CuPy Documentation

Release 4.2.0

Preferred Networks, inc. and Preferred Infrastructure, inc.

Jun 21, 2018

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)))
```

2.2 User-Defined Kernels

CuPy provides easy ways to define two types of CUDA kernels: elementwise kernels and reduction kernels. We first describe how to define and call elementwise kernels, and then describe how to define and call reduction 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.

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 `ndarray`s, 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*, *order='C'*)

Multi-dimensional array on a CUDA device.

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

Parameters

- **shape** (*tuple of ints*) – Length of axes.
- **dtype** – Data type. It must be an argument of `numpy.dtype`.
- **memptr** (`cupy.cuda.MemoryPointer`) – Pointer to the array content head.
- **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
```

(continues on next page)

(continued from previous page)

```
>>> 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
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).

`__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.

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.

See also:

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

`argpartition`(*self*, *kth*, *axis=-1*)
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*)
Returns the indices that would sort an array with stable sorting

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

Returns Array of indices that sort the array.

Return type *cupy.ndarray*

See also:

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

astype (*self*, *dtype*, *order='K'*, *casting=None*, *subok=None*, *copy=True*) → ndarray
Casts the array to given data type.

Parameters

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

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

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

See also:

`numpy.ndarray.astype()`

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

clip (*self*, *a_min=None*, *a_max=None*, *out=None*) → ndarray
Returns an array with values limited to [a_min, a_max].

See also:

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

conj (*self*) → ndarray

copy (*self*, *order='C'*) → ndarray
Returns a copy of the array.

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

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

See also:

`cupy.copy()` for full documentation, `numpy.ndarray.copy()`

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*)
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*)
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.

Returns Copy of the array on host memory.

Return type numpy.ndarray

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*)
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

See also:

numpy.nonzero()

partition (*self*, *kth*, *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()

ravel (*self*) → 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 – 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*)
Returns an array of a different shape and the same content.

See also:

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

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

See also:

cupyx.scatter_add() 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*, *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()

tofile (*self*, *fid*, *sep=''*, *format='%s'*)
Writes the array to a file.

See also:

numpy.ndarray.tolist()

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
Size of whole 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.

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)))
```

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.

Continued on next page

Table 3 – continued from previous page

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)`

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.

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)`

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.

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`

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__`(*self*, **args*, ***kwargs*)

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

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.
`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.

Continued on next page

Table 5 – continued from previous page

cupy.expm1	Computes `exp(x) - 1` elementwise.
cupy.log1p	Computes `log(1 + x)` elementwise.
cupy.sqrt	
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'>**

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.
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`.

Continued on next page

Table 8 – continued from previous page

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 point values

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.

Continued on next page

Table 9 – continued from previous page

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.

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** (*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)`

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

This function currently does not support `order` and `subok` options.

Parameters

- **a** (`cupy.ndarray`) – Base array.
- **dtype** – Data type specifier. The data type of `a` is used by default.

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'>)`

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.

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'>)`

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

This function currently does not support `order` option.

Parameters

- **shape** (*tuple of ints*) – Dimensionalities of the array.
- **dtype** – Data type specifier.

Returns An array filled with ones.

Return type *cupy.ndarray*

See also:

`numpy.ones()`

cupy.ones_like

`cupy.ones_like(a, dtype=None)`

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

This function currently does not support `order` and `subok` options.

Parameters

- **a** (`cupy.ndarray`) – Base array.
- **dtype** – Data type specifier. The dtype of `a` is used by default.

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** (*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)`

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

This function currently does not support `order` and `subok` options.

Parameters

- **a** (`cupy.ndarray`) – Base array.
- **dtype** – Data type specifier. The dtype of `a` is used by default.

Returns An array filled with zeros.

Return type *cupy.ndarray*

See also:

`numpy.zeros_like()`

cupy.full

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

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

This function currently does not support `order` option.

Parameters

- **shape** (*tuple of ints*) – Dimensionalities of the array.
- **fill_value** – A scalar value to fill a new array.
- **dtype** – Data type specifier.

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)`

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

This function currently does not support `order` and `subok` options.

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.

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.asanyarray

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

Converts an object to array.

This is currently equivalent to *asarray()*, since there is no subclass of 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 = <function copy>`

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: `numpy.copy()`, `cupy.ndarray.copy()`

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.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*

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. Also, CuPy does not support `sparse` option yet.

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.
- **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()`

Matrix creation

cupy.diag	Returns a diagonal or a diagonal array.
cupy.diagflat	Creates a diagonal array from the flattened input.

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*

3.3.2 Array Manipulation Routines

Basic manipulations

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()`

Shape manipulation

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

cupy.reshape

`cupy.reshape`(*a*, *newshape*)

Returns an array with new shape and same elements.

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

This function currently does not support `order` option.

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`.

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

Return type *cupy.ndarray*

See also:

`numpy.reshape()`

cupy.ravel

`cupy.ravel(a)`

Returns a flattened array.

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

This function currently does not support `order` option.

Parameters **a** (`cupy.ndarray`) – Array to be flattened.

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

Return type *cupy.ndarray*

See also:

`numpy.ravel()`

Transposition

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()`

Edit dimensionalities

`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_arrays`	Broadcasts given arrays.
`cupy.broadcast_to`	Broadcast an array to a given shape.
`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_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.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.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 along axis

`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*) – The axis to join arrays along.

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 along axis

`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()`

3.3.3 Repeating part of arrays along axis

`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()`

3.3.4 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.

Parameters

- **a** (ndarray) – Array to be rolled.
- **shift** (*int*) – The number of places by which elements are shifted.
- **axis** (*int or None*) – The axis along which elements are shifted. If `axis` is `None`, the array is flattened before shifting, and after that it is reshaped to the original shape.

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()`

3.3.5 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()

3.3.6 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*

See also:

`numpy.fft.irfft()`

cupy.fft.rfft2

`cupy.fft.rfft2(a, s=None, axes=(-2, -1), norm=None)`

Compute the two-dimensional FFT for real input.

Parameters

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

- **s** (*None or tuple of ints*) – Shape to use from the input. If `s` is not given, the lengths of the input along the axes specified by `axes` are used.
- **axes** (*tuple of ints*) – Axes over which to compute the FFT.
- **norm** (None or `"ortho"`) – Keyword to specify the normalization mode.

Returns The transformed array which shape is specified by `s` and type will convert to complex if the input is other. The length of the last axis transformed will be `s[-1]//2+1`.

Return type *cupy.ndarray*

See also:

`numpy.fft.rfft2()`

cupy.fft.irfft2

`cupy.fft.irfft2(a, s=None, axes=(-2, -1), norm=None)`

Compute the two-dimensional inverse FFT for real input.

Parameters

- **a** (`cupy.ndarray`) – Array to be transform.
- **s** (*None or tuple of ints*) – Shape of the output. If `s` is not given, they are determined from the lengths of the input along the axes specified by `axes`.
- **axes** (*tuple of ints*) – Axes over which to compute the FFT.
- **norm** (None or `"ortho"`) – Keyword to specify the normalization mode.

Returns The transformed array which shape is specified by `s` and type will convert to complex if the input is other. If `s` is not given, the length of final transformed axis of output will be *2*(m-1)* where *m* is the length of the final transformed axis of the input.

Return type *cupy.ndarray*

See also:

`numpy.fft.irfft2()`

cupy.fft.rfftn

`cupy.fft.rfftn(a, s=None, axes=None, norm=None)`

Compute the N-dimensional FFT for real input.

Parameters

- **a** (`cupy.ndarray`) – Array to be transform.
- **s** (*None or tuple of ints*) – Shape to use from the input. If `s` is not given, the lengths of the input along the axes specified by `axes` are used.
- **axes** (*tuple of ints*) – Axes over which to compute the FFT.
- **norm** (None or `"ortho"`) – Keyword to specify the normalization mode.

Returns The transformed array which shape is specified by `s` and type will convert to complex if the input is other. The length of the last axis transformed will be `s[-1]//2+1`.

Return type *cupy.ndarray*

See also:

`numpy.fft.rfftn()`

cupy.fft.irfftn

`cupy.fft.irfftn(a, s=None, axes=None, norm=None)`

Compute the N-dimensional inverse FFT for real input.

Parameters

- **a** (`cupy.ndarray`) – Array to be transform.
- **s** (*None or tuple of ints*) – Shape of the output. If `s` is not given, they are determined from the lengths of the input along the axes specified by `axes`.
- **axes** (*tuple of ints*) – Axes over which to compute the FFT.
- **norm** (None or `"ortho"`) – Keyword to specify the normalization mode.

Returns The transformed array which shape is specified by `s` and type will convert to complex if the input is other. If `s` is not given, the length of final transformed axis of output will be `2*(m-1)` where *m* is the length of the final transformed axis of the input.

Return type *cupy.ndarray*

See also:

`numpy.fft.irfftn()`

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`.

3.3.7 Indexing Routines

*cupy.c_*	Translates slice objects to concatenation along the second axis.
*cupy.r_*	Translates slice objects to concatenation along the first axis.
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.ix_*	Construct an open mesh from multiple sequences.
cupy.take	Takes elements of an array at specified indices along an axis.
cupy.choose	
cupy.diag	Returns a diagonal or a diagonal array.
cupy.diagonal	Returns specified diagonals.
cupy.fill_diagonal	Fills the main diagonal of the given array of any dimensionality.

cupy.c_

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

Translates slice objects to concatenation along the second axis.

This is a CuPy object that corresponds to *cupy.r_*, which is useful because of its common occurrence. In particular, arrays will be stacked along their last axis after being upgraded to at least 2-D with 1's post-pended to the shape (column vectors made out of 1-D arrays).

For detailed documentation, see *r_*.

This implementation is partially borrowed from NumPy's one.

Returns Joined array.

Return type *cupy.ndarray*

See also:

`numpy.c_`

Examples

```
>>> a = cupy.array([[1, 2, 3]], dtype=np.int32)
>>> b = cupy.array([[4, 5, 6]], dtype=np.int32)
>>> cupy.c_[a, 0, 0, b]
array([[1, 2, 3, 0, 0, 4, 5, 6]], dtype=int32)
```

cupy.r_

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

Translates slice objects to concatenation along the first axis.

This is a simple way to build up arrays quickly. If the index expression contains comma separated arrays, then stack them along their first axis.

This object can build up from normal CuPy arrays. Therefore, the other objects (e.g. writing strings like '2,3,4', or using imaginary numbers like [1,2,3j], or using string integers like '-1') are not implemented yet compared with NumPy.

This implementation is partially borrowed from NumPy's one.

Returns Joined array.

Return type *cupy.ndarray*

See also:

`numpy.r_`

Examples

```
>>> a = cupy.array([1, 2, 3], dtype=np.int32)
>>> b = cupy.array([4, 5, 6], dtype=np.int32)
>>> cupy.r_[a, 0, 0, b]
array([1, 2, 3, 0, 0, 4, 5, 6], dtype=int32)
```

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

See also:

`numpy.nonzero()`

cupy.where

`cupy.where = <function where>`

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*

See also:

`numpy.where()`

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]]))
```

See also:

`numpy.ix_()`

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.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.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()`

3.3.8 Input and Output

NPZ files

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*)

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.

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*)

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()*.

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()

3.3.9 Linear Algebra

Matrix and vector products

cupy.dot	Returns a dot product of two arrays.
cupy.vdot	Returns the dot product of two vectors.
cupy.inner	Returns the inner product of two arrays.
cupy.outer	Returns the outer product of two vectors.
cupy.matmul	Returns the matrix product of two arrays and is the implementation of the @ operator introduced in Python 3.5 following PEP465.
cupy.tensordot	Returns the tensor dot product of two arrays along specified axes.
cupy.einsum	Evaluates the Einstein summation convention on the operands.
cupy.kron	Returns the kronecker product of two arrays.

cupy.dot

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

Returns a dot product of two arrays.

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

Parameters

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

Returns The dot product of a and b.

Return type *cupy.ndarray*

See also:

numpy.dot()

cupy.vdot

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

Returns the dot product of two vectors.

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

Parameters

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

Returns Zero-dimensional array of the dot product result.

Return type *cupy.ndarray*

See also:

`numpy.vdot()`

cupy.inner

`cupy.inner(a, b)`

Returns the inner product of two arrays.

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

Parameters

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

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

Return type *cupy.ndarray*

See also:

`numpy.inner()`

cupy.outer

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

Returns the outer product of two vectors.

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

Parameters

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

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

Return type *cupy.ndarray*

See also:

`numpy.outer()`

cupy.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)`

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`, `dtype`, `casting` and `optimize` 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.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. Note that in the current implementation a must be a real matrix, and only float32 and float64 are supported.

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

Returns The lower-triangular matrix.

Return type *cupy.ndarray*

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

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 True, 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*

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: Currenlty only 2-D matrix is supported.

Note: CUDA >=8.0 is required.

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*

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.

Note: CUDA >=8.0 is required.

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*

See also:

`numpy.linalg.eigvalsh()`

Norms etc.

cupy.linalg.det	Retruns the deteminant 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 determinat of an array.
cupy.trace	Returns the sum along the diagonals of an array.

cupy.linalg.det

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

Retruns the deteminant 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.

Complex valued matrices and vectors are not supported. 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 determinat of an array.

It calculates the natural logarithm of the deteminant 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 deteminant as a real number `0`, `1` or `-1`. 'logdet' represents the natural logarithm of the absolute of the deteminant. If the deteninant 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*

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.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.

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*

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*

See also:

`numpy.linalg.tensorsolve()`

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*

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*

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*

See also:

`numpy.linalg.tensorinv()`

3.3.10 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.all

cupy.**all = <cupy.core.fusion.reduction object>**

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 = <cupy.core.fusion.reduction object>**

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()

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

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

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

See also:

`numpy.isscalar()`

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='FORTRAN')
>>> 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='FORTRAN'))
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
```

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 operations

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`.

3.3.11 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.deg2rad	Converts angles from degrees to radians elementwise.
cupy.rad2deg	Converts angles from radians to degrees elementwise.
cupy.degrees	Converts angles from radians to degrees elementwise.
cupy.radians	Converts angles from degrees to radians 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

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.rint	Rounds each element of an array to the nearest integer.
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.fix	If given value x is positive, it return floor(x).

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 and products

cupy.sum	Returns the sum of an array along given axes.
cupy.prod	Returns the product of an array along given axes.
cupy.cumsum	Returns the cumulative sum of an array along a given axis.

Continued on next page

Table 47 – continued from previous page

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

cupy.sum

cupy.**sum = <cupy.core.fusion.reduction object>**

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.prod

cupy.**prod = <cupy.core.fusion.reduction object>**

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.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.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()`

Exponential and logarithm functions

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.

Floating point manipulations

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

Continued on next page

Table 49 – continued from previous page

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

Arithmetic operations

cupy.negative	Takes numerical negative elementwise.
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.true_divide	Elementwise true division (i.e.
cupy.floor_divide	Elementwise floor division (i.e.
cupy.power	Computes `x1 ** x2` elementwise.
cupy.fmod	Computes the remainder of C division elementwise.
cupy.mod	Computes the remainder of Python division elementwise.
cupy.remainder	Computes the remainder of Python division elementwise.
cupy.modf	Extracts the fractional and integral parts of an array elementwise.
cupy.reciprocal	Computes `1 / x` elementwise.

Miscellaneous

cupy.clip	Clips the values of an array to a given interval.
cupy.sqrt	
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.blackman	Returns the Blackman window.
cupy.hamming	Returns the Hamming window.
cupy.hanning	Returns the Hanning window.

cupy.clip

`cupy.clip = <function clip>`

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.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()`

3.3.12 Padding

cupy.pad	Returns padded array.

cupy.pad

`cupy.pad(array, pad_width, mode, **keywords)`

Returns padded array. You can specify the padded widths and values.

This function currently supports only `mode=constant` .

Parameters

- **array** (*array-like*) – Input array of rank N.
- **pad_width** (*int or array-like*) – Number of values padded to the edges of each axis. ((before_1, after_1), … (before_N, after_N)) uniquely 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*) –

  'constant' Pads with a constant values.
- **constant_values** (*int or array-like*) – Used in `constant`. The values are padded for each axis. ((before_1, after_1), … (before_N, after_N)) uniquely pad constants for each axis. ((before, after),) yields same before and after constants for each axis. (constant,) or int is a shortcut for before = after = constant for all axes. Default is 0. You cannot specify `cupy.ndarray` .

Returns Padded array of rank equal to `array` with shape increased according to `pad_width` .

Return type *cupy.ndarray*

See also:

`numpy.pad()`

3.3.13 Random Sampling (`cupy.random`)

CuPy's random number generation routines are based on cuRAND. They cover a small fraction of `numpy.random`.

The big difference of *cupy.random* from `numpy.random` is that *cupy.random* supports `dtype` option for most functions. This option enables us to generate float32 values directly without any space overhead.

Sample random data

cupy.random.choice	Returns an array of random values from a given 1-D array.
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.
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.bytes	Returns random bytes.

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.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.bytes

`cupy.random.`**`bytes`**`(`*length*`)`

Returns random bytes.

See also:

`numpy.random.bytes()`

Distributions

`cupy.random.gumbel`	Returns an array of samples drawn from a Gumbel distribution.
`cupy.random.lognormal`	Returns an array of samples drawn from a log normal distribution.
`cupy.random.normal`	Returns an array of normally distributed samples.
`cupy.random.standard_normal`	Returns an array of samples drawn from the standard normal distribution.
`cupy.random.uniform`	Returns an array of uniformly-distributed samples over an interval.

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 μ is `loc` and η is `scale`.

Parameters

- **loc** (*float*) – The location of the mode μ.
- **scale** (*float*) – The scale parameter η.
- **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 destribution.

Return type *cupy.ndarray*

See also:

`cupy.random.RandomState.gumbel()` `numpy.random.gumbel()`

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.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.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.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()`

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	Portable container of a pseudo-random number generator.

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.clock()` 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.

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

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()`

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()`

interval(*mx*, *size*)
Generate multiple integers independently sampled uniformly from `[0, mx]`.

Parameters

- **mx** (`int`) – Upper bound of the interval
- **size** (`None or int or tuple`) – Shape of the array or the scalar returned.

Returns If `None`, an `cupy.ndarray` with shape `()` is returned. If `int`, 1-D array of length size is returned. If `tuple`, multi-dimensional array with shape `size` is returned. Currently, only 32 bit integers can be sampled. If $0 \leq$ `mx` $\leq$ 0x7fffffff, a `numpy.int32` array is returned. If 0x80000000 $\leq$ `mx` $\leq$ 0xffffffff, a `numpy.uint32` array is returned.

Return type int or *cupy.ndarray*

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()`

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()`

permutation(*num*)
Returns a permuted range.

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()`

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_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()`

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()`

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()`

Permutations

cupy.random.shuffle	Shuffles 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()`

3.3.14 Sorting, Searching, and Counting

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.argmax	Returns the indices of the maximum along an axis.
cupy.argmin	Returns the indices of the minimum along an axis.
cupy.partition	Returns a partitioned copy of an array.
cupy.argpartition	Returns the indices that would partially sort an array.
cupy.count_nonzero	Counts the number of non-zero values in the array.
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.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.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*

See also:

`numpy.argmax()`

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*

See also:

numpy.argmin()

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()`

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

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*

See also:

`numpy.flatnonzero()`

3.3.15 Statistics

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.

Continued on next page

Table 58 – continued from previous page

cupy.nanmax	Returns the maximum of an array along an axis ignoring NaN.

cupy.amin

`cupy.`**`amin = <cupy.core.fusion.reduction object>`**

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 = <cupy.core.fusion.reduction object>`**

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*

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*

See also:

`numpy.nanmax()`

Means and variances

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.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()`

Histograms

cupy.bincount	Count number of occurrences of each value in array of non-negative ints.

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*

See also:

`numpy.bincount()`

3.3.16 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.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()`.

3.4 Sparse matrix

CuPy supports sparse matrices using cuSPARSE. These matrices have the same interfaces of SciPy's sparse matrices.

3.4.1 Sparse matrix classes

cupy.sparse.coo_matrix	COOrdinate format sparse matrix.
cupy.sparse.csr_matrix	Compressed Sparse Row matrix.
cupy.sparse.csc_matrix	Compressed Sparse Column matrix.
cupy.sparse.dia_matrix	Sparse matrix with DIAgonal storage.
cupy.sparse.spmatrix	Base class of all sparse matrixes.

cupy.sparse.coo_matrix

class cupy.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 *cupy.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 ()

conjugate ()

copy ()
Returns a copy of this 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 ()
Returns the main diagonal of the matrix

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.

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 *cupy.sparse.coo_matrix*

tocsc(*copy=False*)
Converts the matrix to Compressed Sparse Column format.

Parameters **copy** (*bool*) – If `False`, it shares data arrays as much as possible. Actually this option is ignored because all arrays in a matrix cannot be shared in coo to csc conversion.

Returns Converted matrix.

Return type *cupy.sparse.csc_matrix*

tocsr(*copy=False*)
Converts the matrix to Compressed Sparse Row format.

Parameters **copy** (*bool*) – If False, it shares data arrays as much as possible. Actually this option is ignored because all arrays in a matrix cannot be shared in coo to csr conversion.

Returns Converted matrix.

Return type *cupy.sparse.csr_matrix*

todense(*order=None*, *out=None*)
Return a dense matrix representation of this matrix.

todia(*copy=False*)
Convert this matrix to sparse DIAgonal format.

todok(*copy=False*)
Convert this matrix to Dictionary Of Keys format.

tolil(*copy=False*)
Convert this matrix to LInked List format.

transpose(*axes=None*, *copy=False*)
Returns a transpose matrix.

Parameters

- **axes** – This option is not supported.
- **copy** (*bool*) – If True, a returned matrix shares no data. Otherwise, it shared data arrays as much as possible.

Returns Transpose matrix.

Return type *cupy.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

cupy.sparse.csr_matrix

class cupy.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 *cupy.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`()

`conjugate`()

`copy`()
Returns a copy of this 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()
Returns the main diagonal of the matrix

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 *cupy.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 *cupy.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 *cupy.sparse.csr_matrix*

todense(*order=None*, *out=None*)
Return a dense matrix representation of this matrix.

todia(*copy=False*)
Convert this matrix to sparse DIAgonal format.

todok(*copy=False*)
Convert this matrix to Dictionary Of Keys format.

tolil(*copy=False*)
Convert this matrix to LInked List format.

transpose(*axes=None*, *copy=False*)
Returns a transpose matrix.

Parameters

- **axes** – This option is not supported.
- **copy** (*bool*) – If `True`, a returned matrix shares no data. Otherwise, it shared data arrays as much as possible.

Returns Transpose matrix.

Return type *cupy.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

cupy.sparse.csc_matrix

class cupy.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 *cupy.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()

conjugate()

copy()
Returns a copy of this 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()
Returns the main diagonal of the matrix

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 *cupy.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 *cupy.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 *cupy.sparse.csr_matrix*

todense(*order=None*, *out=None*)
Return a dense matrix representation of this matrix.

todia(*copy=False*)
Convert this matrix to sparse DIAgonal format.

todok(*copy=False*)
Convert this matrix to Dictionary Of Keys format.

tolil(*copy=False*)
Convert this matrix to LInked List format.

transpose(*axes=None, copy=False*)
Returns a transpose matrix.

Parameters

- **axes** – This option is not supported.
- **copy** (*bool*) – If `True`, a returned matrix shares no data. Otherwise, it shared data arrays as much as possible.

Returns Transpose matrix.

Return type *cupy.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

cupy.sparse.dia_matrix

class cupy.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 *cupy.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 ()

conjugate ()

copy ()
Returns a copy of this 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 ()
Returns the main diagonal of the matrix

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 *cupy.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 *cupy.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

cupy.sparse.spmatrix

class cupy.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 *cupy.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 *cupy.sparse.spmatrix*

conj()

conjugate()

copy()
Returns a copy of this matrix.

count_nonzero()
Number of non-zero entries, equivalent to

diagonal()
Returns the main diagonal of the matrix

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

3.4.2 Functions

Building sparse matrices

`cupy.sparse.eye`	Creates a sparse matrix with ones on diagonal.
`cupy.sparse.identity`	Creates an identity matrix in sparse format.

cupy.sparse.eye

cupy.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 *cupy.sparse.spmatrix*

See also:

`scipy.sparse.eye()`

cupy.sparse.identity

`cupy.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 *cupy.sparse.spmatrix*

See also:

`scipy.sparse.identity()`

Identifying sparse matrices

cupy.sparse.issparse	Checks if a given matrix is a sparse matrix.
cupy.sparse.isspmatrix	Checks if a given matrix is a sparse matrix.
cupy.sparse.isspmatrix_csc	Checks if a given matrix is of CSC format.
cupy.sparse.isspmatrix_csr	Checks if a given matrix is of CSR format.
cupy.sparse.isspmatrix_coo	Checks if a given matrix is of COO format.
cupy.sparse.isspmatrix_dia	Checks if a given matrix is of DIA format.

cupy.sparse.issparse

`cupy.sparse.issparse(x)`
Checks if a given matrix is a sparse matrix.

Returns

Returns if x is `cupy.sparse.spmatrix` that is a base class of all sparse matrix classes.

Return type bool

cupy.sparse.isspmatrix

`cupy.sparse.isspmatrix(x)`
Checks if a given matrix is a sparse matrix.

Returns

Returns if x is `cupy.sparse.spmatrix` that is a base class of all sparse matrix classes.

Return type bool

cupy.sparse.isspmatrix_csc

`cupy.sparse.isspmatrix_csc(x)`

Checks if a given matrix is of CSC format.

Returns Returns if `x` is *cupy.sparse.csc_matrix*.

Return type bool

cupy.sparse.isspmatrix_csr

`cupy.sparse.isspmatrix_csr(x)`

Checks if a given matrix is of CSR format.

Returns Returns if `x` is *cupy.sparse.csr_matrix*.

Return type bool

cupy.sparse.isspmatrix_coo

`cupy.sparse.isspmatrix_coo(x)`

Checks if a given matrix is of COO format.

Returns Returns if `x` is *cupy.sparse.coo_matrix*.

Return type bool

cupy.sparse.isspmatrix_dia

`cupy.sparse.isspmatrix_dia(x)`

Checks if a given matrix is of DIA format.

Returns Returns if `x` is *cupy.sparse.dia_matrix*.

Return type bool

Linear Algebra

cupy.sparse.linalg.lsqr	Solves linear system with QR decomposition.

cupy.sparse.linalg.lsqr

`cupy.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* `cupy.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()`

3.5 NumPy-CuPy Generic Code Support

cupy.get_array_module	Returns the array module for arguments.

3.6 Low-Level CUDA Support

3.6.1 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*)

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

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
id – 'int'

3.6.2 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.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.set_allocator	Sets the current allocator for GPU memory.
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**(*Py_ssize_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.

Variables

- ***ptr*** (*int*) – Pointer to the place within the buffer.
- ***size*** (*int*) – Size of the memory allocation in bytes.
- ***device*** (Device) – Device whose memory the pointer refers to.

Methods

Attributes

device

ptr

ptr – 'size_t'

size

size – 'Py_ssize_t'

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**(*Memory mem*, *Py_ssize_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** (Memory) – 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*** (Memory) – The device memory buffer.
- ***ptr*** (*size_t*) – 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*, *Py_ssize_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

mem

ptr

cupy.cuda.PinnedMemoryPointer

class cupy.cuda.**PinnedMemoryPointer**(*mem*, *Py_ssize_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*) → Py_ssize_t

Attributes

mem

ptr

cupy.cuda.alloc

cupy.cuda.**alloc**(*Py_ssize_t 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`**(*Py_ssize_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.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.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*)
Release 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*)
Release free blocks.

free_bytes (*self*)
Get 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

malloc (*self*, *Py_ssize_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*)
Count the total number of free blocks.

Returns The total number of free blocks.

Return type int

total_bytes (*self*)
Get 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*)
Get 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, size_t ptr, Py_ssize_t size*)

free_all_blocks (*self*)
Release free all blocks.

malloc (*self, Py_ssize_t size*) → PinnedMemoryPointer

n_free_blocks (*self*)
Count the total number of free blocks.

Returns The total number of free blocks.

Return type int

3.6.3 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*, **_*)

`alloc_postprocess`(*self*, ***kwargs*)

Callback function invoked after allocating memory from GPU device.

Keyword Arguments

- **device_id** (*int*) – CUDA device ID
- **mem_size** (*int*) – Rounded memory bytesize allocated
- **mem_ptr** (*int*) – Obtained memory pointer. 0 if an error occurred in allocation.

`alloc_preprocess`(*self*, ***kwargs*)

Callback function invoked before allocating memory from GPU device.

Keyword Arguments

- **device_id** (*int*) – CUDA device ID
- **mem_size** (*int*) – Rounded memory bytesize to be allocated

free_postprocess (*self*, ***kwargs*)
Callback function invoked after releasing memory to memory pool.

Keyword Arguments

- **device_id** (*int*) – CUDA device ID
- **mem_size** (*int*) – Memory bytesize
- **mem_ptr** (*int*) – Memory pointer to free
- **pmem_id** (*int*) – Pooled memory object ID.

free_preprocess (*self*, ***kwargs*)
Callback function invoked before releasing memory to memory pool.

Keyword Arguments

- **device_id** (*int*) – CUDA device ID
- **mem_size** (*int*) – Memory bytesize
- **mem_ptr** (*int*) – Memory pointer to free
- **pmem_id** (*int*) – Pooled memory object ID.

malloc_postprocess (*self*, ***kwargs*)
Callback function invoked after retrieving memory from memory pool.

Keyword Arguments

- **device_id** (*int*) – CUDA device ID
- **size** (*int*) – Requested memory bytesize to allocate
- **mem_size** (*int*) – Rounded memory bytesize allocated
- **mem_ptr** (*int*) – Obtained memory pointer. 0 if an error occurred in `malloc`.
- **pmem_id** (*int*) – Pooled memory object ID. 0 if an error occurred in `malloc`.

malloc_preprocess (*self*, ***kwargs*)
Callback function invoked before retrieving memory from memory pool.

Keyword Arguments

- **device_id** (*int*) – CUDA device ID
- **size** (*int*) – Requested memory bytesize to allocate
- **mem_size** (*int*) – Rounded memory bytesize to be allocated

Attributes

name = 'MemoryHook'

cupy.cuda.memory_hooks.DebugPrintHook

class cupy.cuda.memory_hooks.**DebugPrintHook** (*file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>, flush=True*)
Memory hook that prints debug information.

This memory hook outputs the debug information of input arguments of `malloc` and `free` methods involved in the hooked functions at postprocessing time (that is, just after each method is called).

Example

The basic usage is to use it with `with` statement.

Code example:

```
>>> import cupy
>>> from cupy.cuda import memory_hooks
>>>
>>> cupy.cuda.set_allocator(cupy.cuda.MemoryPool().malloc)
>>> with memory_hooks.DebugPrintHook():
...     x = cupy.array([1, 2, 3])
...     del x
```

Output example:

```
{"hook":"alloc","device_id":0,"mem_size":512,"mem_ptr":150496608256}
{"hook":"malloc","device_id":0,"size":24,"mem_size":512,"mem_ptr":150496608256,
↪"pmem_id":"0x7f39200c5278"}
{"hook":"free","device_id":0,"mem_size":512,"mem_ptr":150496608256,"pmem_id":
↪"0x7f39200c5278"}
```

where the output format is JSONL (JSON Lines) and `hook` is the name of hook point, and `device_id` is the CUDA Device ID, and `size` is the requested memory size to allocate, and `mem_size` is the rounded memory size to be allocated, and `mem_ptr` is the memory pointer, and `pmem_id` is the pooled memory object ID.

Variables

- **file** – Output file_like object that redirect to.
- **flush** – If `True`, this hook forcibly flushes the text stream at the end of print. The default is `True`.

Methods

__enter__(*self*)

__exit__(*self*, **_*)

alloc_postprocess(*self*, ***kwargs*)
Callback function invoked after allocating memory from GPU device.

Keyword Arguments

- **device_id** (*int*) – CUDA device ID
- **mem_size** (*int*) – Rounded memory bytesize allocated
- **mem_ptr** (*int*) – Obtained memory pointer. 0 if an error occurred in allocation.

alloc_preprocess(*self*, ***kwargs*)
Callback function invoked before allocating memory from GPU device.

Keyword Arguments

- **device_id** (*int*) – CUDA device ID

- **mem_size** (*int*) – Rounded memory bytesize to be allocated

free_postprocess (*self*, ***kwargs*)
Callback function invoked after releasing memory to memory pool.

Keyword Arguments

- **device_id** (*int*) – CUDA device ID
- **mem_size** (*int*) – Memory bytesize
- **mem_ptr** (*int*) – Memory pointer to free
- **pmem_id** (*int*) – Pooled memory object ID.

free_preprocess (*self*, ***kwargs*)
Callback function invoked before releasing memory to memory pool.

Keyword Arguments

- **device_id** (*int*) – CUDA device ID
- **mem_size** (*int*) – Memory bytesize
- **mem_ptr** (*int*) – Memory pointer to free
- **pmem_id** (*int*) – Pooled memory object ID.

malloc_postprocess (*self*, ***kwargs*)
Callback function invoked after retrieving memory from memory pool.

Keyword Arguments

- **device_id** (*int*) – CUDA device ID
- **size** (*int*) – Requested memory bytesize to allocate
- **mem_size** (*int*) – Rounded memory bytesize allocated
- **mem_ptr** (*int*) – Obtained memory pointer. 0 if an error occurred in `malloc`.
- **pmem_id** (*int*) – Pooled memory object ID. 0 if an error occurred in `malloc`.

malloc_preprocess (*self*, ***kwargs*)
Callback function invoked before retrieving memory from memory pool.

Keyword Arguments

- **device_id** (*int*) – CUDA device ID
- **size** (*int*) – Requested memory bytesize to allocate
- **mem_size** (*int*) – Rounded memory bytesize to be allocated

Attributes

name = 'DebugPrintHook'

cupy.cuda.memory_hooks.LineProfileHook

class `cupy.cuda.memory_hooks.`**LineProfileHook** (*max_depth=0*)
Code line CuPy memory profiler.

This profiler shows line-by-line GPU memory consumption using traceback module. But, note that it can trace only CPython level, no Cython level. ref. https://github.com/cython/cython/issues/1755

Example

Code example:

```
from cupy.cuda import memory_hooks
hook = memory_hooks.LineProfileHook()
with hook:
    # some CuPy codes
hook.print_report()
```

Output example:

```
_root (4.00KB, 4.00KB)
  lib/python3.6/unittest/__main__.py:18:<module> (4.00KB, 4.00KB)
    lib/python3.6/unittest/main.py:255:runTests (4.00KB, 4.00KB)
      tests/cupy_tests/test.py:37:test (1.00KB, 1.00KB)
      tests/cupy_tests/test.py:38:test (1.00KB, 1.00KB)
      tests/cupy_tests/test.py:39:test (2.00KB, 2.00KB)
```

Each line shows:

```
{filename}:{lineno}:{func_name} ({used_bytes}, {acquired_bytes})
```

where *used_bytes* is the memory bytes used from CuPy memory pool, and *acquired_bytes* is the actual memory bytes the CuPy memory pool acquired from GPU device. *_root* is a root node of the stack trace to show total memory usage.

Parameters **max_depth** (*int*) – maximum depth to follow stack traces. Default is 0 (no limit).

Methods

__enter__(*self*)

__exit__(*self*, **_*)

alloc_postprocess(*self*, ***kwargs*)
Callback function invoked after allocating memory from GPU device.

Keyword Arguments

- **device_id** (*int*) – CUDA device ID
- **mem_size** (*int*) – Rounded memory bytesize allocated
- **mem_ptr** (*int*) – Obtained memory pointer. 0 if an error occurred in allocation.

alloc_preprocess(*self*, ***kwargs*)
Callback function invoked before allocating memory from GPU device.

Keyword Arguments

- **device_id** (*int*) – CUDA device ID
- **mem_size** (*int*) – Rounded memory bytesize to be allocated

free_postprocess(*self*, ***kwargs*)
Callback function invoked after releasing memory to memory pool.

Keyword Arguments

- **device_id** (*int*) – CUDA device ID

- **mem_size** (*int*) – Memory bytesize
- **mem_ptr** (*int*) – Memory pointer to free
- **pmem_id** (*int*) – Pooled memory object ID.

free_preprocess (*self*, ***kwargs*)
Callback function invoked before releasing memory to memory pool.

Keyword Arguments

- **device_id** (*int*) – CUDA device ID
- **mem_size** (*int*) – Memory bytesize
- **mem_ptr** (*int*) – Memory pointer to free
- **pmem_id** (*int*) – Pooled memory object ID.

malloc_postprocess (*self*, ***kwargs*)
Callback function invoked after retrieving memory from memory pool.

Keyword Arguments

- **device_id** (*int*) – CUDA device ID
- **size** (*int*) – Requested memory bytesize to allocate
- **mem_size** (*int*) – Rounded memory bytesize allocated
- **mem_ptr** (*int*) – Obtained memory pointer. 0 if an error occurred in `malloc`.
- **pmem_id** (*int*) – Pooled memory object ID. 0 if an error occurred in `malloc`.

malloc_preprocess (*self*, ***kwargs*)
Callback function invoked before retrieving memory from memory pool.

Keyword Arguments

- **device_id** (*int*) – CUDA device ID
- **size** (*int*) – Requested memory bytesize to allocate
- **mem_size** (*int*) – Rounded memory bytesize to be allocated

print_report (*file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*)
Prints a report of line memory profiling.

Attributes

name = 'LineProfileHook'

3.6.4 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** (*size_t*) – Raw stream handle. It can be passed to the CUDA Runtime API via ctypes.

Methods

__enter__(*self*)

__exit__(*self*, **args*)

add_callback(*self*, *callback*, *arg*)
Adds a callback that is called when all queued work is done.

Parameters

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

record(*self*, *event=None*)
Records an event on the stream.

Parameters **event** (*None or* `cupy.cuda.Event`) – CUDA event. If `None`, then a new plain event is created and used.

Returns The recorded event.

Return type *cupy.cuda.Event*

See also:

cupy.cuda.Event.record()

synchronize(*self*)
Waits for the stream completing all queued work.

use(*self*)
Makes this stream current.

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

wait_event(*self*, *event*)
Makes the stream wait for an event.

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

Parameters **event** (`cupy.cuda.Event`) – CUDA event.

__eq__(*self*, *other*)

Attributes

done
True if all work on this stream has been done.

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

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 (*size_t*) – Raw stream handle. It can be passed to the CUDA Runtime API via ctypes.

Methods

record(*self*, *stream=None*)
Records the event to a stream.

Parameters stream (cupy.cuda.Stream) – CUDA stream to record event. The null stream is used by default.

See also:

cupy.cuda.Stream.record()

synchronize(*self*)
Synchronizes all device work to the event.

If the event is created as a blocking event, it also blocks the CPU thread until the event is done.

Attributes

done
True if the event is done.

cupy.cuda.get_elapsed_time

cupy.cuda.**get_elapsed_time**(*start_event*, *end_event*)
Gets the elapsed time between two events.

Parameters

- **start_event** (Event) – Earlier event.
- **end_event** (Event) – Later event.

Returns Elapsed time in milliseconds.

Return type float

3.6.5 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**(*str config_file*, *str output_file*, *int output_mode*) → void
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()` → void

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()` → void

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(str message, int id_color=-1)` → void

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(str message, uint32_t color=0)` → void

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(str message, int id_color=-1)` → void

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**(*str message*, *uint32_t color=0*) → void
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**() → void
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 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.

3.7.1 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.

3.7.2 cupy.clear_memo

cupy.**clear_memo**()
Clears the memoized results for all functions decorated by memoize.

3.8 Custom kernels

cupy.ElementwiseKernel	User-defined elementwise kernel.
cupy.ReductionKernel	User-defined reduction kernel.

3.8.1 cupy.ElementwiseKernel

class cupy.**ElementwiseKernel**(*in_params*, *out_params*, *operation*, *name='kernel'*, *reduce_dims=True*, *preamble=''*, ***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** (*list*) – Options passed to the `nvcc` command.
- **preamble** (*str*) – Fragment of the CUDA-C/C++ code that is inserted at the top of the cu file.
- **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 Arrays are returned according to the `out_params` argument of the `__init__` method.

Attributes

in_params

kwargs

name

nargs

nin

nout

operation

out_params

params

preamble

reduce_dims

3.8.2 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.

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

3.9 Testing Modules

CuPy offers testing utilities to support unit testing. They are under namespace *cupy.testing*.

3.9.1 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`.
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*)

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.
- **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()

3.9.2 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*, *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 cupy.sparse 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*)

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 `cupy.sparse` 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*)

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 `cupy.sparse` 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*)

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 `cupy.sparse` 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*)

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 cupy.sparse module. If None, no argument is given for the modules.

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*)

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 cupy.sparse 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*)

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 `cupy.sparse` 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*, *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 `cupy.sparse` 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`.

3.9.3 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_combinat	Decorator for parameterized test w.r.t.
cupy.testing.for_unsigned_dtypes_combin	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):
...
```

(continues on next page)

(continued from previous page)

```
...     @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 the subset of possible combinations is tested. Specifically, at first, the shuffled lists of `types` are made for each argument name in `names`. Let the lists be `D1`, `D2`, `...`, `Dn` where n is the number of arguments. Then, the combinations to be tested will be `zip(D1, ..., Dn)`. If `full` is `None`, the behavior is switched by setting the environment variable `CUPY_TEST_FULL_COMBINATION=1`.

For example, let `types` be `[float16, float32, float64]` and `names` be `['a_type', 'b_type']`. If `full` is `True`, then the decorated test fixture is executed with all 2^3 patterns. On the other hand, if `full` is `False`, shuffled lists are made for `a_type` and `b_type`. Suppose the lists are `(16, 64,`

32) for a_type and (32, 64, 16) for b_type (prefixes are removed for short). Then the combinations of (a_type, b_type) to be tested are (16, 32), (64, 64) and (32, 16).

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()

3.9.4 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()

3.10 Profiling

3.10.1 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()
... 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()`

3.11 Environment variables

Here are the environment variables CuPy uses.

`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.

3.11.1 For install

These environment variables are only used during installation.

`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.
`NVCC`	Define the compiler to use when compiling CUDA files.

3.12 Difference between CuPy and NumPy

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

3.12.1 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 Intel CPU result.

```
>>> 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)
```

3.12.2 Random methods support dtype argument

NumPy's random value generator does not support dtype option and it always returns a `float32` value. We support the option in CuPy because cuRAND, which is used in CuPy, supports any types of float values.

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

3.12.3 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])
```

3.12.4 Duplicate values in indices

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

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

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

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

3.12.5 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')
```

3.12.6 Data types

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

3.12.7 Array creation from Python objects

Currently, *`cupy.array()`* or *`cupy.asarray()`* cannot create an array from Python object containing CuPy array (e.g., a list of CuPy arrays). Use *`cupy.stack()`* instead.

```
>>> data_cpu = [np.arange(10), np.arange(10)]
>>> np.asarray(data_cpu)
array([[0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
       [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]])
```

```
>>> data_gpu = [cupy.arange(10), cupy.arange(10)]
>>> cupy.asarray(data_gpu)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Unsupported dtype object
>>> cupy.stack(data_gpu)
array([[0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
       [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]])
```

3.12.8 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'>
```

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. Writing a post about CuPy

This document mainly focuses on 1 and 2, though other contributions are also appreciated.

5.2 Release and Milestone

We are using GitHub Flow as our basic working process. In particular, we are using the master branch for our development, and releases are made as tags.

Releases are classified into three groups: major, minor, and revision. This classification is based on following criteria:

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

The release classification is reflected into the version number x.y.z, where x, y, and z corresponds to major, minor, and revision updates, respectively.

We set a milestone for an upcoming release. The milestone is of name 'vX.Y.Z', where the version number represents a revision release at the outset. If at least one *feature* PR is merged in the period, we rename the milestone to represent a minor release (see the next section for the PR types).

See also *API Compatibility Policy*.

5.3 Issues and PRs

Issues and PRs are classified into following categories:

- **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

Issues and PRs are labeled by these categories. This classification is often reflected into its corresponding release category: Feature issues/PRs are contained into minor/major releases and NoCompat issues/PRs are contained into major releases, while other issues/PRs can be contained into any releases including revision ones.

On registering an issue, write precise explanations on what 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). 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 redirect question issues to CuPy User Group.

If you can write code to fix an issue, send a PR to the master branch. Before writing your code for PRs, read through the *Coding Guidelines*. The description of any PR must contain a precise explanation of **what** and **how** you want to do; it is the first documentation of your code for developers, 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 need 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. Since this service is running on our internal server, we do not use it for automatic PR tests to keep the server secure.

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.

5.4 Coding Guidelines

We use PEP8 and a part of OpenStack Style Guidelines related to general coding style as our basic style guidelines.

To check your code, use `autopep8` and `flake8` command installed by `hacking` package:

```
$ pip install autopep8 hacking
$ autopep8 --global-config .pep8 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 --global-config .pep8 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.

5.5 Testing Guidelines

Testing is one of the most important part of your code. You must test your code by unit tests following our testing guidelines.

Note that we are using pytest and mock package for testing, so install them before writing your code:

```
$ pip install pytest mock
```

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
```

Tests are put into the tests/cupy_tests and tests/install_tests directories. These have the same structure as that of cupy and install directories, respectively. 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.

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.

There are many examples of unit tests under the tests directory. They simply use the `unittest` package of the standard library.

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):
    ...

    @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()` or `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):
        ...
```

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.

We leverage doctest as well. You can run doctest by typing `make doctest` at the docs directory:

```
$ cd docs
$ make doctest
```

CHAPTER 6

Installation Guide

6.1 Recommended Environments

We recommend the following Linux distributions.

- Ubuntu 14.04 / 16.04 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: 7.0, 7.5, 8.0, 9.0, 9.1 and 9.2.
  - 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: 2.7.6+, 3.4.3+, 3.5.1+ and 3.6.0+.
- **NumPy**
  - Supported Versions: 1.9, 1.10, 1.11, 1.12, 1.13 and 1.14.
  - 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
```

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: v4, v5, v5.1, v6, v7 and v7.1.
- **NCCL (library to perform collective multi-GPU / multi-node computations)**
  - Supported Versions: v1.3.4, v2, v2.1 and v2.2.

6.3 Install CuPy

Wheels (precompiled binary packages) are available for the recommended environments above. 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
```

Note: The latest version of cuDNN and NCCL libraries are included in these wheels. 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 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 14.04
$ 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.

6.4.1 Using pip

You can install CuPy package via `pip`.

```
$ pip install cupy
```

6.4.2 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 https://github.com/cupy/cupy.git
$ cd cupy
$ pip install .
```

If you are using source tree downloaded from GitHub, you need to install Cython 0.26.1 or later (`pip install cython`).

6.5 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).

6.6 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).

6.7 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).

6.8 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
```

6.9 FAQ

6.9.1 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.

6.9.2 `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 downgrade or upgrade it.

6.9.3 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 ~.

6.9.4 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.

6.9.5 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.

6.9.6 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).

6.9.7 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*.

If you are installing CuPy on Anaconda environment, also make sure that the following packages are not installed.

- cudatoolkit
- cudnn
- nccl

Use `conda uninstall cudatoolkit cudnn nccl` to remove these package.

CHAPTER 7

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.

7.1 CuPy v4

Note: The version number has been bumped from v2 to v4 to align with the versioning of Chainer. Therefore, CuPy v3 does not exist.

7.1.1 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()
```

(continues on next page)

(continued from previous page)

```
# For performance, the size of actual allocation may become larger than the requested␣
↪array size.
print(mempool.used_bytes())   # 512
print(mempool.total_bytes())  # 512

# Even if the array goes out of scope, its memory block is kept in the pool.
a = None
print(mempool.used_bytes())   # 0
print(mempool.total_bytes())  # 512

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

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

```
import cupy
cupy.cuda.set_allocator(None)
cupy.cuda.set_pinned_memory_allocator(None)
```

7.1.2 Compute Capability

CuPy v4 now requires NVIDIA GPU with Compute Capability 3.0 or larger. See the List of CUDA GPUs to check if your GPU supports Compute Capability 3.0.

7.1.3 CUDA Stream

As CUDA Stream is fully supported in CuPy v4, `cupy.cuda.RandomState.set_stream`, the function to change the stream used by the random number generator, has been removed. Please use `cupy.cuda.Stream.use()` instead.

See the discussion in #306 for more details.

7.1.4 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.

7.2 CuPy v2

7.2.1 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 8

License

Copyright (c) 2015 Preferred Infrastructure, Inc.

Copyright (c) 2015 Preferred Networks, Inc.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

8.1 NumPy

The CuPy is designed based on NumPy’s API. CuPy’s source code and documents contain the original NumPy ones.

Copyright (c) 2005-2016, NumPy Developers.

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

8.2 SciPy

The CuPy is designed based on SciPy's API. CuPy's source code and documents contain the original SciPy ones.

Python Module Index

C

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z