CuPy Documentation

Release 1.0.0.1

Preferred Networks, inc. and Preferred Infrastructure, inc.

Jun 02, 2017

Contents

This is the CuPy documentation.

CHAPTER 1

CuPy Overview

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

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

- Basic indexing (indexing by ints, slices, newaxes, and Ellipsis)
- Element types (dtypes): bool_, (u)int{8, 16, 32, 64}, float{16, 32, 64}
- Most of the array creation routines
- Reshaping and transposition
- All operators with broadcasting
- All Universal functions (a.k.a. ufuncs) for elementwise operations except those for complex numbers
- Dot product functions (except einsum) using cuBLAS
- Reduction along axes (sum, max, argmax, etc.)

CuPy also includes following features for performance:

- Customizable memory allocator, and a simple memory pool as an example
- User-defined elementwise kernels
- User-defined reduction kernels
- 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

Installation Guide

Recommended Environments

We recommend these Linux distributions.

- Ubuntu 14.04/16.04 LTS 64bit
- CentOS 7 64bit

The following versions of Python can be used: 2.7.6+, 3.4.3+, 3.5.1+, and 3.6.0+.

Note: We are testing CuPy automatically with Jenkins, where all the above *recommended* environments are tested. We cannot guarantee that CuPy works on other environments including Windows and macOS (especially with CUDA support), even if CuPy looks running correctly.

CuPy is supported on Python 2.7.6+, 3.4.3+, 3.5.1+, 3.6.0+. CuPy uses C++ compiler such as g++. 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++
```

If you use old `setuptools`, upgrade it:

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

Dependencies

Before installing CuPy, we recommend to upgrade `setuptools` if you are using an old one:

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

The following Python packages are required to install CuPy. The latest version of each package will automatically be installed if missing.

- NumPy 1.9, 1.10, 1.11, 1.12
- Six 1.9+

CUDA support

- CUDA 7.0, 7.5, 8.0

cuDNN support

- cuDNN v4, v5, v5.1, v6

NCCL support

- nccl v1.3+

Install CuPy

Install CuPy via pip

We recommend to install CuPy via pip:

```
$ pip install cupy
```

Note: All optional CUDA related libraries, cuDNN and NCCL, need to be installed before installing CuPy. After you update these libraries, please reinstall CuPy because you need to compile and link to the newer version of them.

Install CuPy from source

The tarball of the source tree is available via `pip download cupy` or from the release notes page. You can use `setup.py` to install CuPy from the tarball:

```
$ tar zxf cupy-x.x.x.tar.gz
$ cd cupy-x.x.x
$ python setup.py install
```

You can also install the development version of CuPy from a cloned Git repository:

```
$ git clone https://github.com/pfnet/cupy.git
$ cd cupy
$ python setup.py install
```

When an error occurs...

Use `-vvvv` option with `pip` command. That shows all logs of installation. It may help you:

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

Install CuPy with CUDA

You need to install CUDA Toolkit before installing CuPy. If you have CUDA in a default directory or set `CUDA_PATH` correctly, CuPy installer finds CUDA automatically:

```
$ pip install cupy
```

Note: CuPy installer looks up `CUDA_PATH` environment variable first. If it is empty, the installer looks for `nvcc` command from `PATH` environment variable and use its parent directory as the root directory of CUDA installation. If `nvcc` command is also not found, the installer tries to use the default directory for Ubuntu `/usr/local/cuda`.

If you installed CUDA into a non-default directory, you need to specify the directory with `CUDA_PATH` environment variable:

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

Warning: If you want to use `sudo` to install CuPy, note that `sudo` command initializes all environment variables. Please specify `CUDA_PATH` environment variable inside `sudo` like this:

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

Install CuPy with cuDNN and NCCL

cuDNN is a library for Deep Neural Networks that NVIDIA provides. NCCL is a library for collective multi-GPU communication. CuPy can use cuDNN and NCCL. If you want to enable these libraries, install them before installing CuPy. We recommend you to install developer library of deb package of cuDNN and NCCL.

If you want to install tar-gz version of cuDNN, we recommend you to install it to CUDA directory. For example if you uses Ubuntu Linux, 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 other 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
```

Install CuPy for developers

CuPy uses Cython (>=0.24). Developers need to use Cython to regenerate C++ sources from `pyx` files. We recommend to use `pip` with `-e` option for editable mode:

```
$ pip install -U cython
$ cd /path/to/cupy/source
$ pip install -e .
```

Users need not to install Cython as a distribution package of CuPy only contains generated sources.

Uninstall CuPy

Use pip to uninstall CuPy:

```
$ pip uninstall cupy
```

Note: When you upgrade Chainer, `pip` sometimes install the new version without removing the old one in `site-packages`. In this case, `pip uninstall` only removes the latest one. To ensure that Chainer is completely removed, run the above command repeatedly until `pip` returns an error.

Upgrade CuPy

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

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

Reinstall CuPy

If you want to reinstall CuPy, please uninstall CuPy and then install it. We recommend to use `--no-cache-dir` option as `pip` sometimes uses cache:

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

When you install CuPy without CUDA, and after that you want to use CUDA, please reinstall CuPy. You need to reinstall CuPy when you want to upgrade CUDA.

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
```

FAQ

Warning message "cuDNN is not enabled" appears

You failed to build CuPy with cuDNN. If you don't need cuDNN, ignore this message. Otherwise, retry to install CuPy with cuDNN. `-vvvv` option helps you. See *Install CuPy with cuDNN and NCCL*.

CHAPTER 3

CuPy Tutorial

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

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:

```
```

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 array manipulations are also do in the way similar to NumPy. Take the Euclid norm (a.k.a L2 norm) for example. NumPy has *numpy.lina.g.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.

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.

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 a device

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.

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)))
```

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.

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)
```

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.

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()`.

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='f').reshape(2, 5)
>>> l2norm_kernel(x, axis=1)
array([  5.47722578,  15.96871948], 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 4

CuPy Reference Manual

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

Support features

A list of supported attributes, properties, and methods of ndarray

Memory layout

`base` `ctypes` `itemsize` `flags` `nbytes` `shape` `size` `strides`

Data type

`dtype`

Other attributes

`T`

Array conversion

`tolist()` `tofile()` `dump()` `dumps()` `astype()` `copy()` `view()` `fill()`

Shape manipulation

`reshape()` `transpose()` `swapaxes()` `ravel()` `squeeze()`

Item selection and manipulation

take() diagonal()

Calculation

max() argmax() min() argmin() clip() trace() sum() mean() var() std() prod() dot()

Arithmetic and comparison operations

__lt__() __le__() __gt__() __ge__() __eq__() __ne__() __nonzero__() __neg__() __pos__() __abs__() __invert__() __add__() __sub__() __mul__() __div__() __truediv__() __floordiv__() __mod__() __divmod__() __pow__() __lshift__() __rshift__() __and__() __or__() __xor__() __iadd__() __isub__() __imul__() __idiv__() __itruediv__() __ifloordiv__() __imod__() __ipow__() __ilshift__() __irshift__() __iand__() __ior__() __ixor__()

Special methods

__copy__() __deepcopy__() __reduce__() __array__() __len__() __getitem__() __setitem__() __int__() __long__() __float__() __oct__() __hex__() __repr__() __str__()

Memory transfer

get() set()

A list of supported routines of cupy module

Array creation routines

empty() empty_like() eye() identity() ones() ones_like() zeros() zeros_like() full() full_like()

array() asarray() ascontiguousarray() copy()

arange() linspace()

diag() diagflat()

Array manipulation routines

copyto()

reshape() ravel()

rollaxis() swapaxes() transpose()

atleast_1d() atleast_2d() atleast_3d() broadcast broadcast_arrays() broadcast_to() expand_dims() squeeze()

column_stack() concatenate() dstack() hstack() vstack()

array_split() dsplit() hsplit() split() vsplit()

roll()

Binary operations

bitwise_and bitwise_or bitwise_xor invert left_shift right_shift

Indexing routines

take() diagonal()

Input and output

load() save() savez() savez_compressed()

array_repr() array_str()

Linear algebra

dot() vdot() inner() outer() tensordot()

trace()

Logic functions

isfinite isinf isnan

logical_and logical_or logical_not logical_xor

greater greater_equal less less_equal equal not_equal

Mathematical functions

sin cos tan arcsin arccos arctan hypot arctan2 deg2rad rad2deg degrees radians

sinh cosh tanh arcsinh arccosh arctanh

rint floor ceil trunc

sum() prod()

exp expm1 exp2 log log10 log2 log1p logaddexp logaddexp2

signbit copysign ldexp frexp nextafter

add reciprocal negative multiply divide power subtract true_divide floor_divide fmod mod modf remainder

clip() sqrt square absolute sign maximum minimum fmax fmin

Sorting, searching, and counting

argmax() argmin() count_nonzero() nonzero() flatnonzero() where()

Statistics

`amin()` `amax()`

`mean()` `var()` `std()`

`bincount()`

Padding

`pad()`

External Functions

`scatter_add()`

Other

`asnumpy()`

Multi-Dimensional Array (ndarray)

class `cupy.`**`ndarray`**

Multi-dimensional array on a CUDA device.

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

Parameters

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

T

Shape-reversed view of the array.

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

argmax()

Returns the indices of the maximum along a given axis.

See also:

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

argmin()

Returns the indices of the minimum along a given axis.

See also:

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

astype()

Casts the array to given data type.

Parameters

- **dtype** – Type specifier.
- **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 order, casting, and subok arguments.

See also:

numpy.ndarray.astype()

clip()

Returns an array with values limited to [a_min, a_max].

See also:

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

copy()

Returns a copy of the array.

Parameters **order** (*{'C', 'F'}*) – Row-major (C-style) or column-major (Fortran-style) order. This function currently does not support order 'A' and 'K'.

See also:

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

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.

device

CUDA device on which this array resides.

diagonal()
Returns a view of the specified diagonals.

See also:

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

dot()
Returns the dot product with given array.

See also:

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

dump()
Dumps a pickle of the array to a file.

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

dumps()
Dumps a pickle of the array to a string.

fill()
Fills the array with a scalar value.

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

See also:

numpy.ndarray.fill()

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

flatten()
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()
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 numpy.ndarray

itemsize
Size of each element in bytes.

See also:

numpy.ndarray.itemsize

max()
Returns the maximum along a given axis.

See also:

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

mean()
Returns the mean along a given axis.

See also:

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

min()
Returns the minimum along a given axis.

See also:

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

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

nonzero()
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()

prod()
Returns the product along a given axis.

See also:

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

ravel()
Returns an array flattened into one dimension.

See also:

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

reduced_view()
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()
Returns an array with repeated arrays along an axis.

See also:

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

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

See also:

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

scatter_add()
Adds given values to specified elements of an array.

See also:

cupy.scatter_add() for full documentation.

set()
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.

shape
Lengths of axes.

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

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

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

See also:

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

squeeze()
Returns a view with size-one axes removed.

See also:

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

std()
Returns the standard deviation along a given axis.

See also:

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

strides
Strides of axes in bytes.

See also:

numpy.ndarray.strides

sum()
Returns the sum along a given axis.

See also:

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

swapaxes()
Returns a view of the array with two axes swapped.

See also:

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

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

See also:

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

tofile()
Writes the array to a file.

See also:

numpy.ndarray.tolist()

tolist()
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()
Returns the sum along diagonals of the array.

See also:

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

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

See also:

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

var()
Returns the variance along a given axis.

See also:

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

view()
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()

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

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.

Ufunc class

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.

__call__()
Applies the universal function to arguments elementwise.

Parameters

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

Returns Output array or a tuple of output arrays.

types
A list of type signatures.

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

Available ufuncs

Math operations

add subtract multiply divide logaddexp logaddexp2 true_divide floor_divide negative power remainder mod fmod absolute rint sign exp exp2 log log2 log10 expm1 log1p sqrt square reciprocal

Trigonometric functions

sin cos tan arcsin arccos arctan arctan2 hypot sinh cosh tanh arcsinh arccosh arctanh deg2rad rad2deg

Bit-twiddling functions

bitwise_and bitwise_or bitwise_xor invert left_shift right_shift

Comparison functions

greater greater_equal less less_equal not_equal equal logical_and logical_or logical_xor logical_not maximum minimum fmax fmin

Floating point values

isfinite isinf isnan signbit copysign nextafter modf ldexp frexp fmod floor ceil trunc

ufunc.at

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

Routines

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

Array Creation Routines

Basic creation routines

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**(*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**(*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**(*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**(*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**(*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**(*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 ones.

Return type *cupy.ndarray*

See also:

`numpy.zeros()`

`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 ones.

Return type *cupy.ndarray*

See also:

`numpy.zeros_like()`

`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**(*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**(*obj*, *dtype=None*, *copy=True*, *ndmin=0*)
Creates an array on the current device.

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

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.
- **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*

See also:

`numpy.array()`

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.**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**(*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`**(**args*, ***kwargs*)

Numerical ranges

`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`**(*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`**(*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**(**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**(*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**(*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*

Array Manipulation Routines

Basic manipulations

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**(*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**(*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.**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**(*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**(*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**(**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**(**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**(**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()`

class cupy.**broadcast**
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

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**(*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**(*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**(*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.**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.**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.**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.**vstack**(*tup*)
Stacks arrays vertically.

If an input array has one dimension, then the array is treated as a horizontal vector and stacked along the additional axis at the head. Otherwise, the array is stacked along the first axis.

Parameters **tup** (*sequence of arrays*) – Arrays to be stacked. Each array is converted by *cupy.atleast_2d()* before stacking.

Returns Stacked array.

Return type *cupy.ndarray*

See also:

`numpy.dstack()`

cupy.**hstack**(*tup*)
Stacks arrays horizontally.

If an input array has one dimension, then the array is treated as a horizontal vector and stacked along the first axis. Otherwise, the array is stacked along the second axis.

Parameters **tup** (*sequence of arrays*) – Arrays to be stacked.

Returns Stacked array.

Return type *cupy.ndarray*

See also:

`numpy.hstack()`

cupy.**dstack**(*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.**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()

Splitting arrays along axis

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.**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.**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()

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.**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()

Repeating part of arrays along axis

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**(*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()

Rearranging elements

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**(*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**(*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**(*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**(*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()

Binary Operations

Elementwise bit operations

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 = <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 = <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 = <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 = <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 = <ufunc 'cupy_right_shift'>**
Shifts the bits of each integer element to the right.

Only integer arrays are handled

See also:

numpy.right_shift

Indexing Routines

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.**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.**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.`**`fill_diagonal`**(*a*, *val*, *wrap=False*)

Fill 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()`

`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.

Parameters **`a function, so takes no parameters`** (*`Not`*) –

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.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.

Parameters **a function, so takes no parameters** (*Not*) –

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)
```

Input and Output

NPZ files

`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`**(*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`**(*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`**(*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`**(*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**(*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()

Linear Algebra

Matrix and vector products

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**(*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`**(*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`**(*a*, *b*, *out=None*)

Returns the outer product of two vectors.

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

Parameters

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

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

Return type *cupy.ndarray*

See also:

`numpy.outer()`

`cupy.`**`tensordot`**(*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.
- **out** (`cupy.ndarray`) – Output array.

Returns The tensor dot product of `a` and `b` along the axes specified by `axes`.

Return type *cupy.ndarray*

See also:

numpy.tensordot()

Decompositions

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)`

See also:

numpy.linalg.cholesky()

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', and decompose a matrix `A = (M, N)` into `Q`, `R` with dimensions `(M, K)`, `(K, N)`, where `K = min(M, N)`.

See also:

numpy.linalg.qr()

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.

See also:

numpy.linalg.svd()

Norms etc.

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()`

Logic Functions

Infinities and NaNs

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

See also:

`numpy.isinf`

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

See also:

`numpy.isnan`

Logic operations

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

See also:

`numpy.logical_and`

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

See also:

numpy.logical_or

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

See also:

numpy.logical_not

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

See also:

numpy.logical_xor

Comparison operations

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

See also:

numpy.greater

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

See also:

numpy.greater_equal

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

See also:

numpy.less

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

See also:

numpy.less_equal

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

See also:

numpy.equal

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

See also:

numpy.equal

Mathematical Functions

Trigonometric functions

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

See also:

numpy.sin

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

See also:

numpy.cos

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

See also:

numpy.tan

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

See also:

numpy.arcsin

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

See also:

numpy.arccos

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

See also:

numpy.arctan

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

See also:

numpy.arctan2

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

See also:

numpy.deg2rad, numpy.radians

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

See also:

numpy.rad2deg, numpy.degrees

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

See also:

numpy.rad2deg, numpy.degrees

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

See also:

numpy.deg2rad, numpy.radians

Hyperbolic functions

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

See also:

numpy.sinh

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

See also:

numpy.cosh

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

See also:

numpy.tanh

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

See also:

numpy.arcsinh

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

See also:

numpy.arccosh

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

See also:

numpy.arctanh

Rounding

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

See also:

numpy.rint

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

See also:

numpy.floor

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

See also:

numpy.ceil

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

See also:

numpy.trunc

Sums and products

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

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

Exponential and logarithm functions

cupy.**exp** = <ufunc 'cupy_exp'>
Elementwise exponential function.

See also:

numpy.exp

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

See also:

numpy.expm1

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

See also:

numpy.exp2

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

See also:

numpy.log

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

See also:

numpy.log10

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

See also:

numpy.log2

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

See also:

numpy.log1p

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

See also:

numpy.logaddexp

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

See also:

numpy.logaddexp2

Floating point manipulations

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

See also:

numpy.signbit

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

See also:

numpy.copysign

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

See also:

numpy.ldexp

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

See also:

numpy.nextafter

Arithmetic operations

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

See also:

numpy.negative

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

See also:

numpy.add

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

See also:

numpy.subtract

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

See also:

numpy.multiply

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

See also:

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

See also:

numpy.floor_divide

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

See also:

numpy.power

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

See also:

numpy.fmod

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

See also:

numpy.remainder

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

See also:

numpy.remainder

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

See also:

numpy.reciprocal

Miscellaneous

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

cupy.**sqrt = <ufunc 'cupy_sqrt'>**

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

See also:

numpy.square

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

See also:

numpy.absolute

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

If NaN appears, it returns the NaN.

See also:

numpy.maximum

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

If NaN appears, it returns the other operand.

See also:

numpy.fmax

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

If NaN appears, it returns the other operand.

See also:

numpy.fmin

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**(*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.

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**(**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** (*tuple of 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()

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** (*tuple of 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()

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

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.

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**(*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**(*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**(*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**(*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**(*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()`

Distributions

`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:

`numpy.random.gumbel()`

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**(*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**(*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**(*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**(*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**()
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*

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
```

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()

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, each element of the array is numpy.int32.

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()

rand(**size*, ***kwarg*)
Returns uniform random values over the interval [0, 1).

See also:

cupy.random.rand() for full documentation, numpy.random.RandomState.rand()

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()

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()

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()

Sorting, Searching, and Counting

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

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

Returns Array of the same type and shape as a.

Return type *cupy.ndarray*

Note: For its implementation reason, cupy.sort currently supports only arrays with their rank of one and does not support axis, kind and order parameters that numpy.sort does support.

See also:

numpy.sort()

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**(*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.**count_nonzero**(*a*, *axis=None*)
Counts the number of non-zero values in the array.

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 int or cupy.ndarray of int

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.**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()

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

Statistics

Order statistics

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

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

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**(*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**(*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**(*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**(*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`**(*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()`

External Functions

`cupy.`**`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.])
>>> cupy.scatter_add(a, i, v);
>>> a
array([ 1.,  2.,  0.,  0.,  0.,  0.], dtype=float32)
```

Parameters

- **a** (`ndarray`) – An array that gets added.

- **slices** – It is integer, slices, ellipsis, numpy.newaxis, integer array-like, boolean array-like or tuple of them. It works for slices used for `cupy.ndarray.__getitem__()` and `cupy.ndarray.__setitem__()`.
- **v** (*array-like*) – Values to increment `a` at referenced locations.

Note: It only supports types that are supported by CUDA's atomicAdd when an integer array is included in `slices`. The supported types are `numpy.float32`, `numpy.int32`, `numpy.uint32`, `numpy.uint64` and `numpy.ulonglong`.

Note: `scatter_add()` does not raise an error when indices exceed size of axes. Instead, it wraps indices.

See also:

`numpy.add.at()`.

NumPy-CuPy Generic Code Support

`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)))
```

Low-Level CUDA Support

Device management

class `cupy.cuda.`**`Device`**

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.

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.

synchronize()
Synchronizes the current thread to the device.

use()
Makes this device current.

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

Memory management

class cupy.cuda.**Memory**
Memory allocation on a CUDA device.

This class provides an RAII interface of the CUDA memory allocation.

Parameters

- **device** (cupy.cuda.Device) – Device whose memory the pointer refers to.
- **size** (*int*) – Size of the memory allocation in bytes.

class cupy.cuda.**MemoryPointer**
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*** (cupy.cuda.Device) – Device whose memory the pointer refers to.

- **mem** (Memory) – The device memory buffer.
- **ptr** (*int*) – Pointer to the place within the buffer.

copy_from()
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()
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.

copy_from_device()
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()
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.

copy_from_host()
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()
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.

copy_to_host()
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()
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.

memset()
Fills a memory sequence by constant byte value.

Parameters

- **value** (*int*) – Value to fill.
- **size** (*int*) – Size of the sequence in bytes.

memset_async()
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.

cupy.cuda.**alloc**()
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.**set_allocator**()
Sets the current allocator.

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.

class cupy.cuda.**MemoryPool**
Memory pool for all devices on the machine.

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.

free_all_blocks()
Release free blocks.

free_all_free()
Release free blocks.

malloc()
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)
```

Parameters **size** (*int*) – Size of the memory buffer to allocate in bytes.

Returns Pointer to the allocated buffer.

Return type *MemoryPointer*

n_free_blocks()
Count the total number of free blocks.

Returns The total number of free blocks.

Return type int

Streams and events

class cupy.cuda.**Stream**(*null=False*, *non_blocking=False*)
CUDA stream.

This class handles the CUDA stream handle in RAII way, i.e., when an Stream instance is destroyed by the GC, its handle is also destroyed.

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.
- **non_blocking** (*bool*) – If `True`, the stream does not synchronize with the NULL stream.

Variables **ptr** (*cupy.cuda.runtime.Stream*) – Raw stream handle. It can be passed to the CUDA Runtime API via ctypes.

add_callback(*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.

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

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

wait_event(*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.

class cupy.cuda.**Event**(*block=False*, *disable_timing=False*, *interprocess=False*)
CUDA event, a synchronization point of CUDA streams.

This class handles the CUDA event handle in RAII way, i.e., when an Event instance is destroyed by the GC, its handle is also destroyed.

Parameters

- **block** (*bool*) – If True, the event blocks on the *synchronize()* method.
- **disable_timing** (*bool*) – If True, the event does not prepare the timing data.
- **interprocess** (*bool*) – If True, the event can be passed to other processes.

Variables **ptr** (*cupy.cuda.runtime.Stream*) – Raw stream handle. It can be passed to the CUDA Runtime API via ctypes.

done
True if the event is done.

record(*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()
Synchronizes all device work to the event.

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

cupy.cuda.**get_elapsed_time**(*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

Profiler

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**()
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 coutput file.
- **output_mode** (*int*) – cupy.cuda.profiler.cudaKeyValuePair or cupy.cuda.profiler.cudaCSV.

cupy.cuda.profiler.**start**()
Enable profiling.

A user can enable CUDA profiling. When an error occurs, it raises an exception.

See the CUDA document for detail.

cupy.cuda.profiler.**stop**()
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**()
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**()
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**()
Starts a nestead 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**()
Starts a nestead 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**()
Ends a nestead 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.

Kernel binary memoization

cupy.**memoize**()
Makes a function memoizing the result for each argument and device.

This decorator provides automatic memoization of the function result.

Parameters **for_each_device** (*bool*) – If `True`, it memoizes the results for each device. Otherwise, it memoizes the results only based on the arguments.

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

class cupy.**ElementwiseKernel**
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.

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.

__call__()
Compiles and invokes the reduction kernel.

The compilation runs only if the kernel is not cached. Note that the kernels with different argument dtypes, ndims, or axis are not compatible. It means that single ReductionKernel object may be compiled into multiple kernel binaries.

Parameters **args** – Arguments of the kernel.

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

Testing Modules

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

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**(*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**(*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**(*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**(*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**(*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**(*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**(*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()

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**(*rtol=1e-07*, *atol=0*, *err_msg=''*, *verbose=True*, *name='xp'*, *type_check=True*, *accept_error=False*)
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*) – Sepcify 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 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**(*decimal=6*, *err_msg=''*, *verbose=True*, *name='xp'*, *type_check=True*, *accept_error=False*)

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*) – Sepcify 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 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**(*nulp=1*, *name='xp'*, *type_check=True*, *accept_error=False*)

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*) – Sepcify 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 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**(*maxulp=1*, *dtype=None*, *name='xp'*, *type_check=True*, *accept_error=False*)

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*) – Sepcify 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 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**(*err_msg=''*, *verbose=True*, *name='xp'*, *type_check=True*, *accept_error=False*)

Decorator that checks NumPy results and CuPy ones are equal.

Parameters

- **err_msg** (*str*) – The error message to be printed in case of failure.
- **verbose** (*bool*) – If `True`, the conflicting values are appended to the error message.
- **name** (*str*) – Argument name whose value is either `numpy` or `cupy` module.
- **type_check** (*bool*) – If `True`, consistency of dtype is also checked.
- **accept_error** (*bool, Exception or tuple of Exception*) – Sepcify 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 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**(*err_msg=''*, *verbose=True*, *name='xp'*)

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.

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**(*err_msg=''*, *verbose=True*, *name='xp'*, *type_check=True*, *accept_error=False*)

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*) – Sepcify 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 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**(*name='xp'*)
Decorator that checks the NumPy and CuPy throw same errors.

Parameters

- **name** (*str*) – Argument name whose value is either
- **or cupy module.** (*numpy*) –

Decorated test fixture is required throw same errors even if `xp` is `numpy` or `cupy`.

See also:

cupy.testing.assert_array_less()

Parameterized dtype Test

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

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**(*name='dtype'*, *no_float16=False*, *no_bool=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.

dtypes to be tested: `numpy.float16` (optional), `numpy.float32`, `numpy.float64`, `numpy.dtype('b')`, `numpy.dtype('h')`, `numpy.dtype('i')`, `numpy.dtype('l')`, `numpy.dtype('q')`, `numpy.dtype('B')`, `numpy.dtype('H')`, `numpy.dtype('I')`, `numpy.dtype('L')`, `numpy.dtype('Q')`, and `numpy.bool_` (optional).

The usage is as follows. This test fixture checks if `cPickle` successfully reconstructs *cupy.ndarray* for various dtypes. `dtype` is an argument inserted by the decorator.

```
>>> import unittest
>>> from cupy import testing
>>> @testing.gpu
... class TestNpz(unittest.TestCase):
...
...     @testing.for_all_dtypes()
...     def test_pickle(self, dtype):
...         a = testing.shaped_arange((2, 3, 4), dtype=dtype)
...         s = six.moves.cPickle.dumps(a)
...         b = six.moves.cPickle.loads(s)
...         testing.assert_array_equal(a, b)
```

Typically, we use this decorator in combination with decorators that check consistency between NumPy and CuPy like *cupy.testing.numpy_cupy_allclose()*. The following is such an example.

```
>>> import unittest
>>> from cupy import testing
>>> @testing.gpu
... class TestMean(unittest.TestCase):
...
...     @testing.for_all_dtypes()
...     @testing.numpy_cupy_allclose()
...     def test_mean_all(self, xp, dtype):
...         a = testing.shaped_arange((2, 3), xp, dtype)
...         return a.mean()
```

See also:

cupy.testing.for_dtypes()

cupy.testing.**for_float_dtypes**(*name='dtype'*, *no_float16=False*)
Decorator that checks the fixture with all 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**(*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**(*name='dtype'*)
Decorator that checks the fixture with all 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**(*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_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**(*names=('dtyes',)*, *no_float16=False*, *no_bool=False*, *full=None*)

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()*).

See also:

cupy.testing.for_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`**(*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`**(*names=('dtype',)*, *no_bool=False*, *full=None*)
Decorator for parameterized test w.r.t. the product set of int and boolean.

Parameters

- **names** (*list of str*) – Argument names to which dtypes are passed.
- **no_bool** (*bool*) – If `True`, `numpy.bool_` is omitted from candidate dtypes.
- **full** (*bool*) – If `True`, then all combinations of dtypes will be tested. Otherwise, the subset of combinations will be tested (see description in *cupy.testing.for_dtypes_combination()*).

See also:

cupy.testing.for_dtypes_combination()

Parameterized order Test

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

`cupy.testing.`**`for_orders`**(*orders*, *name='order'*)
Decorator to parameterize tests with order.

Parameters

- **orders** (*list of orders*) – 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`**(*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()

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 cupy-overview for detail.

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 *Install CuPy with CUDA* for details.

Difference between CuPy and NumPy

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

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 eamples. The behavior of NumPy depends on your CPU architecture. This is Intel CPU result.

```
>>> np.array([-1], dtype='f').astype('I')
array([4294967295], dtype=uint32)
>>> cupy.array([-1], dtype='f').astype('I')
array([0], dtype=uint32)
```

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

Boolean values squared

In NumPy implementation, `x ** 2` is calculated using multiplication operator as `x * x`. Because the result of the multiplication of boolean values is boolean, `True ** 2` return boolean value. However, when you use power operator with other arguments, it returns int values. If we aligned the behavior of the squared boolean values of CuPy to that of NumPy, we would have to check their values in advance of the calculation. But it would be slow because it would force CPUs to wait until the calculation on GPUs end. So we decided not to check its value.

```
>>> np.array([True]) ** 2
array([ True], dtype=bool)
>>> cupy.array([True]) ** 2
array([1])
```

Random methods support dtype argument

NumPy's random value generator does not support dtype option and it always resturns 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='f')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: randn() got an unexpected keyword argument 'dtype'
>>> cupy.random.randn(dtype='f')
array(0.10689262300729752, dtype=float32)
```

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])
```

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.float)
>>> 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.float)
>>> a_cpu[i_cpu] = v_cpu
>>> a_cpu
array([ 9998.,  9999.])
```

CHAPTER 5

CuPy 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.

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.

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*.

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.

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.

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 the nose package and the mock package for testing, so install nose and mock before writing your code:

```
$ pip install nose 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:

```
$ python setup.py develop
```

Once the Cython modules are built, you can run unit tests simply by running `nosetests` command at the repository root:

```
$ nosetests
```

It requires CUDA by default. In order to run unit tests that do not require CUDA, pass `--attr='!gpu'` option to the `nosetests` command:

```
$ nosetests path/to/your/test.py --attr='!gpu'
```

Some GPU tests involve multiple GPUs. If you want to run GPU tests with insufficient number of GPUs, specify the number of available GPUs by `--eval-attr='gpu<N'` where `N` is a concrete integer. For example, if you have only one GPU, launch `nosetests` by the following command to skip multi-GPU tests:

```
$ nosetests path/to/gpu/test.py --eval-attr='gpu<2'
```

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 just typing `nosetests` at the repository root:

```
$ nosetests
```

Or you can also specify a root directory to search test scripts from:

```
$ nosetests tests/cupy_tests     # to just run tests of CuPy
$ nosetests 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.

Note: CuPy tests include type-exhaustive test functions which take long time to execute. If you are running tests on a multi-core machine, you can parallelize the tests by following options:

```
$ nosetests --processes=12 --process-timeout=1000 tests/cupy_tests
```

The magic numbers can be modified for your usage. Note that some tests require many CUDA compilations, which require a bit long time. Without the `process-timeout` option, the timeout is set shorter, causing timeout failures for many test cases.

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 `--attr='!gpu'` is given. We also have the `cupy.testing.attr.cudnn` decorator to let `nosetests` know that the test depends on cuDNN.

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, your code is automatically tested by Travis-CI **with –attr='!gpu,!slow' option**. Since Travis-CI does not support CUDA, we cannot check your CUDA-related code automatically. The reviewing process starts after the test passes. Note that reviewers will test your code without the option to check CUDA-related code.

Note: Some of numerically unstable tests might cause errors irrelevant to your changes. In such a case, we ignore the failures and go on to the review process, so do not worry about it.

CHAPTER 6

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.

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.

Processes to Break Backward Compatibilities

Deprecation, Dropping, and Its Preparation

Any APIs may be *deprecated* at some minor updates. In such a case, the deprecation note is added to the API documentation, and the API implementation is changed to fire 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.

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.

Supported Backward Compatibility

This section defines backward compatibilities that minor updates must maintain.

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.

Undocumented behaviors

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

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

Revision update may also contain changes to undefined behaviors. Typical example is a bug fix. Another example is an improvement on implementation, which may change the internal object structures not shown in the documentation. As

a consequence, **even revision updates do not support compatibility of pickling, unless the full layout of pickled objects is clearly documented.**

Documentation Error

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

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

Object Attributes and Properties

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

Functions and Methods

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

Exceptions and Warnings

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

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

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 7

License

CuPy

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

- Neither the name of the NumPy Developers nor the names of any contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

C

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

Z