* |
| numpy.ediff1d | - |
| numpy.einsum | *cupy.einsum* |
| numpy.einsum_path | - |
| numpy.empty | *cupy.empty* |
| numpy.empty_like | *cupy.empty_like* |
| numpy.equal | *cupy.equal* |
| numpy.exp | *cupy.exp* |

Continued on next page

Table 98 – continued from previous page

| NumPy | CuPy |
|---|---|
| numpy.exp2 | *cupy.exp2* |
| numpy.expand_dims | *cupy.expand_dims* |
| numpy.expm1 | *cupy.expm1* |
| numpy.extract | - |
| numpy.eye | *cupy.eye* |
| numpy.fabs | - |
| numpy.fastCopyAndTranspose | - |
| numpy.fill_diagonal | *cupy.fill_diagonal* |
| numpy.find_common_type | cupy.find_common_type (*alias of* numpy.find_common_type) |
| numpy.fix | *cupy.fix* |
| numpy.flatnonzero | *cupy.flatnonzero* |
| numpy.flip | *cupy.flip* |
| numpy.fliplr | *cupy.fliplr* |
| numpy.flipud | *cupy.flipud* |
| numpy.float_power | - |
| numpy.floor | *cupy.floor* |
| numpy.floor_divide | *cupy.floor_divide* |
| numpy.fmax | *cupy.fmax* |
| numpy.fmin | *cupy.fmin* |
| numpy.fmod | *cupy.fmod* |
| numpy.format_float_positional | - |
| numpy.format_float_scientific | - |
| numpy.frexp | *cupy.frexp* |
| numpy.frombuffer | - |
| numpy.fromfile | *cupy.fromfile* |
| numpy.fromfunction | - |
| numpy.fromiter | - |
| numpy.frompyfunc | - |
| numpy.fromregex | - |
| numpy.fromstring | - |
| numpy.full | *cupy.full* |
| numpy.full_like | *cupy.full_like* |
| numpy.fv | - |
| numpy.gcd | - |
| numpy.genfromtxt | - |
| numpy.geomspace | - |
| numpy.get_array_wrap | - |
| numpy.get_include | - |
| numpy.get_printoptions | - |
| numpy.getbufsize | - |
| numpy.geterr | - |
| numpy.geterrcall | - |
| numpy.geterrobj | - |
| numpy.gradient | - |
| numpy.greater | *cupy.greater* |
| numpy.greater_equal | *cupy.greater_equal* |
| numpy.hamming | *cupy.hamming* |
| numpy.hanning | *cupy.hanning* |
| numpy.heaviside | - |

Continued on next page

Table 98 – continued from previous page

| NumPy | CuPy |
|---|---|
| numpy.histogram | *cupy.histogram* |
| numpy.histogram2d | - |
| numpy.histogram_bin_edges | - |
| numpy.histogramdd | - |
| numpy.hsplit | *cupy.hsplit* |
| numpy.hstack | *cupy.hstack* |
| numpy.hypot | *cupy.hypot* |
| numpy.i0 | *cupy.i0* |
| numpy.identity | *cupy.identity* |
| numpy.imag | *cupy.imag* |
| numpy.in1d | *cupy.in1d* |
| numpy.indices | *cupy.indices* |
| numpy.info | - |
| numpy.inner | *cupy.inner* |
| numpy.insert | - |
| numpy.int_asbuffer | - |
| numpy.interp | - |
| numpy.intersect1d | - |
| numpy.invert | *cupy.invert* |
| numpy.ipmt | - |
| numpy.irr | - |
| numpy.is_busday | - |
| numpy.isclose | *cupy.isclose* |
| numpy.iscomplex | *cupy.iscomplex* |
| numpy.iscomplexobj | *cupy.iscomplexobj* |
| numpy.isfinite | *cupy.isfinite* |
| numpy.isfortran | *cupy.isfortran* |
| numpy.isin | *cupy.isin* |
| numpy.isinf | *cupy.isinf* |
| numpy.isnan | *cupy.isnan* |
| numpy.isnat | - |
| numpy.isneginf | - |
| numpy.isposinf | - |
| numpy.isreal | *cupy.isreal* |
| numpy.isrealobj | *cupy.isrealobj* |
| numpy.isscalar | *cupy.isscalar* |
| numpy.issctype | cupy.issctype (*alias of* numpy.issctype) |
| numpy.issubclass_ | cupy.issubclass_ (*alias of* numpy.issubclass_) |
| numpy.issubdtype | cupy.issubdtype (*alias of* numpy.issubdtype) |
| numpy.issubsctype | cupy.issubsctype (*alias of* numpy.issubsctype) |
| numpy.iterable | - |
| numpy.ix_ | *cupy.ix_* |
| numpy.kaiser | - |
| numpy.kron | *cupy.kron* |
| numpy.lcm | - |
| numpy.ldexp | *cupy.ldexp* |
| numpy.left_shift | *cupy.left_shift* |
| numpy.less | *cupy.less* |
| numpy.less_equal | *cupy.less_equal* |

Continued on next page

Table 98 – continued from previous page

| NumPy | CuPy |
|---|---|
| numpy.lexsort | *cupy.lexsort* |
| numpy.linspace | *cupy.linspace* |
| numpy.load | *cupy.load* |
| numpy.loads | - |
| numpy.loadtxt | - |
| numpy.log | *cupy.log* |
| numpy.log10 | *cupy.log10* |
| numpy.log1p | *cupy.log1p* |
| numpy.log2 | *cupy.log2* |
| numpy.logaddexp | *cupy.logaddexp* |
| numpy.logaddexp2 | *cupy.logaddexp2* |
| numpy.logical_and | *cupy.logical_and* |
| numpy.logical_not | *cupy.logical_not* |
| numpy.logical_or | *cupy.logical_or* |
| numpy.logical_xor | *cupy.logical_xor* |
| numpy.logspace | *cupy.logspace* |
| numpy.lookfor | - |
| numpy.mafromtxt | - |
| numpy.mask_indices | - |
| numpy.mat | - |
| numpy.matmul | *cupy.matmul* |
| numpy.max | cupy.max |
| numpy.maximum | *cupy.maximum* |
| numpy.maximum_sctype | - |
| numpy.may_share_memory | cupy.may_share_memory |
| numpy.mean | *cupy.mean* |
| numpy.median | - |
| numpy.meshgrid | *cupy.meshgrid* |
| numpy.min | cupy.min |
| numpy.min_scalar_type | cupy.min_scalar_type (*alias of* numpy.min_scalar_type) |
| numpy.minimum | *cupy.minimum* |
| numpy.mintypecode | cupy.mintypecode (*alias of* numpy.mintypecode) |
| numpy.mirr | - |
| numpy.mod | *cupy.mod* |
| numpy.modf | *cupy.modf* |
| numpy.moveaxis | *cupy.moveaxis* |
| numpy.msort | *cupy.msort* |
| numpy.multiply | *cupy.multiply* |
| numpy.nan_to_num | *cupy.nan_to_num* |
| numpy.nanargmax | *cupy.nanargmax* |
| numpy.nanargmin | *cupy.nanargmin* |
| numpy.nancumprod | - |
| numpy.nancumsum | - |
| numpy.nanmax | *cupy.nanmax* |
| numpy.nanmean | *cupy.nanmean* |
| numpy.nanmedian | - |
| numpy.nanmin | *cupy.nanmin* |
| numpy.nanpercentile | - |
| numpy.nanprod | *cupy.nanprod* |

Continued on next page

Table 98 – continued from previous page

| NumPy | CuPy |
|---|---|
| numpy.nanquantile | - |
| numpy.nanstd | *cupy.nanstd* |
| numpy.nansum | *cupy.nansum* |
| numpy.nanvar | *cupy.nanvar* |
| numpy.ndfromtxt | - |
| numpy.ndim | - |
| numpy.negative | *cupy.negative* |
| numpy.nested_iters | - |
| numpy.nextafter | *cupy.nextafter* |
| numpy.nonzero | *cupy.nonzero* |
| numpy.not_equal | *cupy.not_equal* |
| numpy.nper | - |
| numpy.npv | - |
| numpy.obj2sctype | cupy.obj2sctype (*alias of* numpy.obj2sctype) |
| numpy.ones | *cupy.ones* |
| numpy.ones_like | *cupy.ones_like* |
| numpy.outer | *cupy.outer* |
| numpy.packbits | *cupy.packbits* |
| numpy.pad | *cupy.pad* |
| numpy.partition | *cupy.partition* |
| numpy.percentile | *cupy.percentile* |
| numpy.piecewise | - |
| numpy.place | *cupy.place* |
| numpy.pmt | - |
| numpy.poly | - |
| numpy.polyadd | - |
| numpy.polyder | - |
| numpy.polydiv | - |
| numpy.polyfit | - |
| numpy.polyint | - |
| numpy.polymul | - |
| numpy.polysub | - |
| numpy.polyval | - |
| numpy.positive | - |
| numpy.power | *cupy.power* |
| numpy.ppmt | - |
| numpy.printoptions | - |
| numpy.prod | *cupy.prod* |
| numpy.product | - |
| numpy.promote_types | cupy.promote_types (*alias of* numpy.promote_types) |
| numpy.ptp | - |
| numpy.put | *cupy.put* |
| numpy.put_along_axis | - |
| numpy.putmask | - |
| numpy.pv | - |
| numpy.quantile | - |
| numpy.rad2deg | *cupy.rad2deg* |
| numpy.radians | *cupy.radians* |
| numpy.rate | - |

Continued on next page

Table 98 – continued from previous page

| NumPy | CuPy |
|---|---|
| numpy.ravel | *cupy.ravel* |
| numpy.ravel_multi_index | - |
| numpy.real | *cupy.real* |
| numpy.real_if_close | - |
| numpy.recfromcsv | - |
| numpy.recfromtxt | - |
| numpy.reciprocal | *cupy.reciprocal* |
| numpy.remainder | *cupy.remainder* |
| numpy.repeat | *cupy.repeat* |
| numpy.require | - |
| numpy.reshape | *cupy.reshape* |
| numpy.resize | - |
| numpy.result_type | *cupy.result_type* |
| numpy.right_shift | *cupy.right_shift* |
| numpy.rint | *cupy.rint* |
| numpy.roll | *cupy.roll* |
| numpy.rollaxis | *cupy.rollaxis* |
| numpy.roots | - |
| numpy.rot90 | *cupy.rot90* |
| numpy.round | - |
| numpy.round_ | *cupy.round_* |
| numpy.row_stack | - |
| numpy.safe_eval | - |
| numpy.save | *cupy.save* |
| numpy.savetxt | - |
| numpy.savez | *cupy.savez* |
| numpy.savez_compressed | *cupy.savez_compressed* |
| numpy.sctype2char | cupy.sctype2char (*alias of* numpy.sctype2char) |
| numpy.searchsorted | cupy.searchsorted |
| numpy.select | - |
| numpy.set_numeric_ops | - |
| numpy.set_printoptions | - |
| numpy.set_string_function | - |
| numpy.setbufsize | - |
| numpy.setdiff1d | - |
| numpy.seterr | - |
| numpy.seterrcall | - |
| numpy.seterrobj | - |
| numpy.setxor1d | - |
| numpy.shape | - |
| numpy.shares_memory | cupy.shares_memory |
| numpy.show_config | cupy.show_config |
| numpy.sign | *cupy.sign* |
| numpy.signbit | *cupy.signbit* |
| numpy.sin | *cupy.sin* |
| numpy.sinc | *cupy.sinc* |
| numpy.sinh | *cupy.sinh* |
| numpy.size | cupy.size |
| numpy.sometrue | - |

Continued on next page

Table 98 – continued from previous page

| NumPy | CuPy |
|---|---|
| numpy.sort | *cupy.sort* |
| numpy.sort_complex | - |
| numpy.source | - |
| numpy.spacing | - |
| numpy.split | *cupy.split* |
| numpy.sqrt | *cupy.sqrt* |
| numpy.square | *cupy.square* |
| numpy.squeeze | *cupy.squeeze* |
| numpy.stack | *cupy.stack* |
| numpy.std | *cupy.std* |
| numpy.subtract | *cupy.subtract* |
| numpy.sum | *cupy.sum* |
| numpy.swapaxes | *cupy.swapaxes* |
| numpy.take | *cupy.take* |
| numpy.take_along_axis | *cupy.take_along_axis* |
| numpy.tan | *cupy.tan* |
| numpy.tanh | *cupy.tanh* |
| numpy.tensordot | *cupy.tensordot* |
| numpy.tile | *cupy.tile* |
| numpy.trace | *cupy.trace* |
| numpy.transpose | *cupy.transpose* |
| numpy.trapz | - |
| numpy.tri | *cupy.tri* |
| numpy.tril | *cupy.tril* |
| numpy.tril_indices | - |
| numpy.tril_indices_from | - |
| numpy.trim_zeros | - |
| numpy.triu | *cupy.triu* |
| numpy.triu_indices | - |
| numpy.triu_indices_from | - |
| numpy.true_divide | *cupy.true_divide* |
| numpy.trunc | *cupy.trunc* |
| numpy.typename | cupy.typename (*alias of* numpy.typename) |
| numpy.union1d | - |
| numpy.unique | *cupy.unique* |
| numpy.unpackbits | *cupy.unpackbits* |
| numpy.unravel_index | *cupy.unravel_index* |
| numpy.unwrap | *cupy.unwrap* |
| numpy.vander | - |
| numpy.var | *cupy.var* |
| numpy.vdot | *cupy.vdot* |
| numpy.vsplit | *cupy.vsplit* |
| numpy.vstack | *cupy.vstack* |
| numpy.where | *cupy.where* |
| numpy.who | - |
| numpy.zeros | *cupy.zeros* |
| numpy.zeros_like | *cupy.zeros_like* |

### Multi-Dimensional Array

| NumPy | CuPy |
|---|---|
| numpy.ndarray.all() | *cupy.ndarray.all()* |
| numpy.ndarray.any() | *cupy.ndarray.any()* |
| numpy.ndarray.argmax() | *cupy.ndarray.argmax()* |
| numpy.ndarray.argmin() | *cupy.ndarray.argmin()* |
| numpy.ndarray.argpartition() | *cupy.ndarray.argpartition()* |
| numpy.ndarray.argsort() | *cupy.ndarray.argsort()* |
| numpy.ndarray.astype() | *cupy.ndarray.astype()* |
| numpy.ndarray.byteswap() | - |
| numpy.ndarray.choose() | *cupy.ndarray.choose()* |
| numpy.ndarray.clip() | *cupy.ndarray.clip()* |
| numpy.ndarray.compress() | - |
| numpy.ndarray.conj() | *cupy.ndarray.conj()* |
| numpy.ndarray.conjugate() | - |
| numpy.ndarray.copy() | *cupy.ndarray.copy()* |
| numpy.ndarray.cumprod() | *cupy.ndarray.cumprod()* |
| numpy.ndarray.cumsum() | *cupy.ndarray.cumsum()* |
| numpy.ndarray.diagonal() | *cupy.ndarray.diagonal()* |
| numpy.ndarray.dot() | *cupy.ndarray.dot()* |
| numpy.ndarray.dump() | *cupy.ndarray.dump()* |
| numpy.ndarray.dumps() | *cupy.ndarray.dumps()* |
| numpy.ndarray.fill() | *cupy.ndarray.fill()* |
| numpy.ndarray.flatten() | *cupy.ndarray.flatten()* |
| numpy.ndarray.getfield() | - |
| numpy.ndarray.item() | *cupy.ndarray.item()* |
| numpy.ndarray.itemset() | - |
| numpy.ndarray.max() | *cupy.ndarray.max()* |
| numpy.ndarray.mean() | *cupy.ndarray.mean()* |
| numpy.ndarray.min() | *cupy.ndarray.min()* |
| numpy.ndarray.newbyteorder() | - |
| numpy.ndarray.nonzero() | *cupy.ndarray.nonzero()* |
| numpy.ndarray.partition() | *cupy.ndarray.partition()* |
| numpy.ndarray.prod() | *cupy.ndarray.prod()* |
| numpy.ndarray.ptp() | - |
| numpy.ndarray.put() | *cupy.ndarray.put()* |
| numpy.ndarray.ravel() | *cupy.ndarray.ravel()* |
| numpy.ndarray.repeat() | *cupy.ndarray.repeat()* |
| numpy.ndarray.reshape() | *cupy.ndarray.reshape()* |
| numpy.ndarray.resize() | - |
| numpy.ndarray.round() | *cupy.ndarray.round()* |
| numpy.ndarray.searchsorted() | - |
| numpy.ndarray.setfield() | - |
| numpy.ndarray.setflags() | - |
| numpy.ndarray.sort() | *cupy.ndarray.sort()* |
| numpy.ndarray.squeeze() | *cupy.ndarray.squeeze()* |
| numpy.ndarray.std() | *cupy.ndarray.std()* |
| numpy.ndarray.sum() | *cupy.ndarray.sum()* |
| numpy.ndarray.swapaxes() | *cupy.ndarray.swapaxes()* |

Continued on next page

Table 99 – continued from previous page

| NumPy | CuPy |
|---|---|
| numpy.ndarray.take() | *cupy.ndarray.take()* |
| numpy.ndarray.tobytes() | *cupy.ndarray.tobytes()* |
| numpy.ndarray.tofile() | *cupy.ndarray.tofile()* |
| numpy.ndarray.tolist() | *cupy.ndarray.tolist()* |
| numpy.ndarray.tostring() | - |
| numpy.ndarray.trace() | *cupy.ndarray.trace()* |
| numpy.ndarray.transpose() | *cupy.ndarray.transpose()* |
| numpy.ndarray.var() | *cupy.ndarray.var()* |
| numpy.ndarray.view() | *cupy.ndarray.view()* |

### Linear Algebra

| NumPy | CuPy |
|---|---|
| numpy.linalg.cholesky | *cupy.linalg.cholesky* |
| numpy.linalg.cond | - |
| numpy.linalg.det | *cupy.linalg.det* |
| numpy.linalg.eig | - |
| numpy.linalg.eigh | *cupy.linalg.eigh* |
| numpy.linalg.eigvals | - |
| numpy.linalg.eigvalsh | *cupy.linalg.eigvalsh* |
| numpy.linalg.inv | *cupy.linalg.inv* |
| numpy.linalg.lstsq | *cupy.linalg.lstsq* |
| numpy.linalg.matrix_power | *cupy.linalg.matrix_power* |
| numpy.linalg.matrix_rank | *cupy.linalg.matrix_rank* |
| numpy.linalg.multi_dot | - |
| numpy.linalg.norm | *cupy.linalg.norm* |
| numpy.linalg.pinv | *cupy.linalg.pinv* |
| numpy.linalg.qr | *cupy.linalg.qr* |
| numpy.linalg.slogdet | *cupy.linalg.slogdet* |
| numpy.linalg.solve | *cupy.linalg.solve* |
| numpy.linalg.svd | *cupy.linalg.svd* |
| numpy.linalg.tensorinv | *cupy.linalg.tensorinv* |
| numpy.linalg.tensorsolve | *cupy.linalg.tensorsolve* |

### Discrete Fourier Transform

| NumPy | CuPy |
|---|---|
| numpy.fft.fft | *cupy.fft.fft* |
| numpy.fft.fft2 | *cupy.fft.fft2* |
| numpy.fft.fftfreq | *cupy.fft.fftfreq* |
| numpy.fft.fftn | *cupy.fft.fftn* |
| numpy.fft.fftshift | *cupy.fft.fftshift* |
| numpy.fft.hfft | *cupy.fft.hfft* |
| numpy.fft.ifft | *cupy.fft.ifft* |
| numpy.fft.ifft2 | *cupy.fft.ifft2* |
| numpy.fft.ifftn | *cupy.fft.ifftn* |
| numpy.fft.ifftshift | *cupy.fft.ifftshift* |
| numpy.fft.ihfft | *cupy.fft.ihfft* |
| numpy.fft.irfft | *cupy.fft.irfft* |
| numpy.fft.irfft2 | *cupy.fft.irfft2* |
| numpy.fft.irfftn | *cupy.fft.irfftn* |
| numpy.fft.rfft | *cupy.fft.rfft* |
| numpy.fft.rfft2 | *cupy.fft.rfft2* |
| numpy.fft.rfftfreq | *cupy.fft.rfftfreq* |
| numpy.fft.rfftn | *cupy.fft.rfftn* |

### Random Sampling

| NumPy | CuPy |
|---|---|
| numpy.random.beta | *cupy.random.beta* |
| numpy.random.binomial | *cupy.random.binomial* |
| numpy.random.bytes | *cupy.random.bytes* |
| numpy.random.chisquare | *cupy.random.chisquare* |
| numpy.random.choice | *cupy.random.choice* |
| numpy.random.default_rng | - |
| numpy.random.dirichlet | *cupy.random.dirichlet* |
| numpy.random.exponential | *cupy.random.exponential* |
| numpy.random.f | *cupy.random.f* |
| numpy.random.gamma | *cupy.random.gamma* |
| numpy.random.geometric | *cupy.random.geometric* |
| numpy.random.get_state | - |
| numpy.random.gumbel | *cupy.random.gumbel* |
| numpy.random.hypergeometric | *cupy.random.hypergeometric* |
| numpy.random.laplace | *cupy.random.laplace* |
| numpy.random.logistic | *cupy.random.logistic* |
| numpy.random.lognormal | *cupy.random.lognormal* |
| numpy.random.logseries | *cupy.random.logseries* |
| numpy.random.multinomial | *cupy.random.multinomial* |
| numpy.random.multivariate_normal | *cupy.random.multivariate_normal* |
| numpy.random.negative_binomial | *cupy.random.negative_binomial* |
| numpy.random.noncentral_chisquare | *cupy.random.noncentral_chisquare* |
| numpy.random.noncentral_f | *cupy.random.noncentral_f* |
| numpy.random.normal | *cupy.random.normal* |

Continued on next page

Table 100 – continued from previous page

| NumPy | CuPy |
|---|---|
| numpy.random.pareto | *cupy.random.pareto* |
| numpy.random.permutation | *cupy.random.permutation* |
| numpy.random.poisson | *cupy.random.poisson* |
| numpy.random.power | *cupy.random.power* |
| numpy.random.rand | *cupy.random.rand* |
| numpy.random.randint | *cupy.random.randint* |
| numpy.random.randn | *cupy.random.randn* |
| numpy.random.random | *cupy.random.random* |
| numpy.random.random_integers | *cupy.random.random_integers* |
| numpy.random.random_sample | *cupy.random.random_sample* |
| numpy.random.ranf | *cupy.random.ranf* |
| numpy.random.rayleigh | *cupy.random.rayleigh* |
| numpy.random.sample | *cupy.random.sample* |
| numpy.random.seed | *cupy.random.seed* |
| numpy.random.set_state | - |
| numpy.random.shuffle | *cupy.random.shuffle* |
| numpy.random.standard_cauchy | *cupy.random.standard_cauchy* |
| numpy.random.standard_exponential | *cupy.random.standard_exponential* |
| numpy.random.standard_gamma | *cupy.random.standard_gamma* |
| numpy.random.standard_normal | *cupy.random.standard_normal* |
| numpy.random.standard_t | *cupy.random.standard_t* |
| numpy.random.triangular | *cupy.random.triangular* |
| numpy.random.uniform | *cupy.random.uniform* |
| numpy.random.vonmises | *cupy.random.vonmises* |
| numpy.random.wald | *cupy.random.wald* |
| numpy.random.weibull | *cupy.random.weibull* |
| numpy.random.zipf | *cupy.random.zipf* |

### 3.15.2 SciPy / CuPy APIs

#### Discrete Fourier Transform

| SciPy | CuPy |
|---|---|
| scipy.fftpack.cc_diff | - |
| scipy.fftpack.cs_diff | - |
| scipy.fftpack.dct | - |
| scipy.fftpack.dctn | - |
| scipy.fftpack.diff | - |
| scipy.fftpack.dst | - |
| scipy.fftpack.dstn | - |
| scipy.fftpack.fft | *cupyx.scipy.fftpack.fft* |
| scipy.fftpack.fft2 | *cupyx.scipy.fftpack.fft2* |
| scipy.fftpack.fftfreq | - |
| scipy.fftpack.fftn | *cupyx.scipy.fftpack.fftn* |
| scipy.fftpack.fftshift | - |
| scipy.fftpack.hilbert | - |
| scipy.fftpack.idct | - |
| scipy.fftpack.idctn | - |

Continued on next page

Table 101 – continued from previous page

| SciPy | CuPy |
|---|---|
| scipy.fftpack.idst | - |
| scipy.fftpack.idstn | - |
| scipy.fftpack.ifft | *cupyx.scipy.fftpack.ifft* |
| scipy.fftpack.ifft2 | *cupyx.scipy.fftpack.ifft2* |
| scipy.fftpack.ifftn | *cupyx.scipy.fftpack.ifftn* |
| scipy.fftpack.ifftshift | - |
| scipy.fftpack.ihilbert | - |
| scipy.fftpack.irfft | *cupyx.scipy.fftpack.irfft* |
| scipy.fftpack.itilbert | - |
| scipy.fftpack.next_fast_len | - |
| scipy.fftpack.rfft | *cupyx.scipy.fftpack.rfft* |
| scipy.fftpack.rfftfreq | - |
| scipy.fftpack.sc_diff | - |
| scipy.fftpack.shift | - |
| scipy.fftpack.ss_diff | - |
| scipy.fftpack.tilbert | - |

### Sparse Matrices

| SciPy | CuPy |
|---|---|
| scipy.sparse.block_diag | - |
| scipy.sparse.bmat | - |
| scipy.sparse.diags | *cupyx.scipy.sparse.diags* |
| scipy.sparse.eye | *cupyx.scipy.sparse.eye* |
| scipy.sparse.find | - |
| scipy.sparse.hstack | - |
| scipy.sparse.identity | *cupyx.scipy.sparse.identity* |
| scipy.sparse.issparse | *cupyx.scipy.sparse.issparse* |
| scipy.sparse.isspmatrix | *cupyx.scipy.sparse.isspmatrix* |
| scipy.sparse.isspmatrix_bsr | - |
| scipy.sparse.isspmatrix_coo | *cupyx.scipy.sparse.isspmatrix_coo* |
| scipy.sparse.isspmatrix_csc | *cupyx.scipy.sparse.isspmatrix_csc* |
| scipy.sparse.isspmatrix_csr | *cupyx.scipy.sparse.isspmatrix_csr* |
| scipy.sparse.isspmatrix_dia | *cupyx.scipy.sparse.isspmatrix_dia* |
| scipy.sparse.isspmatrix_dok | - |
| scipy.sparse.isspmatrix_lil | - |
| scipy.sparse.kron | - |
| scipy.sparse.kronsum | - |
| scipy.sparse.load_npz | - |
| scipy.sparse.rand | *cupyx.scipy.sparse.rand* |
| scipy.sparse.random | *cupyx.scipy.sparse.random* |
| scipy.sparse.save_npz | - |
| scipy.sparse.spdiags | *cupyx.scipy.sparse.spdiags* |
| scipy.sparse.tril | - |
| scipy.sparse.triu | - |
| scipy.sparse.vstack | - |

### Sparse Linear Algebra

| SciPy | CuPy |
|---|---|
| scipy.sparse.linalg.aslinearoperator | - |
| scipy.sparse.linalg.bicg | - |
| scipy.sparse.linalg.bicgstab | - |
| scipy.sparse.linalg.cg | - |
| scipy.sparse.linalg.cgs | - |
| scipy.sparse.linalg.eigs | - |
| scipy.sparse.linalg.eigsh | - |
| scipy.sparse.linalg.expm | - |
| scipy.sparse.linalg.expm_multiply | - |
| scipy.sparse.linalg.factorized | - |
| scipy.sparse.linalg.gcrotmk | - |
| scipy.sparse.linalg.gmres | - |
| scipy.sparse.linalg.inv | - |
| scipy.sparse.linalg.lgmres | - |
| scipy.sparse.linalg.lobpcg | - |
| scipy.sparse.linalg.lsmr | - |
| scipy.sparse.linalg.lsqr | *cupyx.scipy.sparse.linalg.lsqr* |
| scipy.sparse.linalg.minres | - |
| scipy.sparse.linalg.norm | - |
| scipy.sparse.linalg.onenormest | - |
| scipy.sparse.linalg.qmr | - |
| scipy.sparse.linalg.spilu | - |
| scipy.sparse.linalg.splu | - |
| scipy.sparse.linalg.spsolve | - |
| scipy.sparse.linalg.spsolve_triangular | - |
| scipy.sparse.linalg.svds | - |
| scipy.sparse.linalg.use_solver | - |

### Advanced Linear Algebra

| SciPy | CuPy |
|---|---|
| scipy.linalg.block_diag | - |
| scipy.linalg.cdf2rdf | - |
| scipy.linalg.cho_factor | - |
| scipy.linalg.cho_solve | - |
| scipy.linalg.cho_solve_banded | - |
| scipy.linalg.cholesky_banded | - |
| scipy.linalg.circulant | - |
| scipy.linalg.clarkson_woodruff_transform | - |
| scipy.linalg.companion | - |
| scipy.linalg.coshm | - |
| scipy.linalg.cosm | - |
| scipy.linalg.dft | - |
| scipy.linalg.diagsvd | - |
| scipy.linalg.eig_banded | - |
| scipy.linalg.eigh_tridiagonal | - |

Continued on next page

Table 102 – continued from previous page

| SciPy | CuPy |
|---|---|
| scipy.linalg.eigvals_banded | - |
| scipy.linalg.eigvalsh_tridiagonal | - |
| scipy.linalg.expm | - |
| scipy.linalg.expm_cond | - |
| scipy.linalg.expm_frechet | - |
| scipy.linalg.fiedler | - |
| scipy.linalg.fiedler_companion | - |
| scipy.linalg.find_best_blas_type | - |
| scipy.linalg.fractional_matrix_power | - |
| scipy.linalg.funm | - |
| scipy.linalg.get_blas_funcs | - |
| scipy.linalg.get_lapack_funcs | - |
| scipy.linalg.hadamard | - |
| scipy.linalg.hankel | - |
| scipy.linalg.helmert | - |
| scipy.linalg.hessenberg | - |
| scipy.linalg.hilbert | - |
| scipy.linalg.invhilbert | - |
| scipy.linalg.invpascal | - |
| scipy.linalg.kron | - |
| scipy.linalg.ldl | - |
| scipy.linalg.leslie | - |
| scipy.linalg.logm | - |
| scipy.linalg.lu | - |
| scipy.linalg.lu_factor | *cupyx.scipy.linalg.lu_factor* |
| scipy.linalg.lu_solve | *cupyx.scipy.linalg.lu_solve* |
| scipy.linalg.matrix_balance | - |
| scipy.linalg.null_space | - |
| scipy.linalg.ordqz | - |
| scipy.linalg.orth | - |
| scipy.linalg.orthogonal_procrustes | - |
| scipy.linalg.pascal | - |
| scipy.linalg.pinv2 | - |
| scipy.linalg.pinvh | - |
| scipy.linalg.polar | - |
| scipy.linalg.qr_delete | - |
| scipy.linalg.qr_insert | - |
| scipy.linalg.qr_multiply | - |
| scipy.linalg.qr_update | - |
| scipy.linalg.qz | - |
| scipy.linalg.rq | - |
| scipy.linalg.rsf2csf | - |
| scipy.linalg.schur | - |
| scipy.linalg.signm | - |
| scipy.linalg.sinhm | - |
| scipy.linalg.sinm | - |
| scipy.linalg.solve_banded | - |
| scipy.linalg.solve_circulant | - |
| scipy.linalg.solve_continuous_are | - |

Continued on next page

Table 102 – continued from previous page

| SciPy | CuPy |
|---|---|
| scipy.linalg.solve_continuous_lyapunov | - |
| scipy.linalg.solve_discrete_are | - |
| scipy.linalg.solve_discrete_lyapunov | - |
| scipy.linalg.solve_lyapunov | - |
| scipy.linalg.solve_sylvester | - |
| scipy.linalg.solve_toeplitz | - |
| scipy.linalg.solve_triangular | *cupyx.scipy.linalg.solve_triangular* |
| scipy.linalg.solveh_banded | - |
| scipy.linalg.sqrtm | - |
| scipy.linalg.subspace_angles | - |
| scipy.linalg.svdvals | - |
| scipy.linalg.tanhm | - |
| scipy.linalg.tanm | - |
| scipy.linalg.toeplitz | - |
| scipy.linalg.tri | - |
| scipy.linalg.tril | - |
| scipy.linalg.triu | - |

### Multidimensional Image Processing

| SciPy | CuPy |
|---|---|
| scipy.ndimage.affine_transform | *cupyx.scipy.ndimage.affine_transform* |
| scipy.ndimage.binary_closing | - |
| scipy.ndimage.binary_dilation | - |
| scipy.ndimage.binary_erosion | - |
| scipy.ndimage.binary_fill_holes | - |
| scipy.ndimage.binary_hit_or_miss | - |
| scipy.ndimage.binary_opening | - |
| scipy.ndimage.binary_propagation | - |
| scipy.ndimage.black_tophat | - |
| scipy.ndimage.center_of_mass | - |
| scipy.ndimage.convolve | *cupyx.scipy.ndimage.convolve* |
| scipy.ndimage.convolve1d | - |
| scipy.ndimage.correlate | *cupyx.scipy.ndimage.correlate* |
| scipy.ndimage.correlate1d | - |
| scipy.ndimage.distance_transform_bf | - |
| scipy.ndimage.distance_transform_cdt | - |
| scipy.ndimage.distance_transform_edt | - |
| scipy.ndimage.extrema | - |
| scipy.ndimage.find_objects | - |
| scipy.ndimage.fourier_ellipsoid | - |
| scipy.ndimage.fourier_gaussian | - |
| scipy.ndimage.fourier_shift | - |
| scipy.ndimage.fourier_uniform | - |
| scipy.ndimage.gaussian_filter | - |
| scipy.ndimage.gaussian_filter1d | - |
| scipy.ndimage.gaussian_gradient_magnitude | - |
| scipy.ndimage.gaussian_laplace | - |

Continued on next page

Table 103 – continued from previous page

| SciPy | CuPy |
|---|---|
| scipy.ndimage.generate_binary_structure | - |
| scipy.ndimage.generic_filter | - |
| scipy.ndimage.generic_filter1d | - |
| scipy.ndimage.generic_gradient_magnitude | - |
| scipy.ndimage.generic_laplace | - |
| scipy.ndimage.geometric_transform | - |
| scipy.ndimage.grey_closing | - |
| scipy.ndimage.grey_dilation | - |
| scipy.ndimage.grey_erosion | - |
| scipy.ndimage.grey_opening | - |
| scipy.ndimage.histogram | - |
| scipy.ndimage.iterate_structure | - |
| scipy.ndimage.label | - |
| scipy.ndimage.labeled_comprehension | - |
| scipy.ndimage.laplace | - |
| scipy.ndimage.map_coordinates | *cupyx.scipy.ndimage.map_coordinates* |
| scipy.ndimage.maximum | - |
| scipy.ndimage.maximum_filter | - |
| scipy.ndimage.maximum_filter1d | - |
| scipy.ndimage.maximum_position | - |
| scipy.ndimage.mean | - |
| scipy.ndimage.median | - |
| scipy.ndimage.median_filter | - |
| scipy.ndimage.minimum | - |
| scipy.ndimage.minimum_filter | - |
| scipy.ndimage.minimum_filter1d | - |
| scipy.ndimage.minimum_position | - |
| scipy.ndimage.morphological_gradient | - |
| scipy.ndimage.morphological_laplace | - |
| scipy.ndimage.percentile_filter | - |
| scipy.ndimage.prewitt | - |
| scipy.ndimage.rank_filter | - |
| scipy.ndimage.rotate | *cupyx.scipy.ndimage.rotate* |
| scipy.ndimage.shift | *cupyx.scipy.ndimage.shift* |
| scipy.ndimage.sobel | - |
| scipy.ndimage.spline_filter | - |
| scipy.ndimage.spline_filter1d | - |
| scipy.ndimage.standard_deviation | - |
| scipy.ndimage.sum | - |
| scipy.ndimage.uniform_filter | - |
| scipy.ndimage.uniform_filter1d | - |
| scipy.ndimage.variance | - |
| scipy.ndimage.watershed_ift | - |
| scipy.ndimage.white_tophat | - |
| scipy.ndimage.zoom | *cupyx.scipy.ndimage.zoom* |

### Special Functions

| SciPy | CuPy |
|---|---|
| scipy.special.agm | - |
| scipy.special.ai_zeros | - |
| scipy.special.airy | - |
| scipy.special.airye | - |
| scipy.special.assoc_laguerre | - |
| scipy.special.bdtr | - |
| scipy.special.bdtrc | - |
| scipy.special.bdtri | - |
| scipy.special.bdtrik | - |
| scipy.special.bdtrin | - |
| scipy.special.bei | - |
| scipy.special.bei_zeros | - |
| scipy.special.beip | - |
| scipy.special.beip_zeros | - |
| scipy.special.ber | - |
| scipy.special.ber_zeros | - |
| scipy.special.bernoulli | - |
| scipy.special.berp | - |
| scipy.special.berp_zeros | - |
| scipy.special.besselpoly | - |
| scipy.special.beta | - |
| scipy.special.betainc | - |
| scipy.special.betaincinv | - |
| scipy.special.betaln | - |
| scipy.special.bi_zeros | - |
| scipy.special.binom | - |
| scipy.special.boxcox | - |
| scipy.special.boxcox1p | - |
| scipy.special.btdtr | - |
| scipy.special.btdtri | - |
| scipy.special.btdtria | - |
| scipy.special.btdtrib | - |
| scipy.special.c_roots | - |
| scipy.special.cbrt | - |
| scipy.special.cg_roots | - |
| scipy.special.chdtr | - |
| scipy.special.chdtrc | - |
| scipy.special.chdtri | - |
| scipy.special.chdtriv | - |
| scipy.special.chebyc | - |
| scipy.special.chebys | - |
| scipy.special.chebyt | - |
| scipy.special.chebyu | - |
| scipy.special.chndtr | - |
| scipy.special.chndtridf | - |
| scipy.special.chndtrinc | - |
| scipy.special.chndtrix | - |
| scipy.special.clpmn | - |
| scipy.special.comb | - |
| scipy.special.cosdg | - |

Continued on next page

Table 104 – continued from previous page

| SciPy | CuPy |
|---|---|
| scipy.special.cosm1 | - |
| scipy.special.cotdg | - |
| scipy.special.dawsn | - |
| scipy.special.digamma | *cupyx.scipy.special.digamma* |
| scipy.special.diric | - |
| scipy.special.ellip_harm | - |
| scipy.special.ellip_harm_2 | - |
| scipy.special.ellip_normal | - |
| scipy.special.ellipe | - |
| scipy.special.ellipeinc | - |
| scipy.special.ellipj | - |
| scipy.special.ellipk | - |
| scipy.special.ellipkinc | - |
| scipy.special.ellipkm1 | - |
| scipy.special.entr | - |
| scipy.special.erf | *cupyx.scipy.special.erf* |
| scipy.special.erf_zeros | - |
| scipy.special.erfc | *cupyx.scipy.special.erfc* |
| scipy.special.erfcinv | *cupyx.scipy.special.erfcinv* |
| scipy.special.erfcx | *cupyx.scipy.special.erfcx* |
| scipy.special.erfi | - |
| scipy.special.erfinv | *cupyx.scipy.special.erfinv* |
| scipy.special.euler | - |
| scipy.special.eval_chebyc | - |
| scipy.special.eval_chebys | - |
| scipy.special.eval_chebyt | - |
| scipy.special.eval_chebyu | - |
| scipy.special.eval_gegenbauer | - |
| scipy.special.eval_genlaguerre | - |
| scipy.special.eval_hermite | - |
| scipy.special.eval_hermitenorm | - |
| scipy.special.eval_jacobi | - |
| scipy.special.eval_laguerre | - |
| scipy.special.eval_legendre | - |
| scipy.special.eval_sh_chebyt | - |
| scipy.special.eval_sh_chebyu | - |
| scipy.special.eval_sh_jacobi | - |
| scipy.special.eval_sh_legendre | - |
| scipy.special.exp1 | - |
| scipy.special.exp10 | - |
| scipy.special.exp2 | - |
| scipy.special.expi | - |
| scipy.special.expit | - |
| scipy.special.expm1 | - |
| scipy.special.expn | - |
| scipy.special.exprel | - |
| scipy.special.factorial | - |
| scipy.special.factorial2 | - |
| scipy.special.factorialk | - |

Continued on next page

Table 104 – continued from previous page

| SciPy | CuPy |
|---|---|
| scipy.special.fdtr | - |
| scipy.special.fdtrc | - |
| scipy.special.fdtri | - |
| scipy.special.fdtridfd | - |
| scipy.special.fresnel | - |
| scipy.special.fresnel_zeros | - |
| scipy.special.fresnelc_zeros | - |
| scipy.special.fresnels_zeros | - |
| scipy.special.gamma | *cupyx.scipy.special.gamma* |
| scipy.special.gammainc | - |
| scipy.special.gammaincc | - |
| scipy.special.gammainccinv | - |
| scipy.special.gammaincinv | - |
| scipy.special.gammaln | *cupyx.scipy.special.gammaln* |
| scipy.special.gammasgn | - |
| scipy.special.gdtr | - |
| scipy.special.gdtrc | - |
| scipy.special.gdtria | - |
| scipy.special.gdtrib | - |
| scipy.special.gdtrix | - |
| scipy.special.gegenbauer | - |
| scipy.special.genlaguerre | - |
| scipy.special.geterr | - |
| scipy.special.h1vp | - |
| scipy.special.h2vp | - |
| scipy.special.h_roots | - |
| scipy.special.hankel1 | - |
| scipy.special.hankel1e | - |
| scipy.special.hankel2 | - |
| scipy.special.hankel2e | - |
| scipy.special.he_roots | - |
| scipy.special.hermite | - |
| scipy.special.hermitenorm | - |
| scipy.special.huber | - |
| scipy.special.hyp0f1 | - |
| scipy.special.hyp1f1 | - |
| scipy.special.hyp2f1 | - |
| scipy.special.hyperu | - |
| scipy.special.i0 | *cupyx.scipy.special.i0* |
| scipy.special.i0e | - |
| scipy.special.i1 | *cupyx.scipy.special.i1* |
| scipy.special.i1e | - |
| scipy.special.inv_boxcox | - |
| scipy.special.inv_boxcox1p | - |
| scipy.special.it2i0k0 | - |
| scipy.special.it2j0y0 | - |
| scipy.special.it2struve0 | - |
| scipy.special.itairy | - |
| scipy.special.iti0k0 | - |

Continued on next page

Table 104 – continued from previous page

| SciPy | CuPy |
|---|---|
| scipy.special.itj0y0 | - |
| scipy.special.itmodstruve0 | - |
| scipy.special.itstruve0 | - |
| scipy.special.iv | - |
| scipy.special.ive | - |
| scipy.special.ivp | - |
| scipy.special.j0 | *cupyx.scipy.special.j0* |
| scipy.special.j1 | *cupyx.scipy.special.j1* |
| scipy.special.j_roots | - |
| scipy.special.jacobi | - |
| scipy.special.jn | - |
| scipy.special.jn_zeros | - |
| scipy.special.jnjnp_zeros | - |
| scipy.special.jnp_zeros | - |
| scipy.special.jnyn_zeros | - |
| scipy.special.js_roots | - |
| scipy.special.jv | - |
| scipy.special.jve | - |
| scipy.special.jvp | - |
| scipy.special.k0 | - |
| scipy.special.k0e | - |
| scipy.special.k1 | - |
| scipy.special.k1e | - |
| scipy.special.kei | - |
| scipy.special.kei_zeros | - |
| scipy.special.keip | - |
| scipy.special.keip_zeros | - |
| scipy.special.kelvin | - |
| scipy.special.kelvin_zeros | - |
| scipy.special.ker | - |
| scipy.special.ker_zeros | - |
| scipy.special.kerp | - |
| scipy.special.kerp_zeros | - |
| scipy.special.kl_div | - |
| scipy.special.kn | - |
| scipy.special.kolmogi | - |
| scipy.special.kolmogorov | - |
| scipy.special.kv | - |
| scipy.special.kve | - |
| scipy.special.kvp | - |
| scipy.special.l_roots | - |
| scipy.special.la_roots | - |
| scipy.special.laguerre | - |
| scipy.special.lambertw | - |
| scipy.special.legendre | - |
| scipy.special.lmbda | - |
| scipy.special.log1p | - |
| scipy.special.log_ndtr | - |
| scipy.special.loggamma | - |

Continued on next page

Table 104 – continued from previous page

| SciPy | CuPy |
|---|---|
| scipy.special.logit | - |
| scipy.special.logsumexp | - |
| scipy.special.lpmn | - |
| scipy.special.lpmv | - |
| scipy.special.lpn | - |
| scipy.special.lqmn | - |
| scipy.special.lqn | - |
| scipy.special.mathieu_a | - |
| scipy.special.mathieu_b | - |
| scipy.special.mathieu_cem | - |
| scipy.special.mathieu_even_coef | - |
| scipy.special.mathieu_modcem1 | - |
| scipy.special.mathieu_modcem2 | - |
| scipy.special.mathieu_modsem1 | - |
| scipy.special.mathieu_modsem2 | - |
| scipy.special.mathieu_odd_coef | - |
| scipy.special.mathieu_sem | - |
| scipy.special.modfresnelm | - |
| scipy.special.modfresnelp | - |
| scipy.special.modstruve | - |
| scipy.special.multigammaln | - |
| scipy.special.nbdtr | - |
| scipy.special.nbdtrc | - |
| scipy.special.nbdtri | - |
| scipy.special.nbdtrik | - |
| scipy.special.nbdtrin | - |
| scipy.special.ncfdtr | - |
| scipy.special.ncfdtri | - |
| scipy.special.ncfdtridfd | - |
| scipy.special.ncfdtridfn | - |
| scipy.special.ncfdtrinc | - |
| scipy.special.nctdtr | - |
| scipy.special.nctdtridf | - |
| scipy.special.nctdtrinc | - |
| scipy.special.nctdtrit | - |
| scipy.special.ndtr | *cupyx.scipy.special.ndtr* |
| scipy.special.ndtri | - |
| scipy.special.nrdtrimn | - |
| scipy.special.nrdtrisd | - |
| scipy.special.obl_ang1 | - |
| scipy.special.obl_ang1_cv | - |
| scipy.special.obl_cv | - |
| scipy.special.obl_cv_seq | - |
| scipy.special.obl_rad1 | - |
| scipy.special.obl_rad1_cv | - |
| scipy.special.obl_rad2 | - |
| scipy.special.obl_rad2_cv | - |
| scipy.special.owens_t | - |
| scipy.special.p_roots | - |

Continued on next page

Table 104 – continued from previous page

| SciPy | CuPy |
|---|---|
| scipy.special.pbdn_seq | - |
| scipy.special.pbdv | - |
| scipy.special.pbdv_seq | - |
| scipy.special.pbvv | - |
| scipy.special.pbvv_seq | - |
| scipy.special.pbwa | - |
| scipy.special.pdtr | - |
| scipy.special.pdtrc | - |
| scipy.special.pdtri | - |
| scipy.special.pdtrik | - |
| scipy.special.perm | - |
| scipy.special.poch | - |
| scipy.special.polygamma | *cupyx.scipy.special.polygamma* |
| scipy.special.pro_ang1 | - |
| scipy.special.pro_ang1_cv | - |
| scipy.special.pro_cv | - |
| scipy.special.pro_cv_seq | - |
| scipy.special.pro_rad1 | - |
| scipy.special.pro_rad1_cv | - |
| scipy.special.pro_rad2 | - |
| scipy.special.pro_rad2_cv | - |
| scipy.special.ps_roots | - |
| scipy.special.pseudo_huber | - |
| scipy.special.psi | - |
| scipy.special.radian | - |
| scipy.special.rel_entr | - |
| scipy.special.rgamma | - |
| scipy.special.riccati_jn | - |
| scipy.special.riccati_yn | - |
| scipy.special.roots_chebyc | - |
| scipy.special.roots_chebys | - |
| scipy.special.roots_chebyt | - |
| scipy.special.roots_chebyu | - |
| scipy.special.roots_gegenbauer | - |
| scipy.special.roots_genlaguerre | - |
| scipy.special.roots_hermite | - |
| scipy.special.roots_hermitenorm | - |
| scipy.special.roots_jacobi | - |
| scipy.special.roots_laguerre | - |
| scipy.special.roots_legendre | - |
| scipy.special.roots_sh_chebyt | - |
| scipy.special.roots_sh_chebyu | - |
| scipy.special.roots_sh_jacobi | - |
| scipy.special.roots_sh_legendre | - |
| scipy.special.round | - |
| scipy.special.s_roots | - |
| scipy.special.seterr | - |
| scipy.special.sh_chebyt | - |
| scipy.special.sh_chebyu | - |

Continued on next page

Table 104 – continued from previous page

| SciPy | CuPy |
|---|---|
| scipy.special.sh_jacobi | - |
| scipy.special.sh_legendre | - |
| scipy.special.shichi | - |
| scipy.special.sici | - |
| scipy.special.sinc | - |
| scipy.special.sindg | - |
| scipy.special.smirnov | - |
| scipy.special.smirnovi | - |
| scipy.special.softmax | - |
| scipy.special.spence | - |
| scipy.special.sph_harm | - |
| scipy.special.spherical_in | - |
| scipy.special.spherical_jn | - |
| scipy.special.spherical_kn | - |
| scipy.special.spherical_yn | - |
| scipy.special.stdtr | - |
| scipy.special.stdtridf | - |
| scipy.special.stdtrit | - |
| scipy.special.struve | - |
| scipy.special.t_roots | - |
| scipy.special.tandg | - |
| scipy.special.tklmbda | - |
| scipy.special.ts_roots | - |
| scipy.special.u_roots | - |
| scipy.special.us_roots | - |
| scipy.special.voigt_profile | - |
| scipy.special.wofz | - |
| scipy.special.wrightomega | - |
| scipy.special.xlog1py | - |
| scipy.special.xlogy | - |
| scipy.special.y0 | *cupyx.scipy.special.y0* |
| scipy.special.y0_zeros | - |
| scipy.special.y1 | *cupyx.scipy.special.y1* |
| scipy.special.y1_zeros | - |
| scipy.special.y1p_zeros | - |
| scipy.special.yn | - |
| scipy.special.yn_zeros | - |
| scipy.special.ynp_zeros | - |
| scipy.special.yv | - |
| scipy.special.yve | - |
| scipy.special.yvp | - |
| scipy.special.zeta | *cupyx.scipy.special.zeta* |
| scipy.special.zetac | - |

CHAPTER 4

# API Compatibility Policy

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

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

## 4.1 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.

## 4.2 Processes to Break Backward Compatibilities

### 4.2.1 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.

### 4.2.2 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.

## 4.3 Supported Backward Compatibility

This section defines backward compatibilities that minor updates must maintain.

### 4.3.1 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.

### 4.3.2 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.**

### 4.3.3 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.

---

### 4.3.4 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.

### 4.3.5 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.

### 4.3.6 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.

## 4.4 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 5

# 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.

## 5.1 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.

## 5.2 Development Cycle

This section explains the development process of CuPy. Before contributing to CuPy, it is strongly recommended to understand the development cycle.

### 5.2.1 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 prelease 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.

### 5.2.2 Release Cycle

The first one is the track of **stable versions**, which is a series of revision updates for the latest major version. The second one is the track of **development versions**, which is a series of pre-releases for the upcoming major version.

Consider that `X.0.0` is the latest major version and `Y.0.0`, `Z.0.0` are the succeeding major versions. Then, the timeline of the updates is depicted by the following table.

| Date | ver X | ver Y | ver Z |
|---|---|---|---|
| 0 weeks | X.0.0rc1 | – | – |
| 4 weeks | X.0.0 | Y.0.0a1 | – |
| 8 weeks | X.1.0* | Y.0.0b1 | – |
| 12 weeks | X.2.0* | Y.0.0rc1 | – |
| 16 weeks | – | Y.0.0 | Z.0.0a1 |

(* These might be revision releases)

The dates shown in the left-most column are relative to the release of `X.0.0rc1`. In particular, each revision/minor release is made four weeks after the previous one of the same major version, and the pre-release of the upcoming major version is made at the same time. Whether these releases are revision or minor is determined based on the contents of each update.

Note that there are only three stable releases for the versions `X.x.x`. During the parallel development of `Y.0.0` and `Z.0.0a1`, the version `Y` is treated as an **almost-stable version** and `Z` is treated as a development version.

If there is a critical bug found in `X.x.x` after stopping the development of version `X`, we may release a hot-fix for this version at any time.

We create a milestone for each upcoming release at GitHub. The GitHub milestone is basically used for collecting the issues and PRs resolved in the release.

### 5.2.3 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.*

### 5.2.4 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!

## 5.3 Issues and Pull Requests

In this section, we explain how to file issues and send pull requests (PRs).

### 5.3.1 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.**

### 5.3.2 How to File an Issue

On registering an issue, write precise explanations on how you want CuPy to be. Bug reports must include necessary and sufficient conditions to reproduce the bugs. Feature requests must include **what** you want to do (and **why** you want to do, if needed) with CuPy. You can contain your thoughts on **how** to realize it into the feature requests, though **what** part is most important for discussions.

> **Warning:** If you have a question on usages of CuPy, it is highly recommended to send a post to CuPy User Group instead of the issue tracker. The issue tracker is not a place to share knowledge on practices. We may suggest these places and immediately close how-to question issues.

### 5.3.3 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.

## 5.4 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.

## 5.5 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
```

### 5.5.1 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
```

**Note:** It's not officially supported, but you can use ccache to reduce compilation time. On Ubuntu 16.04, you can set up as follows:

```
$ sudo apt-get install ccache
$ export PATH=/usr/lib/ccache:$PATH
```

See ccache for details.

If you want to use ccache for nvcc, please install ccache v3.3 or later. You also need to set environment variable `NVCC='ccache nvcc'`.

Once Cython modules are built, you can run unit tests by running the following command at the repository root:

```
$ python -m pytest
```

CUDA must be installed to run unit tests.

Some GPU tests require cuDNN to run. In order to skip unit tests that require cuDNN, specify `-m='not cudnn'` option:

```
$ python -m pytest path/to/your/test.py -m='not cudnn'
```

Some GPU tests involve multiple GPUs. If you want to run GPU tests with insufficient number of GPUs, specify the number of available GPUs to `CUPY_TEST_GPU_LIMIT`. For example, if you have only one GPU, launch `pytest` by the following command to skip multi-GPU tests:

```
$ export CUPY_TEST_GPU_LIMIT=1
$ python -m pytest path/to/gpu/test.py
```

Following this naming convention, you can run all the tests by running the following command at the repository root:

```
$ python -m pytest
```

Or you can also specify a root directory to search test scripts from:

```
$ python -m pytest tests/cupy_tests     # to just run tests of CuPy
$ python -m pytest tests/install_tests  # to just run tests of installation modules
```

If you modify the code related to existing unit tests, you must run appropriate commands.

### 5.5.2 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`.

### 5.5.3 How to Write Tests

There are many examples of unit tests under the tests directory, so reading some of them is a good and recommended way to learn how to write tests for CuPy. They simply use the `unittest` package of the standard library, while some tests are using utilities from *`cupy.testing`*.

In addition to the *Coding Guidelines* mentioned above, the following rules are applied to the test code:

- All test classes must inherit from `unittest.TestCase`.
- Use `unittest` features to write tests, except for the following cases:
  - Use `assert` statement instead of `self.assert*` methods (e.g., write `assert x == 1` instead of `self.assertEqual(x, 1)`).
  - Use `with pytest.raises(...):` instead of `with self.assertRaises(...):`.

---

**Note:** We are incrementally applying the above style. Some existing tests may be using the old style (`self.assertRaises`, etc.), but all newly written tests should follow the above style.

---

Even if your patch includes GPU-related code, your tests should not fail without GPU capability. Test functions that require CUDA must be tagged by the `cupy.testing.attr.gpu`:

```
import unittest
from cupy.testing import attr

class TestMyFunc(unittest.TestCase):
    ...
```

(continues on next page)

(continued from previous page)

```
    @attr.gpu
    def test_my_gpu_func(self):
        ...
```

The functions tagged by the `gpu` decorator are skipped if `CUPY_TEST_GPU_LIMIT=0` environment variable is set. We also have the `cupy.testing.attr.cudnn` decorator to let `pytest` know that the test depends on cuDNN. The test functions decorated by `cudnn` are skipped if `-m='not cudnn'` is given.

The test functions decorated by `gpu` must not depend on multiple GPUs. In order to write tests for multiple GPUs, use `cupy.testing.attr.multi_gpu()` decorators instead:

```
import unittest
from cupy.testing import attr

class TestMyFunc(unittest.TestCase):
    ...

    @attr.multi_gpu(2)  # specify the number of required GPUs here
    def test_my_two_gpu_func(self):
        ...
```

If your test requires too much time, add `cupy.testing.attr.slow` decorator. The test functions decorated by `slow` are skipped if `-m='not slow'` is given:

```
import unittest
from cupy.testing import attr

class TestMyFunc(unittest.TestCase):
    ...

    @attr.slow
    def test_my_slow_func(self):
        ...
```

---

**Note:** If you want to specify more than two attributes, use `and` operator like `-m='not cudnn and not slow'`. See detail in the document of pytest.

---

Once you send a pull request, Travis-CI automatically checks if your code meets our coding guidelines described above. Since Travis-CI does not support CUDA, we cannot run unit tests automatically. The reviewing process starts after the automatic check passes. Note that reviewers will test your code without the option to check CUDA-related code.

---

**Note:** Some of numerically unstable tests might cause errors irrelevant to your changes. In such a case, we ignore the failures and go on to the review process, so do not worry about it!

---

## 5.6 Documentation

When adding a new feature to the framework, you also need to document it in the reference.

**Note:** If you are unsure about how to fix the documentation, you can submit a pull request without doing so. Reviewers will help you fix the documentation appropriately.

The documentation source is stored under docs directory and written in reStructuredText format.

To build the documentation, you need to install Sphinx:

```
$ pip install sphinx sphinx_rtd_theme
```

Then you can build the documentation in HTML format locally:

```
$ cd docs
$ make html
```

HTML files are generated under `build/html` directory. Open `index.html` with the browser and see if it is rendered as expected.

**Note:** Docstrings (documentation comments in the source code) are collected from the installed CuPy module. If you modified docstrings, make sure to install the module (e.g., using *pip install -e .*) before building the documentation.

CHAPTER 6

# Installation Guide

- *Build fails with CUDA 11.0 on Ubuntu 16.04, CentOS 6 or 7*

## 6.1 Recommended Environments

We recommend the following Linux distributions.

- Ubuntu 16.04 / 18.04 LTS (64-bit)
- CentOS 7 (64-bit)

**Note:** We are automatically testing CuPy on all the recommended environments above. We cannot guarantee that CuPy works on other environments including Windows and macOS, even if CuPy may seem to be running correctly.

## 6.2 Requirements

You need to have the following components to use CuPy.

- **NVIDIA CUDA GPU**
  - Compute Capability of the GPU must be at least 3.0.
- **CUDA Toolkit**
  - Supported Versions: 8.0, 9.0, 9.1, 9.2, 10.0, 10.1, 10.2 and 11.0.
  - If you have multiple versions of CUDA Toolkit installed, CuPy will choose one of the CUDA installations automatically. See *Working with Custom CUDA Installation* for details.
- **Python**
  - Supported Versions: 3.5.1+, 3.6.0+, 3.7.0+ and 3.8.0+.
- **NumPy**
  - Supported Versions: 1.9, 1.10, 1.11, 1.12, 1.13, 1.14, 1.15, 1.16, 1.17, 1.18 and 1.19.
  - NumPy will be installed automatically during the installation of CuPy.

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

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

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

**Note:** Python 2 is not supported in CuPy v7.x releases. Please consider migrating Python 3 or use CuPy v6.x, which is the last version that supports Python 2.

### 6.2.1 Optional Libraries

Some features in CuPy will only be enabled if the corresponding libraries are installed.

- **cuDNN (library to accelerate deep neural network computations)**
  - Supported Versions: v5, v5.1, v6, v7, v7.1, v7.2, v7.3, v7.4, v7.5, v7.6 and v8.0.
- **NCCL (library to perform collective multi-GPU / multi-node computations)**
  - Supported Versions: v1.3.4, v2, v2.1, v2.2, v2.3, v2.4, v2.5, v2.6 and v2.7.
- **cuTENSOR (library for high-performance tensor operations)**
  - Supported Versions: v1.0.0 (experimental)

## 6.3 Install CuPy

Wheels (precompiled binary packages) are available for Linux (Python 3.5 or later) and Windows (Python 3.6 or later). Package names are different depending on the CUDA version you have installed on your host.

```
(For CUDA 8.0)
$ pip install cupy-cuda80

(For CUDA 9.0)
$ pip install cupy-cuda90

(For CUDA 9.1)
$ pip install cupy-cuda91

(For CUDA 9.2)
$ pip install cupy-cuda92

(For CUDA 10.0)
$ pip install cupy-cuda100

(For CUDA 10.1)
$ pip install cupy-cuda101

(For CUDA 10.2)
$ pip install cupy-cuda102

(For CUDA 11.0)
$ pip install cupy-cuda110
```

**Note:** The latest version of cuDNN and NCCL libraries are included in these wheels except for CUDA 11.0. For CUDA 11.0, you need to manually download and install cuDNN 8.0.x. For other CUDA versions, you don't have to install them manually.

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

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

## 6.4 Install 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:

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

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:

```
$ 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.

## 6.5 Install CuPy from Source

It is recommended to use wheels 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.

When installing from source, C++ compiler such as `g++` is required. You need to install it before installing CuPy. This is typical installation method for each platform:

```
# Ubuntu 16.04
$ apt-get install g++

# CentOS 7
$ yum install gcc-c++
```

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

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

### 6.5.1 Using pip

You can install CuPy package via `pip`.

```
$ pip install cupy
```

### 6.5.2 Using Tarball

The tarball of the source tree is available via `pip download cupy` or from the release notes page. You can install CuPy from the tarball:

```
$ pip install cupy-x.x.x.tar.gz
```

You can also install the development version of CuPy from a cloned Git repository:

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

If you are using source tree downloaded from GitHub, you need to install Cython 0.28.0 or later (`pip install cython`).

## 6.6 Uninstall CuPy

Use pip to uninstall CuPy:

```
$ pip uninstall cupy
```

**Note:** When you upgrade Chainer, `pip` sometimes installs the new version without removing the old one in `site-packages`. In this case, `pip uninstall` only removes the latest one. To ensure that CuPy is completely removed, run the above command repeatedly until `pip` returns an error.

**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.

## 6.7 Upgrade 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).

## 6.8 Reinstall CuPy

If you want to reinstall CuPy, please uninstall CuPy and then install it. When reinstalling CuPy, we recommend to use `--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).

## 6.9 Run CuPy with Docker

We are providing the official Docker image. Use nvidia-docker command to run CuPy image with GPU. You can login to the environment with bash, and run the Python interpreter:

```
$ nvidia-docker run -it cupy/cupy /bin/bash
```

Or run the interpreter directly:

```
$ nvidia-docker run -it cupy/cupy /usr/bin/python
```

## 6.10 FAQ

### 6.10.1 Warning message "cuDNN is not enabled" appears when using Chainer

You failed to build CuPy with cuDNN. If you don't need cuDNN, ignore this message. Otherwise, retry to install CuPy with cuDNN.

See *Installing cuDNN and NCCL* and *pip fails to install CuPy* for details.

### 6.10.2 `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.

### 6.10.3 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 you to install it under CUDA 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
```

---

**Note:** Use full paths for the environment variables. `distutils` that is used in the setup script does not expand the home directory mark ~.

---

### 6.10.4 Working with Custom CUDA Installation

If you have installed CUDA on the non-default directory or have multiple CUDA versions installed, 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 in each directory set in `PATH` environment variable.
3. `/usr/local/cuda`

For example, you can tell CuPy to use non-default CUDA directory by `CUDA_PATH` environment variable:

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

---

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

---

### 6.10.5 Using custom `nvcc` command during installation

If you want to use a custom `nvcc` compiler (for example, to use `ccache`) to build CuPy, please set `NVCC` environment variables before installing CuPy:

```
export NVCC='ccache nvcc'
```

**Note:** During runtime, you don't need to set this environment variable since CuPy doesn't use the nvcc command.

### 6.10.6 Installation for Developers

If you are hacking CuPy source code, we recommend you to use `pip` with `-e` option for editable mode:

```
$ cd /path/to/cupy/source
$ 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).

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

If CuPy does not work at all with `CompileException`, 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`:

```
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*.

### 6.10.8 Build fails with CUDA 11.0 on Ubuntu 16.04, CentOS 6 or 7

In order to build CuPy from source with CUDA 11.0 on systems with legacy GCC (g++-5 or earlier), you need to manually set up g++-6 or later and configure `NVCC` environment variable.

On Ubuntu 16.04:

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

On CentOS 6 / 7:

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

CHAPTER 7

# [Experimental] Installation Guide for ROCm environemt



This is an experimental feature. We recommend only for advanced users to use this.

## 7.1 Recommended Environments

We recommend the following Linux distributions.

- Ubuntu 16.04 / 18.04 LTS (64-bit)

## 7.2 Requirements

You need to have the following components to use CuPy.

- GPU supported by ROCm (AMD GPUs or NVIDIA GPUs)
- **ROCm**
  - Supported Versions: ROCm 2.6+.
- Python
- NumPy

And please install ROCm libraries.

```
$ sudo apt install hipblas hipsparse rocrand rocthrust
```

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

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

## 7.3 Install CuPy from Source

It is recommended to use wheels whenever possible. However, there is currently no wheels for the ROCm environment, so you have to build it from source.

When installing from source, C++ compiler such as `g++` is required. You need to install it before installing CuPy. This is typical installation method for each platform:

```
# Ubuntu 16.04
$ apt-get install g++
```

**Note:** If you want to upgrade or downgrade the version of ROCm, you may need to reinstall CuPy after that. See *Reinstall CuPy* for details.

### 7.3.1 Using pip

You can install CuPy package via `pip`. It builds CuPy from source.

```
$ 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
```

### 7.3.2 Using Tarball

The tarball of the source tree is available via `pip download cupy` or from the release notes page. You can install CuPy from the tarball:

```
$ pip install cupy-x.x.x.tar.gz
```

You can also install the development version of CuPy from a cloned Git repository:

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

If you are using the source tree downloaded from GitHub, you need to install Cython 0.28.0 or later (`pip install cython`).

## 7.4 Uninstall CuPy

Use pip to uninstall CuPy:

```
$ pip uninstall cupy
```

**Note:** When you upgrade Chainer, `pip` sometimes installs the new version without removing the old one in `site-packages`. In this case, `pip uninstall` only removes the latest one. To ensure that CuPy is completely removed, run the above command repeatedly until `pip` returns an error.

## 7.5 Upgrade CuPy

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

```
$ 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 -U cupy
```

## 7.6 Reinstall CuPy

If you want to reinstall CuPy, please uninstall CuPy first, and then install again. When reinstalling CuPy, we recommend to use `--no-cache-dir` option as `pip` caches the previously built binaries:

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

## 7.7 FAQ

### 7.7.1 `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 to investigate the details of errors. 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., `ROCM_HOME`), you need to specify them inside `sudo` like this:

```
$ sudo ROCM_HOME=/opt/rocm 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 downgrade or upgrade it.

CHAPTER 8

# Upgrade Guide

This is a list of changes introduced in each release that users should be aware of when migrating from older versions. Most changes are carefully designed not to break existing code; however changes that may possibly break them are highlighted with a box.

## 8.1 CuPy v7

### 8.1.1 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.

## 8.2 CuPy v6

### 8.2.1 Binary Packages Ignore `LD_LIBRARY_PATH`

Prior to CuPy v6, `LD_LIBRARY_PATH` environment variable can be used to override cuDNN / NCCL libraries bundled in the binary distribution (also known as wheels). In CuPy v6, `LD_LIBRARY_PATH` will be ignored during discovery of cuDNN / NCCL; CuPy binary distributions always use libraries that comes with the package to avoid errors caused by unexpected override.

## 8.3 CuPy v5

### 8.3.1 `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.

### 8.3.2 Dropped Support for CUDA 7.0 / 7.5

CuPy v5 no longer supports CUDA 7.0 / 7.5.

### 8.3.3 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.

## 8.4 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.

### 8.4.1 Default Memory Pool

Prior to CuPy v4, memory pool was only enabled by default when CuPy is used with Chainer. In CuPy v4, memory pool is now enabled by default, even when you use CuPy without Chainer. The memory pool significantly improves the performance by mitigating the overhead of memory allocation and CPU/GPU synchronization.

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

To access the default memory pool instance, use *`get_default_memory_pool()`* and *`get_default_pinned_memory_pool()`*. You can access the statistics and free all unused memory blocks "cached" in the memory pool.

```
import cupy
a = cupy.ndarray(100, dtype=cupy.float32)
mempool = cupy.get_default_memory_pool()

# For performance, the size of actual allocation may become larger than the requested
↪array size.
print(mempool.used_bytes())    # 512
print(mempool.total_bytes())   # 512
```

(continues on next page)

(continued from previous page)

```
# Even if the array goes out of scope, its memory block is kept in the pool.
a = None
print(mempool.used_bytes())    # 0
print(mempool.total_bytes())   # 512

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

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

```
import cupy
cupy.cuda.set_allocator(None)
cupy.cuda.set_pinned_memory_allocator(None)
```

### 8.4.2 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.

### 8.4.3 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.

### 8.4.4 `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.

### 8.4.5 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.

## 8.5 CuPy v2

### 8.5.1 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.

CHAPTER 9

# License



## 9.1 NumPy

The CuPy is designed based on NumPy's API. CuPy's source code and documents contain the original NumPy ones.

- Neither the name of the NumPy Developers nor the names of any contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

## 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

# Index

## Symbols

## A

## B



## C

## D



## E

## F

## G

## H



## I



## J



## K



## L

## M

## N

## O



## P



## Q



## R

## S

## T

## U



## V



## W



## Y



## Z