scikit-cuda Documentation

Release 0.5.2

Lev Givon

Sep 18, 2018

Contents

scikit-cuda provides Python interfaces to many of the functions in the CUDA device/runtime, CUBLAS, CUFFT, and CUSOLVER libraries distributed as part of NVIDIA's CUDA Programming Toolkit, as well as interfaces to select functions in the CULA Dense Toolkit. Both low-level wrapper functions similar to their C counterparts and high-level functions comparable to those in NumPy and Scipy are provided.

Python wrappers for cuDNN by Hannes Bretschneider are available here.

CHAPTER 1

Contents

1.1 Installation

1.1.1 Quick Installation

If you have pip installed, you should be able to install the latest stable release of `scikit-cuda` by running the following:

```
pip install scikit-cuda
```

All dependencies should be automatically downloaded and installed if they are not already on your system.

1.1.2 Obtaining the Latest Software

The latest stable and development versions of `scikit-cuda` can be downloaded from GitHub

Online documentation is available at https://scikit-cuda.readthedocs.org

1.1.3 Installation Dependencies

`scikit-cuda` requires that the following software packages be installed:

- Python 2.7 or 3.4.
- Setuptools 0.6c10 or later.
- Mako 1.0.1 or later.
- NumPy 1.2.0 or later.
- PyCUDA 2016.1 or later (some parts of `scikit-cuda` might not work properly with earlier versions).
- NVIDIA CUDA Toolkit 5.0 or later.

Note that both Python and the CUDA Toolkit must be built for the same architecture, i.e., Python compiled for a 32-bit architecture will not find the libraries provided by a 64-bit CUDA installation. CUDA versions from 7.0 onwards are 64-bit.

To run the unit tests, the following packages are also required:

- nose 0.11 or later.
- SciPy 0.14.0 or later.

Some of the linear algebra functionality relies on the CULA toolkit; as of 2017, it is available to premium tier users of E.M. Photonics' HPC site Celerity Tools:

- CULA R16a or later.

To build the documentation, the following packages are also required:

- Docutils 0.5 or later.
- Jinja2 2.2 or later.
- Pygments 0.8 or later.
- Sphinx 1.0.1 or later.
- Sphinx ReadTheDocs Theme 0.1.6 or later.

1.1.4 Platform Support

The software has been developed and tested on Linux; it should also work on other Unix-like platforms supported by the above packages. Parts of the package may work on Windows as well, but remain untested.

1.1.5 Building and Installation

`scikit-cuda` searches for CUDA libraries in the system library search path when imported. You may have to modify this path (e.g., by adding the path to the CUDA libraries to `/etc/ld.so.conf` and running `ldconfig` as root or to the `LD_LIBRARY_PATH` environmental variable on Linux, or by adding the CUDA library path to the `DYLD_LIBRARY_PATH` on MacOSX) if the libraries are not being found.

To build and install the toolbox, download and unpack the source release and run:

```
python setup.py install
```

from within the main directory in the release. To rebuild the documentation, run:

```
python setup.py build_sphinx
```

1.1.6 Running the Unit Tests

To run all of the package unit tests, download and unpack the package source tarball and run:

```
python setup.py test
```

from within the main directory in the archive. Tests for individual modules (found in the `tests/` subdirectory) can also be run directly.

1.1.7 Getting Started

The functions provided by `scikit-cuda` are grouped into several submodules in the `skcuda` namespace package. Sample code demonstrating how to use different parts of the toolbox is located in the `demos/` subdirectory of the source release. Many of the high-level functions also contain doctests that describe their usage.

1.2 Reference

1.2.1 Library Wrapper Routines

CUBLAS Routines

Helper Routines

cublasCheckStatus	Raise CUBLAS exception
cublasCreate	Initialize CUBLAS.
cublasDestroy	Release CUBLAS resources.
cublasGetCurrentCtx	Get current CUBLAS context.
cublasGetStream	Set current CUBLAS library stream.
cublasGetVersion	Get CUBLAS version.
cublasSetStream	Set current CUBLAS library stream.

skcuda.cublas.cublasCheckStatus

skcuda.cublas.**cublasCheckStatus**(*status*)

Raise CUBLAS exception

Raise an exception corresponding to the specified CUBLAS error code.

Parameters **status** (*int*) – CUBLAS error code.

See also:

`cublasExceptions()`

skcuda.cublas.cublasCreate

skcuda.cublas.**cublasCreate**()

Initialize CUBLAS.

Initializes CUBLAS and creates a handle to a structure holding the CUBLAS library context.

Returns **handle** – CUBLAS context.

Return type int

References

cublasCreate

skcuda.cublas.cublasDestroy

skcuda.cublas.**cublasDestroy**(*handle*)

Release CUBLAS resources.

Releases hardware resources used by CUBLAS.

Parameters **handle** (*int*) – CUBLAS context.

References

cublasDestroy

skcuda.cublas.cublasGetCurrentCtx

skcuda.cublas.**cublasGetCurrentCtx**()

Get current CUBLAS context.

Returns the current context used by CUBLAS.

Returns **handle** – CUBLAS context.

Return type int

skcuda.cublas.cublasGetStream

skcuda.cublas.**cublasGetStream**(*handle*)

Set current CUBLAS library stream.

Parameters **handle** (*int*) – CUBLAS context.

Returns **id** – Stream ID.

Return type int

References

cublasGetStream

skcuda.cublas.cublasGetVersion

skcuda.cublas.**cublasGetVersion**(*handle*)

Get CUBLAS version.

Returns version number of installed CUBLAS libraries.

Parameters **handle** (*int*) – CUBLAS context.

Returns **version** – CUBLAS version.

Return type int

References

cublasGetVersion

skcuda.cublas.cublasSetStream

skcuda.cublas.**cublasSetStream**(*handle*, *id*)
Set current CUBLAS library stream.

Parameters

- **handle** (*id*) – CUBLAS context.
- **id** (*int*) – Stream ID.

References

cublasSetStream

Wrapper Routines

Single Precision BLAS1 Routines

cublasIsamax	Index of maximum magnitude element.
cublasIsamin	Index of minimum magnitude element (single precision real).
cublasSasum	Sum of absolute values of single precision real vector.
cublasSaxpy	Vector addition (single precision real).
cublasScopy	Vector copy (single precision real)
cublasSdot	Vector dot product (single precision real)
cublasSnrm2	Euclidean norm (2-norm) of real vector.
cublasSrot	Apply a real rotation to real vectors (single precision)
cublasSrotg	Construct a single precision real Givens rotation matrix.
cublasSrotm	Apply a single precision real modified Givens rotation.
cublasSrotmg	Construct a single precision real modified Givens rotation matrix.
cublasSscal	Scale a single precision real vector by a single precision real scalar.
cublasSswap	Swap single precision real vectors.
cublasCaxpy	Vector addition (single precision complex).
cublasCcopy	Vector copy (single precision complex)
cublasCdotc	Vector dot product (single precision complex)
cublasCdotu	Vector dot product (single precision complex)
cublasCrot	Apply a complex rotation to complex vectors (single precision)

Continued on next page

Table 2 – continued from previous page

cublasCrotg	Construct a single precision complex Givens rotation matrix.
cublasCscal	Scale a single precision complex vector by a single precision complex scalar.
cublasCsrot	Apply a complex rotation to complex vectors (single precision)
cublasCsscal	Scale a single precision complex vector by a single precision real scalar.
cublasCswap	Swap single precision complex vectors.
cublasIcamax	Index of maximum magnitude element.
cublasIcamin	Index of minimum magnitude element (single precision complex).
cublasScasum	Sum of absolute values of single precision complex vector.
cublasScnrm2	Euclidean norm (2-norm) of real vector.

skcuda.cublas.cublasIsamax

skcuda.cublas.**cublasIsamax**(*handle*, *n*, *x*, *incx*)

Index of maximum magnitude element.

Finds the smallest index of the maximum magnitude element of a single precision real vector.

Note: for complex arguments x, the "magnitude" is defined as *abs(x.real) + abs(x.imag)*, *not* as *abs(x)*.

Parameters

- **handle** (*int*) – CUBLAS context.
- **n** (*int*) – Number of elements in input vector.
- **x** (*ctypes.c_void_p*) – Pointer to single precision real input vector.
- **incx** (*int*) – Storage spacing between elements of *x*.

Returns **idx** – Index of maximum magnitude element.

Return type int

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> x = np.random.rand(5).astype(np.float32)
>>> x_gpu = gpuarray.to_gpu(x)
>>> h = cublasCreate()
>>> m = cublasIsamax(h, x_gpu.size, x_gpu.gpudata, 1)
>>> cublasDestroy(h)
>>> np.allclose(m, np.argmax(abs(x.real) + abs(x.imag)))
True
```

Notes

This function returns a 0-based index.

References

cublasI<t>amax

skcuda.cublas.cublasIsamin

skcuda.cublas.**cublasIsamin**(*handle*, *n*, *x*, *incx*)

Index of minimum magnitude element (single precision real).

Finds the smallest index of the minimum magnitude element of a single precision real vector.

Note: for complex arguments x, the "magnitude" is defined as *abs(x.real) + abs(x.imag)*, *not* as *abs(x)*.

Parameters

- **handle** (*int*) – CUBLAS context.
- **n** (*int*) – Number of elements in input vector.
- **x** (*ctypes.c_void_p*) – Pointer to single precision real input vector.
- **incx** (*int*) – Storage spacing between elements of *x*.

Returns **idx** – Index of minimum magnitude element.

Return type int

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> x = np.random.rand(5).astype(np.float32)
>>> x_gpu = gpuarray.to_gpu(x)
>>> h = cublasCreate()
>>> m = cublasIsamin(h, x_gpu.size, x_gpu.gpudata, 1)
>>> cublasDestroy(h)
>>> np.allclose(m, np.argmin(abs(x.real) + abs(x.imag)))
True
```

Notes

This function returns a 0-based index.

References

cublasI<t>amin

skcuda.cublas.cublasSasum

skcuda.cublas.**cublasSasum**(*handle*, *n*, *x*, *incx*)

Sum of absolute values of single precision real vector.

Computes the sum of the absolute values of the elements of a single precision real vector.

Note: if the vector is complex, then this computes the sum *sum(abs(x.real)) + sum(abs(x.imag))*

Parameters

- **handle** (*int*) – CUBLAS context.
- **n** (*int*) – Number of elements in input vector.
- **x** (*ctypes.c_void_p*) – Pointer to single precision real input vector.
- **incx** (*int*) – Storage spacing between elements of *x*.

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> x = np.random.rand(5).astype(np.float32)
>>> x_gpu = gpuarray.to_gpu(x)
>>> h = cublasCreate()
>>> s = cublasSasum(h, x_gpu.size, x_gpu.gpudata, 1)
>>> cublasDestroy(h)
>>> np.allclose(s, abs(x.real).sum() + abs(x.imag).sum())
True
```

Returns **s** – Sum of absolute values.

Return type numpy.float32

References

cublas<t>sum

skcuda.cublas.cublasSaxpy

skcuda.cublas.**cublasSaxpy**(*handle*, *n*, *alpha*, *x*, *incx*, *y*, *incy*)

Vector addition (single precision real).

Computes the sum of a single precision real vector scaled by a single precision real scalar and another single precision real vector.

Parameters

- **handle** (*int*) – CUBLAS context.
- **n** (*int*) – Number of elements in input vectors.
- **alpha** (*numpy.float32*) – Scalar.
- **x** (*ctypes.c_void_p*) – Pointer to single precision input vector.
- **incx** (*int*) – Storage spacing between elements of *x*.
- **y** (*ctypes.c_void_p*) – Pointer to single precision input/output vector.
- **incy** (*int*) – Storage spacing between elements of *y*.

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> alpha = np.float32(np.random.rand())
>>> x = np.random.rand(5).astype(np.float32)
>>> y = np.random.rand(5).astype(np.float32)
>>> x_gpu = gpuarray.to_gpu(x)
>>> y_gpu = gpuarray.to_gpu(y)
>>> h = cublasCreate()
>>> cublasSaxpy(h, x_gpu.size, alpha, x_gpu.gpudata, 1, y_gpu.gpudata, 1)
>>> cublasDestroy(h)
>>> np.allclose(y_gpu.get(), alpha*x+y)
True
```

Notes

Both *x* and *y* must contain *n* elements.

References

cublas<t>axpy

skcuda.cublas.cublasScopy

skcuda.cublas.**cublasScopy**(*handle*, *n*, *x*, *incx*, *y*, *incy*)

Vector copy (single precision real)

Copies a single precision real vector to another single precision real vector.

Parameters

- **handle** (*int*) – CUBLAS context.
- **n** (*int*) – Number of elements in input vectors.
- **x** (*ctypes.c_void_p*) – Pointer to single precision real input vector.
- **incx** (*int*) – Storage spacing between elements of *x*.
- **y** (*ctypes.c_void_p*) – Pointer to single precision real output vector.
- **incy** (*int*) – Storage spacing between elements of *y*.

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> x = np.random.rand(5).astype(np.float32)
>>> x_gpu = gpuarray.to_gpu(x)
>>> y_gpu = gpuarray.zeros_like(x_gpu)
```

(continues on next page)

(continued from previous page)

```
>>> h = cublasCreate()
>>> cublasScopy(h, x_gpu.size, x_gpu.gpudata, 1, y_gpu.gpudata, 1)
>>> cublasDestroy(h)
>>> np.allclose(y_gpu.get(), x_gpu.get())
True
```

Notes

Both *x* and *y* must contain *n* elements.

References

cublas<t>copy

skcuda.cublas.cublasSdot

skcuda.cublas.**cublasSdot**(*handle*, *n*, *x*, *incx*, *y*, *incy*)

Vector dot product (single precision real)

Computes the dot product of two single precision real vectors. cublasCdotc and cublasZdotc use the conjugate of the first vector when computing the dot product.

Parameters

- **handle** (*int*) – CUBLAS context.
- **n** (*int*) – Number of elements in input vectors.
- **x** (*ctypes.c_void_p*) – Pointer to single precision real input vector.
- **incx** (*int*) – Storage spacing between elements of *x*.
- **y** (*ctypes.c_void_p*) – Pointer to single precision real input/output vector.
- **incy** (*int*) – Storage spacing between elements of *y*.

Returns **d** – Dot product of *x* and *y*.

Return type np.float32

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> x = np.float32(np.random.rand(5))
>>> y = np.float32(np.random.rand(5))
>>> x_gpu = gpuarray.to_gpu(x)
>>> y_gpu = gpuarray.to_gpu(y)
>>> h = cublasCreate()
>>> d = cublasSdot(h, x_gpu.size, x_gpu.gpudata, 1, y_gpu.gpudata, 1)
>>> cublasDestroy(h)
>>> np.allclose(d, np.dot(x, y))
True
```

Notes

Both *x* and *y* must contain *n* elements.

References

cublas<t>dot

skcuda.cublas.cublasSnrm2

skcuda.cublas.**cublasSnrm2**(*handle*, *n*, *x*, *incx*)

Euclidean norm (2-norm) of real vector.

Computes the Euclidean norm of a single precision real vector.

Parameters

- **handle** (*int*) – CUBLAS context.
- **n** (*int*) – Number of elements in input vectors.
- **x** (*ctypes.c_void_p*) – Pointer to single precision real input vector.
- **incx** (*int*) – Storage spacing between elements of *x*.

Returns **nrm** – Euclidean norm of *x*.

Return type numpy.float32

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> x = np.float32(np.random.rand(5))
>>> x_gpu = gpuarray.to_gpu(x)
>>> h = cublasCreate()
>>> nrm = cublasSnrm2(h, x.size, x_gpu.gpudata, 1)
>>> cublasDestroy(h)
>>> np.allclose(nrm, np.linalg.norm(x))
True
```

References

cublas<t>nrm2

skcuda.cublas.cublasSrot

skcuda.cublas.**cublasSrot**(*handle*, *n*, *x*, *incx*, *y*, *incy*, *c*, *s*)

Apply a real rotation to real vectors (single precision)

Multiplies the single precision matrix *[[c, s], [-s.conj(), c]]* with the 2 x *n* single precision matrix *[[x.T], [y.T]]*.

Parameters

- **handle** (*int*) – CUBLAS context.
- **n** (*int*) – Number of elements in input vectors.
- **x** (*ctypes.c_void_p*) – Pointer to single precision real input/output vector.
- **incx** (*int*) – Storage spacing between elements of *x*.
- **y** (*ctypes.c_void_p*) – Pointer to single precision real input/output vector.
- **incy** (*int*) – Storage spacing between elements of *y*.
- **c** (*numpy.float32*) – Element of rotation matrix.
- **s** (*numpy.float32*) – Element of rotation matrix.

Notes

Both *x* and *y* must contain *n* elements.

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> s = np.float32(np.random.rand()); c = np.float32(np.random.rand());
>>> x = np.random.rand(5).astype(np.float32)
>>> y = np.random.rand(5).astype(np.float32)
>>> x_gpu = gpuarray.to_gpu(x)
>>> y_gpu = gpuarray.to_gpu(y)
>>> h = cublasCreate()
>>> cublasSrot(h, x.size, x_gpu.gpudata, 1, y_gpu.gpudata, 1, c, s)
>>> cublasDestroy(h)
>>> np.allclose(x_gpu.get(), c*x+s*y)
True
>>> np.allclose(y_gpu.get(), -s.conj()*x+c*y)
True
```

References

cublas<t>rot

skcuda.cublas.cublasSrotg

skcuda.cublas.**cublasSrotg**(*handle*, *a*, *b*)

Construct a single precision real Givens rotation matrix.

Constructs the single precision real Givens rotation matrix *G = [[c, s], [-s.conj(), c]]* such that *dot(G, [[a], [b]] == [[r], [0]]*, where *c**2+s**2 == 1* and *r == a**2+b**2* for real numbers and *c**2+(conj(s)*s) == 1* and *r == (a/abs(a))*sqrt(abs(a)**2+abs(b)**2)* for *a != 0* and *r == b* for *a == 0*.

Parameters

- **handle** (*int*) – CUBLAS context.

- **b** (*a,*) – Entries of vector whose second entry should be zeroed out by the rotation.

Returns

- **r** (*numpy.float32*) – Defined above.
- **c** (*numpy.float32*) – Cosine component of rotation matrix.
- **s** (*numpy.float32*) – Sine component of rotation matrix.

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> a = np.float32(np.random.rand())
>>> b = np.float32(np.random.rand())
>>> h = cublasCreate()
>>> r, c, s = cublasSrotg(h, a, b)
>>> cublasDestroy(h)
>>> np.allclose(np.dot(np.array([[c, s], [-np.conj(s), c]]), np.array([[a],␣
↪[b]])), np.array([[r], [0.0]]), atol=1e-6)
True
```

References

cublas<t>rotg

skcuda.cublas.cublasSrotm

skcuda.cublas.**cublasSrotm**(*handle*, *n*, *x*, *incx*, *y*, *incy*, *sparam*)

Apply a single precision real modified Givens rotation.

Applies the single precision real modified Givens rotation matrix *h* to the 2 x *n* matrix *[[x.T], [y.T]]*.

Parameters

- **handle** (*int*) – CUBLAS context.
- **n** (*int*) – Number of elements in input vectors.
- **x** (*ctypes.c_void_p*) – Pointer to single precision real input/output vector.
- **incx** (*int*) – Storage spacing between elements of *x*.
- **y** (*ctypes.c_void_p*) – Pointer to single precision real input/output vector.
- **incy** (*int*) – Storage spacing between elements of *y*.
- **sparam** (*numpy.ndarray*) – sparam[0] contains the *flag* described below; sparam[1:5] contains the values *[h00, h10, h01, h11]* that determine the rotation matrix *h*.

Notes

The rotation matrix may assume the following values:

for *flag* == -1.0, *h* == *[[h00, h01], [h10, h11]]* for *flag* == 0.0, *h* == *[[1.0, h01], [h10, 1.0]]* for *flag* == 1.0, *h* == *[[h00, 1.0], [-1.0, h11]]* for *flag* == -2.0, *h* == *[[1.0, 0.0], [0.0, 1.0]]*

Both *x* and *y* must contain *n* elements.

References

cublas<t>srotm

skcuda.cublas.cublasSrotmg

skcuda.cublas.**cublasSrotmg**(*handle*, *d1*, *d2*, *x1*, *y1*)

Construct a single precision real modified Givens rotation matrix.

Constructs the single precision real modified Givens rotation matrix *h = [[h11, h12], [h21, h22]]* that zeros out the second entry of the vector *[[sqrt(d1)*x1], [sqrt(d2)*x2]]*.

Parameters

- **handle** (*int*) – CUBLAS context.
- **d1** (*numpy.float32*) – single precision real value.
- **d2** (*numpy.float32*) – single precision real value.
- **x1** (*numpy.float32*) – single precision real value.
- **x2** (*numpy.float32*) – single precision real value.

Returns **sparam** – sparam[0] contains the *flag* described below; sparam[1:5] contains the values *[h00, h10, h01, h11]* that determine the rotation matrix *h*.

Return type numpy.ndarray

Notes

The rotation matrix may assume the following values:

for *flag* == -1.0, *h* == *[[h00, h01], [h10, h11]]* for *flag* == 0.0, *h* == *[[1.0, h01], [h10, 1.0]]* for *flag* == 1.0, *h* == *[[h00, 1.0], [-1.0, h11]]* for *flag* == -2.0, *h* == *[[1.0, 0.0], [0.0, 1.0]]*

References

cublas<t>rotmg

skcuda.cublas.cublasSscal

skcuda.cublas.**cublasSscal**(*handle*, *n*, *alpha*, *x*, *incx*)

Scale a single precision real vector by a single precision real scalar.

Replaces a single precision real vector *x* with *alpha * x*.

Parameters

- **handle** (*int*) – CUBLAS context.
- **n** (*int*) – Number of elements in input vectors.
- **alpha** (*numpy.float32*) – Scalar multiplier.
- **x** (*ctypes.c_void_p*) – Pointer to single precision real input/output vector.
- **incx** (*int*) – Storage spacing between elements of *x*.

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> x = np.random.rand(5).astype(np.float32)
>>> x_gpu = gpuarray.to_gpu(x)
>>> alpha = np.float32(np.random.rand())
>>> h = cublasCreate()
>>> cublasSscal(h, x.size, alpha, x_gpu.gpudata, 1)
>>> cublasDestroy(h)
>>> np.allclose(x_gpu.get(), alpha*x)
True
```

References

cublas<t>scal

skcuda.cublas.cublasSswap

skcuda.cublas.**cublasSswap**(*handle*, *n*, *x*, *incx*, *y*, *incy*)

Swap single precision real vectors.

Swaps the contents of one single precision real vector with those of another single precision real vector.

Parameters

- **handle** (*int*) – CUBLAS context.
- **n** (*int*) – Number of elements in input vectors.
- **x** (*ctypes.c_void_p*) – Pointer to single precision real input/output vector.
- **incx** (*int*) – Storage spacing between elements of *x*.
- **y** (*ctypes.c_void_p*) – Pointer to single precision real input/output vector.
- **incy** (*int*) – Storage spacing between elements of *y*.

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> x = np.random.rand(5).astype(np.float32)
>>> y = np.random.rand(5).astype(np.float32)
>>> x_gpu = gpuarray.to_gpu(x)
>>> y_gpu = gpuarray.to_gpu(y)
>>> h = cublasCreate()
>>> cublasSswap(h, x.size, x_gpu.gpudata, 1, y_gpu.gpudata, 1)
>>> cublasDestroy(h)
>>> np.allclose(x_gpu.get(), y)
True
>>> np.allclose(y_gpu.get(), x)
True
```

Notes

Both *x* and *y* must contain *n* elements.

References

cublas<t>swap

skcuda.cublas.cublasCaxpy

skcuda.cublas.**cublasCaxpy**(*handle*, *n*, *alpha*, *x*, *incx*, *y*, *incy*)

Vector addition (single precision complex).

Computes the sum of a single precision complex vector scaled by a single precision complex scalar and another single precision complex vector.

Parameters

- **handle** (*int*) – CUBLAS context.
- **n** (*int*) – Number of elements in input vectors.
- **alpha** (*numpy.complex64*) – Scalar.
- **x** (*ctypes.c_void_p*) – Pointer to single precision input vector.
- **incx** (*int*) – Storage spacing between elements of *x*.
- **y** (*ctypes.c_void_p*) – Pointer to single precision input/output vector.
- **incy** (*int*) – Storage spacing between elements of *y*.

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> alpha = np.complex64(np.random.rand()+1j*np.random.rand())
>>> x = (np.random.rand(5)+1j*np.random.rand(5)).astype(np.complex64)
```

(continues on next page)

(continued from previous page)

```
>>> y = (np.random.rand(5)+1j*np.random.rand(5)).astype(np.complex64)
>>> x_gpu = gpuarray.to_gpu(x)
>>> y_gpu = gpuarray.to_gpu(y)
>>> h = cublasCreate()
>>> cublasCaxpy(h, x_gpu.size, alpha, x_gpu.gpudata, 1, y_gpu.gpudata, 1)
>>> cublasDestroy(h)
>>> np.allclose(y_gpu.get(), alpha*x+y)
True
```

Notes

Both *x* and *y* must contain *n* elements.

References

cublas<t>axpy

skcuda.cublas.cublasCcopy

skcuda.cublas.**cublasCcopy**(*handle*, *n*, *x*, *incx*, *y*, *incy*)

Vector copy (single precision complex)

Copies a single precision complex vector to another single precision complex vector.

Parameters

- **handle** (*int*) – CUBLAS context.
- **n** (*int*) – Number of elements in input vectors.
- **x** (*ctypes.c_void_p*) – Pointer to single precision complex input vector.
- **incx** (*int*) – Storage spacing between elements of *x*.
- **y** (*ctypes.c_void_p*) – Pointer to single precision complex output vector.
- **incy** (*int*) – Storage spacing between elements of *y*.

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> x = (np.random.rand(5)+np.random.rand(5)).astype(np.complex64)
>>> x_gpu = gpuarray.to_gpu(x)
>>> y_gpu = gpuarray.zeros_like(x_gpu)
>>> h = cublasCreate()
>>> cublasCcopy(h, x_gpu.size, x_gpu.gpudata, 1, y_gpu.gpudata, 1)
>>> cublasDestroy(h)
>>> np.allclose(y_gpu.get(), x_gpu.get())
True
```

Notes

Both *x* and *y* must contain *n* elements.

References

cublas<t>copy

skcuda.cublas.cublasCdotc

skcuda.cublas.**cublasCdotc**(*handle*, *n*, *x*, *incx*, *y*, *incy*)

Vector dot product (single precision complex)

Computes the dot product of two single precision complex vectors. cublasCdotc and cublasZdotc use the conjugate of the first vector when computing the dot product.

Parameters

- **handle** (*int*) – CUBLAS context.
- **n** (*int*) – Number of elements in input vectors.
- **x** (*ctypes.c_void_p*) – Pointer to single precision complex input vector.
- **incx** (*int*) – Storage spacing between elements of *x*.
- **y** (*ctypes.c_void_p*) – Pointer to single precision complex input/output vector.
- **incy** (*int*) – Storage spacing between elements of *y*.

Returns **d** – Dot product of *x* and *y*.

Return type np.complex64

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> x = (np.random.rand(5)+1j*np.random.rand(5)).astype(np.complex64)
>>> y = (np.random.rand(5)+1j*np.random.rand(5)).astype(np.complex64)
>>> x_gpu = gpuarray.to_gpu(x)
>>> y_gpu = gpuarray.to_gpu(y)
>>> h = cublasCreate()
>>> d = cublasCdotc(h, x_gpu.size, x_gpu.gpudata, 1, y_gpu.gpudata, 1)
>>> cublasDestroy(h)
>>> np.allclose(d, np.dot(np.conj(x), y))
True
```

Notes

Both *x* and *y* must contain *n* elements.

References

cublas<t>dot

skcuda.cublas.cublasCdotu

skcuda.cublas.**cublasCdotu**(*handle*, *n*, *x*, *incx*, *y*, *incy*)

Vector dot product (single precision complex)

Computes the dot product of two single precision complex vectors. cublasCdotc and cublasZdotc use the conjugate of the first vector when computing the dot product.

Parameters

- **handle** (*int*) – CUBLAS context.
- **n** (*int*) – Number of elements in input vectors.
- **x** (*ctypes.c_void_p*) – Pointer to single precision complex input vector.
- **incx** (*int*) – Storage spacing between elements of *x*.
- **y** (*ctypes.c_void_p*) – Pointer to single precision complex input/output vector.
- **incy** (*int*) – Storage spacing between elements of *y*.

Returns **d** – Dot product of *x* and *y*.

Return type np.complex64

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> x = (np.random.rand(5)+1j*np.random.rand(5)).astype(np.complex64)
>>> y = (np.random.rand(5)+1j*np.random.rand(5)).astype(np.complex64)
>>> x_gpu = gpuarray.to_gpu(x)
>>> y_gpu = gpuarray.to_gpu(y)
>>> h = cublasCreate()
>>> d = cublasCdotu(h, x_gpu.size, x_gpu.gpudata, 1, y_gpu.gpudata, 1)
>>> cublasDestroy(h)
>>> np.allclose(d, np.dot(x, y))
True
```

Notes

Both *x* and *y* must contain *n* elements.

References

cublas<t>dot

skcuda.cublas.cublasCrot

skcuda.cublas.**cublasCrot**(*handle*, *n*, *x*, *incx*, *y*, *incy*, *c*, *s*)

Apply a complex rotation to complex vectors (single precision)

Multiplies the single precision matrix *[[c, s], [-s.conj(), c]]* with the 2 x *n* single precision matrix *[[x.T], [y.T]]*.

Parameters

- **handle** (*int*) – CUBLAS context.
- **n** (*int*) – Number of elements in input vectors.
- **x** (*ctypes.c_void_p*) – Pointer to single precision complex input/output vector.
- **incx** (*int*) – Storage spacing between elements of *x*.
- **y** (*ctypes.c_void_p*) – Pointer to single precision complex input/output vector.
- **incy** (*int*) – Storage spacing between elements of *y*.
- **c** (*numpy.float32*) – Element of rotation matrix.
- **s** (*numpy.complex64*) – Element of rotation matrix.

Notes

Both *x* and *y* must contain *n* elements.

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> s = np.complex64(np.random.rand()+1j*np.random.rand()); c = np.float32(np.
↪random.rand());
>>> x = (np.random.rand(5)+1j*np.random.rand(5)).astype(np.complex64)
>>> y = (np.random.rand(5)+1j*np.random.rand(5)).astype(np.complex64)
>>> x_gpu = gpuarray.to_gpu(x)
>>> y_gpu = gpuarray.to_gpu(y)
>>> h = cublasCreate()
>>> cublasCrot(h, x.size, x_gpu.gpudata, 1, y_gpu.gpudata, 1, c, s)
>>> cublasDestroy(h)
>>> np.allclose(x_gpu.get(), c*x+s*y)
True
>>> np.allclose(y_gpu.get(), -s.conj()*x+c*y)
True
```

References

cublas<t>rot

skcuda.cublas.cublasCrotg

skcuda.cublas.**cublasCrotg**(*handle*, *a*, *b*)

Construct a single precision complex Givens rotation matrix.

Constructs the single precision complex Givens rotation matrix *G = [[c, s], [-s.conj(), c]]* such that *dot(G, [[a], [b]] == [[r], [0]]*, where *c**2+s**2 == 1* and *r == a**2+b**2* for real numbers and *c**2+(conj(s)*s) == 1* and *r == (a/abs(a))*sqrt(abs(a)**2+abs(b)**2)* for *a != 0* and *r == b* for *a == 0*.

Parameters

- **handle** (*int*) – CUBLAS context.
- **b** (*a,*) – Entries of vector whose second entry should be zeroed out by the rotation.

Returns

- **r** (*numpy.complex64*) – Defined above.
- **c** (*numpy.float32*) – Cosine component of rotation matrix.
- **s** (*numpy.complex64*) – Sine component of rotation matrix.

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> a = np.complex64(np.random.rand()+1j*np.random.rand())
>>> b = np.complex64(np.random.rand()+1j*np.random.rand())
>>> h = cublasCreate()
>>> r, c, s = cublasCrotg(h, a, b)
>>> cublasDestroy(h)
>>> np.allclose(np.dot(np.array([[c, s], [-np.conj(s), c]]), np.array([[a],␣
↪[b]])), np.array([[r], [0.0]]), atol=1e-6)
True
```

References

cublas<t>rotg

skcuda.cublas.cublasCscal

skcuda.cublas.**cublasCscal**(*handle*, *n*, *alpha*, *x*, *incx*)

Scale a single precision complex vector by a single precision complex scalar.

Replaces a single precision complex vector *x* with *alpha * x*.

Parameters

- **handle** (*int*) – CUBLAS context.
- **n** (*int*) – Number of elements in input vectors.
- **alpha** (*numpy.complex64*) – Scalar multiplier.
- **x** (*ctypes.c_void_p*) – Pointer to single precision complex input/output vector.

- **incx** (*int*) – Storage spacing between elements of *x*.

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> x = (np.random.rand(5)+1j*np.random.rand(5)).astype(np.complex64)
>>> x_gpu = gpuarray.to_gpu(x)
>>> alpha = np.complex64(np.random.rand()+1j*np.random.rand())
>>> h = cublasCreate()
>>> cublasCscal(h, x.size, alpha, x_gpu.gpudata, 1)
>>> cublasDestroy(h)
>>> np.allclose(x_gpu.get(), alpha*x)
True
```

References

cublas<t>scal

skcuda.cublas.cublasCsrot

skcuda.cublas.**cublasCsrot**(*handle*, *n*, *x*, *incx*, *y*, *incy*, *c*, *s*)

Apply a complex rotation to complex vectors (single precision)

Multiplies the single precision matrix *[[c, s], [-s.conj(), c]]* with the 2 x *n* single precision matrix *[[x.T], [y.T]]*.

Parameters

- **handle** (*int*) – CUBLAS context.
- **n** (*int*) – Number of elements in input vectors.
- **x** (*ctypes.c_void_p*) – Pointer to single precision complex input/output vector.
- **incx** (*int*) – Storage spacing between elements of *x*.
- **y** (*ctypes.c_void_p*) – Pointer to single precision complex input/output vector.
- **incy** (*int*) – Storage spacing between elements of *y*.
- **c** (*numpy.float32*) – Element of rotation matrix.
- **s** (*numpy.float32*) – Element of rotation matrix.

Notes

Both *x* and *y* must contain *n* elements.

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> s = np.float32(np.random.rand()); c = np.float32(np.random.rand());
>>> x = (np.random.rand(5)+1j*np.random.rand(5)).astype(np.complex64)
>>> y = (np.random.rand(5)+1j*np.random.rand(5)).astype(np.complex64)
>>> x_gpu = gpuarray.to_gpu(x)
>>> y_gpu = gpuarray.to_gpu(y)
>>> h = cublasCreate()
>>> cublasCsrot(h, x.size, x_gpu.gpudata, 1, y_gpu.gpudata, 1, c, s)
>>> cublasDestroy(h)
>>> np.allclose(x_gpu.get(), c*x+s*y)
True
>>> np.allclose(y_gpu.get(), -s.conj()*x+c*y)
True
```

References

cublas<t>rot

skcuda.cublas.cublasCsscal

skcuda.cublas.**cublasCsscal**(*handle*, *n*, *alpha*, *x*, *incx*)

Scale a single precision complex vector by a single precision real scalar.

Replaces a single precision complex vector *x* with *alpha * x*.

Parameters

- **handle** (*int*) – CUBLAS context.
- **n** (*int*) – Number of elements in input vectors.
- **alpha** (*numpy.float32*) – Scalar multiplier.
- **x** (*ctypes.c_void_p*) – Pointer to single precision complex input/output vector.
- **incx** (*int*) – Storage spacing between elements of *x*.

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> x = (np.random.rand(5)+1j*np.random.rand(5)).astype(np.complex64)
>>> x_gpu = gpuarray.to_gpu(x)
>>> alpha = np.float32(np.random.rand())
>>> h = cublasCreate()
>>> cublasCsscal(h, x.size, alpha, x_gpu.gpudata, 1)
>>> cublasDestroy(h)
>>> np.allclose(x_gpu.get(), alpha*x)
True
```

References

cublas<t>scal

skcuda.cublas.cublasCswap

skcuda.cublas.**cublasCswap**(*handle*, *n*, *x*, *incx*, *y*, *incy*)

Swap single precision complex vectors.

Swaps the contents of one single precision complex vector with those of another single precision complex vector.

Parameters

- **handle** (*int*) – CUBLAS context.
- **n** (*int*) – Number of elements in input vectors.
- **x** (*ctypes.c_void_p*) – Pointer to single precision complex input/output vector.
- **incx** (*int*) – Storage spacing between elements of *x*.
- **y** (*ctypes.c_void_p*) – Pointer to single precision complex input/output vector.
- **incy** (*int*) – Storage spacing between elements of *y*.

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> x = (np.random.rand(5)+1j*np.random.rand(5)).astype(np.complex64)
>>> y = (np.random.rand(5)+1j*np.random.rand(5)).astype(np.complex64)
>>> x_gpu = gpuarray.to_gpu(x)
>>> y_gpu = gpuarray.to_gpu(y)
>>> h = cublasCreate()
>>> cublasCswap(h, x.size, x_gpu.gpudata, 1, y_gpu.gpudata, 1)
>>> cublasDestroy(h)
>>> np.allclose(x_gpu.get(), y)
True
>>> np.allclose(y_gpu.get(), x)
True
```

Notes

Both *x* and *y* must contain *n* elements.

References

cublas<t>swap

skcuda.cublas.cublasIcamax

skcuda.cublas.**cublasIcamax**(*handle*, *n*, *x*, *incx*)

Index of maximum magnitude element.

Finds the smallest index of the maximum magnitude element of a single precision complex vector.

Note: for complex arguments x, the "magnitude" is defined as *abs(x.real) + abs(x.imag)*, *not* as *abs(x)*.

Parameters

- **handle** (*int*) – CUBLAS context.
- **n** (*int*) – Number of elements in input vector.
- **x** (*ctypes.c_void_p*) – Pointer to single precision complex input vector.
- **incx** (*int*) – Storage spacing between elements of *x*.

Returns **idx** – Index of maximum magnitude element.

Return type int

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> x = (np.random.rand(5)+1j*np.random.rand(5)).astype(np.complex64)
>>> x_gpu = gpuarray.to_gpu(x)
>>> h = cublasCreate()
>>> m = cublasIcamax(h, x_gpu.size, x_gpu.gpudata, 1)
>>> cublasDestroy(h)
>>> np.allclose(m, np.argmax(abs(x.real) + abs(x.imag)))
True
```

Notes

This function returns a 0-based index.

References

cublasI<t>amax

skcuda.cublas.cublasIcamin

skcuda.cublas.**cublasIcamin**(*handle*, *n*, *x*, *incx*)

Index of minimum magnitude element (single precision complex).

Finds the smallest index of the minimum magnitude element of a single precision complex vector.

Note: for complex arguments x, the "magnitude" is defined as *abs(x.real) + abs(x.imag)*, *not* as *abs(x)*.

Parameters

- **handle** (*int*) – CUBLAS context.

- **n** (*int*) – Number of elements in input vector.
- **x** (*ctypes.c_void_p*) – Pointer to single precision complex input vector.
- **incx** (*int*) – Storage spacing between elements of *x*.

Returns **idx** – Index of minimum magnitude element.

Return type int

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> x = (np.random.rand(5)+1j*np.random.rand(5)).astype(np.complex64)
>>> x_gpu = gpuarray.to_gpu(x)
>>> h = cublasCreate()
>>> m = cublasIcamin(h, x_gpu.size, x_gpu.gpudata, 1)
>>> cublasDestroy(h)
>>> np.allclose(m, np.argmin(abs(x.real) + abs(x.imag)))
True
```

Notes

This function returns a 0-based index.

References

cublasI<t>amin

skcuda.cublas.cublasScasum

skcuda.cublas.**cublasScasum**(*handle*, *n*, *x*, *incx*)

Sum of absolute values of single precision complex vector.

Computes the sum of the absolute values of the elements of a single precision complex vector.

Note: if the vector is complex, then this computes the sum *sum(abs(x.real)) + sum(abs(x.imag))*

Parameters

- **handle** (*int*) – CUBLAS context.
- **n** (*int*) – Number of elements in input vector.
- **x** (*ctypes.c_void_p*) – Pointer to single precision complex input vector.
- **incx** (*int*) – Storage spacing between elements of *x*.

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> x = (np.random.rand(5)+1j*np.random.rand(5)).astype(np.complex64)
>>> x_gpu = gpuarray.to_gpu(x)
>>> h = cublasCreate()
>>> s = cublasScasum(h, x_gpu.size, x_gpu.gpudata, 1)
>>> cublasDestroy(h)
>>> np.allclose(s, abs(x.real).sum() + abs(x.imag).sum())
True
```

Returns **s** – Sum of absolute values.

Return type numpy.float32

References

cublas<t>sum

skcuda.cublas.cublasScnrm2

skcuda.cublas.**cublasScnrm2**(*handle*, *n*, *x*, *incx*)

Euclidean norm (2-norm) of real vector.

Computes the Euclidean norm of a single precision complex vector.

Parameters

- **handle** (*int*) – CUBLAS context.
- **n** (*int*) – Number of elements in input vectors.
- **x** (*ctypes.c_void_p*) – Pointer to single precision complex input vector.
- **incx** (*int*) – Storage spacing between elements of *x*.

Returns **nrm** – Euclidean norm of *x*.

Return type numpy.complex64

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> x = (np.random.rand(5)+1j*np.random.rand(5)).astype(np.complex64)
>>> x_gpu = gpuarray.to_gpu(x)
>>> h = cublasCreate()
>>> nrm = cublasScnrm2(h, x.size, x_gpu.gpudata, 1)
>>> cublasDestroy(h)
>>> np.allclose(nrm, np.linalg.norm(x))
True
```

References

cublas<t>nrm2

Double Precision BLAS1 Routines

cublasIdamax	Index of maximum magnitude element.
cublasIdamin	Index of minimum magnitude element (double precision real).
cublasDasum	Sum of absolute values of double precision real vector.
cublasDaxpy	Vector addition (double precision real).
cublasDcopy	Vector copy (double precision real)
cublasDdot	Vector dot product (double precision real)
cublasDnrm2	Euclidean norm (2-norm) of real vector.
cublasDrot	Apply a real rotation to real vectors (double precision)
cublasDrotg	Construct a double precision real Givens rotation matrix.
cublasDrotm	Apply a double precision real modified Givens rotation.
cublasDrotmg	Construct a double precision real modified Givens rotation matrix.
cublasDscal	Scale a double precision real vector by a double precision real scalar.
cublasDswap	Swap double precision real vectors.
cublasDzasum	Sum of absolute values of double precision complex vector.
cublasDznrm2	Euclidean norm (2-norm) of real vector.
cublasIzamax	Index of maximum magnitude element.
cublasIzamin	Index of minimum magnitude element (double precision complex).
cublasZaxpy	Vector addition (double precision complex).
cublasZcopy	Vector copy (double precision complex)
cublasZdotc	Vector dot product (double precision complex)
cublasZdotu	Vector dot product (double precision complex)
cublasZdrot	Apply a complex rotation to complex vectors (double precision)
cublasZdscal	Scale a double precision complex vector by a double precision real scalar.
cublasZrot	Apply a complex rotation to complex vectors (double precision)
cublasZrotg	Construct a double precision complex Givens rotation matrix.
cublasZscal	Scale a double precision complex vector by a double precision complex scalar.
cublasZswap	Swap double precision complex vectors.

skcuda.cublas.cublasIdamax

skcuda.cublas.**cublasIdamax**(*handle*, *n*, *x*, *incx*)

Index of maximum magnitude element.

Finds the smallest index of the maximum magnitude element of a double precision real vector.

Note: for complex arguments x, the "magnitude" is defined as *abs(x.real) + abs(x.imag)*, *not* as *abs(x)*.

Parameters

- **handle** (*int*) – CUBLAS context.
- **n** (*int*) – Number of elements in input vector.
- **x** (*ctypes.c_void_p*) – Pointer to double precision real input vector.
- **incx** (*int*) – Storage spacing between elements of *x*.

Returns **idx** – Index of maximum magnitude element.

Return type int

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> x = np.random.rand(5).astype(np.float64)
>>> x_gpu = gpuarray.to_gpu(x)
>>> h = cublasCreate()
>>> m = cublasIdamax(h, x_gpu.size, x_gpu.gpudata, 1)
>>> cublasDestroy(h)
>>> np.allclose(m, np.argmax(abs(x.real) + abs(x.imag)))
True
```

Notes

This function returns a 0-based index.

References

cublasI<t>amax

skcuda.cublas.cublasIdamin

skcuda.cublas.**cublasIdamin**(*handle*, *n*, *x*, *incx*)

Index of minimum magnitude element (double precision real).

Finds the smallest index of the minimum magnitude element of a double precision real vector.

Note: for complex arguments x, the "magnitude" is defined as *abs(x.real) + abs(x.imag)*, *not* as *abs(x)*.

Parameters

- **handle** (*int*) – CUBLAS context.

- **n** (*int*) – Number of elements in input vector.
- **x** (*ctypes.c_void_p*) – Pointer to double precision real input vector.
- **incx** (*int*) – Storage spacing between elements of *x*.

Returns **idx** – Index of minimum magnitude element.

Return type int

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> x = np.random.rand(5).astype(np.float64)
>>> x_gpu = gpuarray.to_gpu(x)
>>> h = cublasCreate()
>>> m = cublasIdamin(h, x_gpu.size, x_gpu.gpudata, 1)
>>> cublasDestroy(h)
>>> np.allclose(m, np.argmin(abs(x.real) + abs(x.imag)))
True
```

Notes

This function returns a 0-based index.

References

cublasI<t>amin

skcuda.cublas.cublasDasum

skcuda.cublas.**cublasDasum**(*handle*, *n*, *x*, *incx*)

Sum of absolute values of double precision real vector.

Computes the sum of the absolute values of the elements of a double precision real vector.

Note: if the vector is complex, then this computes the sum *sum(abs(x.real)) + sum(abs(x.imag))*

Parameters

- **handle** (*int*) – CUBLAS context.
- **n** (*int*) – Number of elements in input vector.
- **x** (*ctypes.c_void_p*) – Pointer to double precision real input vector.
- **incx** (*int*) – Storage spacing between elements of *x*.

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> x = np.random.rand(5).astype(np.float64)
>>> x_gpu = gpuarray.to_gpu(x)
>>> h = cublasCreate()
>>> s = cublasDasum(h, x_gpu.size, x_gpu.gpudata, 1)
>>> cublasDestroy(h)
>>> np.allclose(s, abs(x.real).sum() + abs(x.imag).sum())
True
```

Returns **s** – Sum of absolute values.

Return type numpy.float64

References

cublas<t>sum

skcuda.cublas.cublasDaxpy

skcuda.cublas.**cublasDaxpy**(*handle*, *n*, *alpha*, *x*, *incx*, *y*, *incy*)

Vector addition (double precision real).

Computes the sum of a double precision real vector scaled by a double precision real scalar and another double precision real vector.

Parameters

- **handle** (*int*) – CUBLAS context.
- **n** (*int*) – Number of elements in input vectors.
- **alpha** (*numpy.float64*) – Scalar.
- **x** (*ctypes.c_void_p*) – Pointer to single precision input vector.
- **incx** (*int*) – Storage spacing between elements of *x*.
- **y** (*ctypes.c_void_p*) – Pointer to single precision input/output vector.
- **incy** (*int*) – Storage spacing between elements of *y*.

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> alpha = np.float64(np.random.rand())
>>> x = np.random.rand(5).astype(np.float64)
>>> y = np.random.rand(5).astype(np.float64)
>>> x_gpu = gpuarray.to_gpu(x)
>>> y_gpu = gpuarray.to_gpu(y)
>>> h = cublasCreate()
>>> cublasDaxpy(h, x_gpu.size, alpha, x_gpu.gpudata, 1, y_gpu.gpudata, 1)
```

(continues on next page)

(continued from previous page)

```
>>> cublasDestroy(h)
>>> np.allclose(y_gpu.get(), alpha*x+y)
True
```

Notes

Both *x* and *y* must contain *n* elements.

References

cublas<t>axpy

skcuda.cublas.cublasDcopy

skcuda.cublas.**cublasDcopy**(*handle*, *n*, *x*, *incx*, *y*, *incy*)

Vector copy (double precision real)

Copies a double precision real vector to another double precision real vector.

Parameters

- **handle** (*int*) – CUBLAS context.
- **n** (*int*) – Number of elements in input vectors.
- **x** (*ctypes.c_void_p*) – Pointer to double precision real input vector.
- **incx** (*int*) – Storage spacing between elements of *x*.
- **y** (*ctypes.c_void_p*) – Pointer to double precision real output vector.
- **incy** (*int*) – Storage spacing between elements of *y*.

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> x = np.random.rand(5).astype(np.float64)
>>> x_gpu = gpuarray.to_gpu(x)
>>> y_gpu = gpuarray.zeros_like(x_gpu)
>>> h = cublasCreate()
>>> cublasDcopy(h, x_gpu.size, x_gpu.gpudata, 1, y_gpu.gpudata, 1)
>>> cublasDestroy(h)
>>> np.allclose(y_gpu.get(), x_gpu.get())
True
```

Notes

Both *x* and *y* must contain *n* elements.

References

cublas<t>copy

skcuda.cublas.cublasDdot

skcuda.cublas.**cublasDdot**(*handle*, *n*, *x*, *incx*, *y*, *incy*)

Vector dot product (double precision real)

Computes the dot product of two double precision real vectors. cublasCdotc and cublasZdotc use the conjugate of the first vector when computing the dot product.

Parameters

- **handle** (*int*) – CUBLAS context.
- **n** (*int*) – Number of elements in input vectors.
- **x** (*ctypes.c_void_p*) – Pointer to double precision real input vector.
- **incx** (*int*) – Storage spacing between elements of *x*.
- **y** (*ctypes.c_void_p*) – Pointer to double precision real input/output vector.
- **incy** (*int*) – Storage spacing between elements of *y*.

Returns **d** – Dot product of *x* and *y*.

Return type np.float64

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> x = np.float64(np.random.rand(5))
>>> y = np.float64(np.random.rand(5))
>>> x_gpu = gpuarray.to_gpu(x)
>>> y_gpu = gpuarray.to_gpu(y)
>>> h = cublasCreate()
>>> d = cublasDdot(h, x_gpu.size, x_gpu.gpudata, 1, y_gpu.gpudata, 1)
>>> cublasDestroy(h)
>>> np.allclose(d, np.dot(x, y))
True
```

Notes

Both *x* and *y* must contain *n* elements.

References

cublas<t>dot

skcuda.cublas.cublasDnrm2

skcuda.cublas.**cublasDnrm2**(*handle*, *n*, *x*, *incx*)

Euclidean norm (2-norm) of real vector.

Computes the Euclidean norm of a double precision real vector.

Parameters

- **handle** (*int*) – CUBLAS context.
- **n** (*int*) – Number of elements in input vectors.
- **x** (*ctypes.c_void_p*) – Pointer to double precision real input vector.
- **incx** (*int*) – Storage spacing between elements of *x*.

Returns **nrm** – Euclidean norm of *x*.

Return type numpy.float64

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> x = np.float64(np.random.rand(5))
>>> x_gpu = gpuarray.to_gpu(x)
>>> h = cublasCreate()
>>> nrm = cublasDnrm2(h, x.size, x_gpu.gpudata, 1)
>>> cublasDestroy(h)
>>> np.allclose(nrm, np.linalg.norm(x))
True
```

References

cublas<t>nrm2

skcuda.cublas.cublasDrot

skcuda.cublas.**cublasDrot**(*handle*, *n*, *x*, *incx*, *y*, *incy*, *c*, *s*)

Apply a real rotation to real vectors (double precision)

Multiplies the double precision matrix *[[c, s], [-s.conj(), c]]* with the 2 x *n* double precision matrix *[[x.T], [y.T]]*.

Parameters

- **handle** (*int*) – CUBLAS context.
- **n** (*int*) – Number of elements in input vectors.
- **x** (*ctypes.c_void_p*) – Pointer to double precision real input/output vector.
- **incx** (*int*) – Storage spacing between elements of *x*.
- **y** (*ctypes.c_void_p*) – Pointer to double precision real input/output vector.
- **incy** (*int*) – Storage spacing between elements of *y*.

- **c** (*numpy.float64*) – Element of rotation matrix.
- **s** (*numpy.float64*) – Element of rotation matrix.

Notes

Both *x* and *y* must contain *n* elements.

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> s = np.float64(np.random.rand()); c = np.float64(np.random.rand());
>>> x = np.random.rand(5).astype(np.float64)
>>> y = np.random.rand(5).astype(np.float64)
>>> x_gpu = gpuarray.to_gpu(x)
>>> y_gpu = gpuarray.to_gpu(y)
>>> h = cublasCreate()
>>> cublasDrot(h, x.size, x_gpu.gpudata, 1, y_gpu.gpudata, 1, c, s)
>>> cublasDestroy(h)
>>> np.allclose(x_gpu.get(), c*x+s*y)
True
>>> np.allclose(y_gpu.get(), -s.conj()*x+c*y)
True
```

References

cublas<t>rot

skcuda.cublas.cublasDrotg

skcuda.cublas.**cublasDrotg**(*handle*, *a*, *b*)

Construct a double precision real Givens rotation matrix.

Constructs the double precision real Givens rotation matrix *G = [[c, s], [-s.conj(), c]]* such that *dot(G, [[a], [b]] == [[r], [0]]*, where *c**2+s**2 == 1* and *r == a**2+b**2* for real numbers and *c**2+(conj(s)*s) == 1* and *r == (a/abs(a))*sqrt(abs(a)**2+abs(b)**2)* for *a != 0* and *r == b* for *a == 0*.

Parameters

- **handle** (*int*) – CUBLAS context.
- **b** (*a,*) – Entries of vector whose second entry should be zeroed out by the rotation.

Returns

- **r** (*numpy.float64*) – Defined above.
- **c** (*numpy.float64*) – Cosine component of rotation matrix.
- **s** (*numpy.float64*) – Sine component of rotation matrix.

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> a = np.float64(np.random.rand())
>>> b = np.float64(np.random.rand())
>>> h = cublasCreate()
>>> r, c, s = cublasDrotg(h, a, b)
>>> cublasDestroy(h)
>>> np.allclose(np.dot(np.array([[c, s], [-np.conj(s), c]]), np.array([[a],␣
↪[b]])), np.array([[r], [0.0]]), atol=1e-6)
True
```

References

cublas<t>rotg

skcuda.cublas.cublasDrotm

skcuda.cublas.**cublasDrotm**(*handle*, *n*, *x*, *incx*, *y*, *incy*, *sparam*)

Apply a double precision real modified Givens rotation.

Applies the double precision real modified Givens rotation matrix *h* to the 2 x *n* matrix *[[x.T], [y.T]]*.

Parameters

- **handle** (*int*) – CUBLAS context.
- **n** (*int*) – Number of elements in input vectors.
- **x** (*ctypes.c_void_p*) – Pointer to double precision real input/output vector.
- **incx** (*int*) – Storage spacing between elements of *x*.
- **y** (*ctypes.c_void_p*) – Pointer to double precision real input/output vector.
- **incy** (*int*) – Storage spacing between elements of *y*.
- **sparam** (*numpy.ndarray*) – sparam[0] contains the *flag* described below; sparam[1:5] contains the values *[h00, h10, h01, h11]* that determine the rotation matrix *h*.

Notes

The rotation matrix may assume the following values:

for *flag* == -1.0, *h* == *[[h00, h01], [h10, h11]]* for *flag* == 0.0, *h* == *[[1.0, h01], [h10, 1.0]]* for *flag* == 1.0, *h* == *[[h00, 1.0], [-1.0, h11]]* for *flag* == -2.0, *h* == *[[1.0, 0.0], [0.0, 1.0]]*

Both *x* and *y* must contain *n* elements.

References

cublas<t>srotm

skcuda.cublas.cublasDrotmg

skcuda.cublas.**cublasDrotmg**(*handle*, *d1*, *d2*, *x1*, *y1*)

Construct a double precision real modified Givens rotation matrix.

Constructs the double precision real modified Givens rotation matrix *h = [[h11, h12], [h21, h22]]* that zeros out the second entry of the vector *[[sqrt(d1)*x1], [sqrt(d2)*x2]]*.

Parameters

- **handle** (*int*) – CUBLAS context.
- **d1** (*numpy.float64*) – double precision real value.
- **d2** (*numpy.float64*) – double precision real value.
- **x1** (*numpy.float64*) – double precision real value.
- **x2** (*numpy.float64*) – double precision real value.

Returns **sparam** – sparam[0] contains the *flag* described below; sparam[1:5] contains the values *[h00, h10, h01, h11]* that determine the rotation matrix *h*.

Return type numpy.ndarray

Notes

The rotation matrix may assume the following values:

for *flag* == -1.0, *h* == *[[h00, h01], [h10, h11]]* for *flag* == 0.0, *h* == *[[1.0, h01], [h10, 1.0]]* for *flag* == 1.0, *h* == *[[h00, 1.0], [-1.0, h11]]* for *flag* == -2.0, *h* == *[[1.0, 0.0], [0.0, 1.0]]*

References

cublas<t>rotmg

skcuda.cublas.cublasDscal

skcuda.cublas.**cublasDscal**(*handle*, *n*, *alpha*, *x*, *incx*)

Scale a double precision real vector by a double precision real scalar.

Replaces a double precision real vector *x* with *alpha * x*.

Parameters

- **handle** (*int*) – CUBLAS context.
- **n** (*int*) – Number of elements in input vectors.
- **alpha** (*numpy.float64*) – Scalar multiplier.
- **x** (*ctypes.c_void_p*) – Pointer to double precision real input/output vector.
- **incx** (*int*) – Storage spacing between elements of *x*.

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> x = np.random.rand(5).astype(np.float64)
>>> x_gpu = gpuarray.to_gpu(x)
>>> alpha = np.float64(np.random.rand())
>>> h = cublasCreate()
>>> cublasDscal(h, x.size, alpha, x_gpu.gpudata, 1)
>>> cublasDestroy(h)
>>> np.allclose(x_gpu.get(), alpha*x)
True
```

References

cublas<t>scal

skcuda.cublas.cublasDswap

skcuda.cublas.**cublasDswap**(*handle*, *n*, *x*, *incx*, *y*, *incy*)

Swap double precision real vectors.

Swaps the contents of one double precision real vector with those of another double precision real vector.

Parameters

- **handle** (*int*) – CUBLAS context.
- **n** (*int*) – Number of elements in input vectors.
- **x** (*ctypes.c_void_p*) – Pointer to double precision real input/output vector.
- **incx** (*int*) – Storage spacing between elements of *x*.
- **y** (*ctypes.c_void_p*) – Pointer to double precision real input/output vector.
- **incy** (*int*) – Storage spacing between elements of *y*.

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> x = np.random.rand(5).astype(np.float64)
>>> y = np.random.rand(5).astype(np.float64)
>>> x_gpu = gpuarray.to_gpu(x)
>>> y_gpu = gpuarray.to_gpu(y)
>>> h = cublasCreate()
>>> cublasDswap(h, x.size, x_gpu.gpudata, 1, y_gpu.gpudata, 1)
>>> cublasDestroy(h)
>>> np.allclose(x_gpu.get(), y)
True
```

(continues on next page)

(continued from previous page)

```
>>> np.allclose(y_gpu.get(), x)
True
```

Notes

Both *x* and *y* must contain *n* elements.

References

cublas<t>swap

skcuda.cublas.cublasDzasum

skcuda.cublas.**cublasDzasum**(*handle*, *n*, *x*, *incx*)

Sum of absolute values of double precision complex vector.

Computes the sum of the absolute values of the elements of a double precision complex vector.

Note: if the vector is complex, then this computes the sum *sum(abs(x.real)) + sum(abs(x.imag))*

Parameters

- **handle** (*int*) – CUBLAS context.
- **n** (*int*) – Number of elements in input vector.
- **x** (*ctypes.c_void_p*) – Pointer to double precision complex input vector.
- **incx** (*int*) – Storage spacing between elements of *x*.

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> x = (np.random.rand(5)+1j*np.random.rand(5)).astype(np.complex128)
>>> x_gpu = gpuarray.to_gpu(x)
>>> h = cublasCreate()
>>> s = cublasDzasum(h, x_gpu.size, x_gpu.gpudata, 1)
>>> cublasDestroy(h)
>>> np.allclose(s, abs(x.real).sum() + abs(x.imag).sum())
True
```

Returns **s** – Sum of absolute values.

Return type numpy.float64

References

cublas<t>sum

skcuda.cublas.cublasDznrm2

skcuda.cublas.**cublasDznrm2**(*handle*, *n*, *x*, *incx*)

Euclidean norm (2-norm) of real vector.

Computes the Euclidean norm of a double precision complex vector.

Parameters

- **handle** (*int*) – CUBLAS context.
- **n** (*int*) – Number of elements in input vectors.
- **x** (*ctypes.c_void_p*) – Pointer to double precision complex input vector.
- **incx** (*int*) – Storage spacing between elements of *x*.

Returns **nrm** – Euclidean norm of *x*.

Return type numpy.complex128

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> x = (np.random.rand(5)+1j*np.random.rand(5)).astype(np.complex128)
>>> x_gpu = gpuarray.to_gpu(x)
>>> h = cublasCreate()
>>> nrm = cublasDznrm2(h, x.size, x_gpu.gpudata, 1)
>>> cublasDestroy(h)
>>> np.allclose(nrm, np.linalg.norm(x))
True
```

References

cublas<t>nrm2

skcuda.cublas.cublasIzamax

skcuda.cublas.**cublasIzamax**(*handle*, *n*, *x*, *incx*)

Index of maximum magnitude element.

Finds the smallest index of the maximum magnitude element of a double precision complex vector.

Note: for complex arguments x, the "magnitude" is defined as *abs(x.real) + abs(x.imag)*, *not* as *abs(x)*.

Parameters

- **handle** (*int*) – CUBLAS context.
- **n** (*int*) – Number of elements in input vector.
- **x** (*ctypes.c_void_p*) – Pointer to double precision complex input vector.
- **incx** (*int*) – Storage spacing between elements of *x*.

Returns **idx** – Index of maximum magnitude element.

Return type int

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> x = (np.random.rand(5)+1j*np.random.rand(5)).astype(np.complex128)
>>> x_gpu = gpuarray.to_gpu(x)
>>> h = cublasCreate()
>>> m = cublasIzamax(h, x_gpu.size, x_gpu.gpudata, 1)
>>> cublasDestroy(h)
>>> np.allclose(m, np.argmax(abs(x.real) + abs(x.imag)))
True
```

Notes

This function returns a 0-based index.

References

cublasI<t>amax

skcuda.cublas.cublasIzamin

skcuda.cublas.**cublasIzamin**(*handle*, *n*, *x*, *incx*)

Index of minimum magnitude element (double precision complex).

Finds the smallest index of the minimum magnitude element of a double precision complex vector.

Note: for complex arguments x, the "magnitude" is defined as *abs(x.real) + abs(x.imag)*, *not* as *abs(x)*.

Parameters

- **handle** (*int*) – CUBLAS context.
- **n** (*int*) – Number of elements in input vector.
- **x** (*ctypes.c_void_p*) – Pointer to double precision complex input vector.
- **incx** (*int*) – Storage spacing between elements of *x*.

Returns **idx** – Index of minimum magnitude element.

Return type int

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> x = (np.random.rand(5)+1j*np.random.rand(5)).astype(np.complex128)
>>> x_gpu = gpuarray.to_gpu(x)
```

(continues on next page)

(continued from previous page)

```
>>> h = cublasCreate()
>>> m = cublasIzamin(h, x_gpu.size, x_gpu.gpudata, 1)
>>> cublasDestroy(h)
>>> np.allclose(m, np.argmin(abs(x.real) + abs(x.imag)))
True
```

Notes

This function returns a 0-based index.

References

cublasI<t>amin

skcuda.cublas.cublasZaxpy

skcuda.cublas.**cublasZaxpy**(*handle*, *n*, *alpha*, *x*, *incx*, *y*, *incy*)

Vector addition (double precision complex).

Computes the sum of a double precision complex vector scaled by a double precision complex scalar and another double precision complex vector.

Parameters

- **handle** (*int*) – CUBLAS context.
- **n** (*int*) – Number of elements in input vectors.
- **alpha** (*numpy.complex128*) – Scalar.
- **x** (*ctypes.c_void_p*) – Pointer to single precision input vector.
- **incx** (*int*) – Storage spacing between elements of *x*.
- **y** (*ctypes.c_void_p*) – Pointer to single precision input/output vector.
- **incy** (*int*) – Storage spacing between elements of *y*.

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> alpha = np.complex128(np.random.rand()+1j*np.random.rand())
>>> x = (np.random.rand(5)+1j*np.random.rand(5)).astype(np.complex128)
>>> y = (np.random.rand(5)+1j*np.random.rand(5)).astype(np.complex128)
>>> x_gpu = gpuarray.to_gpu(x)
>>> y_gpu = gpuarray.to_gpu(y)
>>> h = cublasCreate()
>>> cublasZaxpy(h, x_gpu.size, alpha, x_gpu.gpudata, 1, y_gpu.gpudata, 1)
>>> cublasDestroy(h)
>>> np.allclose(y_gpu.get(), alpha*x+y)
True
```

Notes

Both *x* and *y* must contain *n* elements.

References

cublas<t>axpy

skcuda.cublas.cublasZcopy

skcuda.cublas.**cublasZcopy**(*handle*, *n*, *x*, *incx*, *y*, *incy*)

Vector copy (double precision complex)

Copies a double precision complex vector to another double precision complex vector.

Parameters

- **handle** (*int*) – CUBLAS context.
- **n** (*int*) – Number of elements in input vectors.
- **x** (*ctypes.c_void_p*) – Pointer to double precision complex input vector.
- **incx** (*int*) – Storage spacing between elements of *x*.
- **y** (*ctypes.c_void_p*) – Pointer to double precision complex output vector.
- **incy** (*int*) – Storage spacing between elements of *y*.

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> x = (np.random.rand(5)+np.random.rand(5)).astype(np.complex128)
>>> x_gpu = gpuarray.to_gpu(x)
>>> y_gpu = gpuarray.zeros_like(x_gpu)
>>> h = cublasCreate()
>>> cublasZcopy(h, x_gpu.size, x_gpu.gpudata, 1, y_gpu.gpudata, 1)
>>> cublasDestroy(h)
>>> np.allclose(y_gpu.get(), x_gpu.get())
True
```

Notes

Both *x* and *y* must contain *n* elements.

References

cublas<t>copy

skcuda.cublas.cublasZdotc

skcuda.cublas.**cublasZdotc**(*handle*, *n*, *x*, *incx*, *y*, *incy*)

Vector dot product (double precision complex)

Computes the dot product of two double precision complex vectors. cublasCdotc and cublasZdotc use the conjugate of the first vector when computing the dot product.

Parameters

- **handle** (*int*) – CUBLAS context.
- **n** (*int*) – Number of elements in input vectors.
- **x** (*ctypes.c_void_p*) – Pointer to double precision complex input vector.
- **incx** (*int*) – Storage spacing between elements of *x*.
- **y** (*ctypes.c_void_p*) – Pointer to double precision complex input/output vector.
- **incy** (*int*) – Storage spacing between elements of *y*.

Returns **d** – Dot product of *x* and *y*.

Return type np.complex128

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> x = (np.random.rand(5)+1j*np.random.rand(5)).astype(np.complex128)
>>> y = (np.random.rand(5)+1j*np.random.rand(5)).astype(np.complex128)
>>> x_gpu = gpuarray.to_gpu(x)
>>> y_gpu = gpuarray.to_gpu(y)
>>> h = cublasCreate()
>>> d = cublasZdotc(h, x_gpu.size, x_gpu.gpudata, 1, y_gpu.gpudata, 1)
>>> cublasDestroy(h)
>>> np.allclose(d, np.dot(np.conj(x), y))
True
```

Notes

Both *x* and *y* must contain *n* elements.

References

cublas<t>dot

skcuda.cublas.cublasZdotu

skcuda.cublas.**cublasZdotu**(*handle*, *n*, *x*, *incx*, *y*, *incy*)

Vector dot product (double precision complex)

Computes the dot product of two double precision complex vectors. cublasCdotc and cublasZdotc use the conjugate of the first vector when computing the dot product.

Parameters

- **handle** (*int*) – CUBLAS context.
- **n** (*int*) – Number of elements in input vectors.
- **x** (*ctypes.c_void_p*) – Pointer to double precision complex input vector.
- **incx** (*int*) – Storage spacing between elements of *x*.
- **y** (*ctypes.c_void_p*) – Pointer to double precision complex input/output vector.
- **incy** (*int*) – Storage spacing between elements of *y*.

Returns **d** – Dot product of *x* and *y*.

Return type np.complex128

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> x = (np.random.rand(5)+1j*np.random.rand(5)).astype(np.complex128)
>>> y = (np.random.rand(5)+1j*np.random.rand(5)).astype(np.complex128)
>>> x_gpu = gpuarray.to_gpu(x)
>>> y_gpu = gpuarray.to_gpu(y)
>>> h = cublasCreate()
>>> d = cublasZdotu(h, x_gpu.size, x_gpu.gpudata, 1, y_gpu.gpudata, 1)
>>> cublasDestroy(h)
>>> np.allclose(d, np.dot(x, y))
True
```

Notes

Both *x* and *y* must contain *n* elements.

References

cublas<t>dot

skcuda.cublas.cublasZdrot

skcuda.cublas.**cublasZdrot**(*handle*, *n*, *x*, *incx*, *y*, *incy*, *c*, *s*)

Apply a complex rotation to complex vectors (double precision)

Multiplies the double precision matrix *[[c, s], [-s.conj(), c]]* with the 2 x *n* double precision matrix *[[x.T], [y.T]]*.

Parameters

- **handle** (*int*) – CUBLAS context.
- **n** (*int*) – Number of elements in input vectors.
- **x** (*ctypes.c_void_p*) – Pointer to double precision complex input/output vector.

- **incx** (*int*) – Storage spacing between elements of *x*.
- **y** (*ctypes.c_void_p*) – Pointer to double precision complex input/output vector.
- **incy** (*int*) – Storage spacing between elements of *y*.
- **c** (*numpy.float64*) – Element of rotation matrix.
- **s** (*numpy.float64*) – Element of rotation matrix.

Notes

Both *x* and *y* must contain *n* elements.

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> s = np.float64(np.random.rand()); c = np.float64(np.random.rand());
>>> x = (np.random.rand(5)+1j*np.random.rand(5)).astype(np.complex128)
>>> y = (np.random.rand(5)+1j*np.random.rand(5)).astype(np.complex128)
>>> x_gpu = gpuarray.to_gpu(x)
>>> y_gpu = gpuarray.to_gpu(y)
>>> h = cublasCreate()
>>> cublasZdrot(h, x.size, x_gpu.gpudata, 1, y_gpu.gpudata, 1, c, s)
>>> cublasDestroy(h)
>>> np.allclose(x_gpu.get(), c*x+s*y)
True
>>> np.allclose(y_gpu.get(), -s.conj()*x+c*y)
True
```

References

cublas<t>rot

skcuda.cublas.cublasZdscal

skcuda.cublas.**cublasZdscal**(*handle*, *n*, *alpha*, *x*, *incx*)

Scale a double precision complex vector by a double precision real scalar.

Replaces a double precision complex vector *x* with *alpha * x*.

Parameters

- **handle** (*int*) – CUBLAS context.
- **n** (*int*) – Number of elements in input vectors.
- **alpha** (*numpy.float64*) – Scalar multiplier.
- **x** (*ctypes.c_void_p*) – Pointer to double precision complex input/output vector.
- **incx** (*int*) – Storage spacing between elements of *x*.

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> x = (np.random.rand(5)+1j*np.random.rand(5)).astype(np.complex128)
>>> x_gpu = gpuarray.to_gpu(x)
>>> alpha = np.float64(np.random.rand())
>>> h = cublasCreate()
>>> cublasZdscal(h, x.size, alpha, x_gpu.gpudata, 1)
>>> cublasDestroy(h)
>>> np.allclose(x_gpu.get(), alpha*x)
True
```

References

cublas<t>scal

skcuda.cublas.cublasZrot

skcuda.cublas.**cublasZrot**(*handle*, *n*, *x*, *incx*, *y*, *incy*, *c*, *s*)

Apply a complex rotation to complex vectors (double precision)

Multiplies the double precision matrix *[[c, s], [-s.conj(), c]]* with the 2 x *n* double precision matrix *[[x.T], [y.T]]*.

Parameters

- **handle** (*int*) – CUBLAS context.
- **n** (*int*) – Number of elements in input vectors.
- **x** (*ctypes.c_void_p*) – Pointer to double precision complex input/output vector.
- **incx** (*int*) – Storage spacing between elements of *x*.
- **y** (*ctypes.c_void_p*) – Pointer to double precision complex input/output vector.
- **incy** (*int*) – Storage spacing between elements of *y*.
- **c** (*numpy.float64*) – Element of rotation matrix.
- **s** (*numpy.complex128*) – Element of rotation matrix.

Notes

Both *x* and *y* must contain *n* elements.

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> s = np.complex128(np.random.rand()+1j*np.random.rand()); c = np.float64(np.
↪random.rand());
```

(continues on next page)

(continued from previous page)

```
>>> x = (np.random.rand(5)+1j*np.random.rand(5)).astype(np.complex128)
>>> y = (np.random.rand(5)+1j*np.random.rand(5)).astype(np.complex128)
>>> x_gpu = gpuarray.to_gpu(x)
>>> y_gpu = gpuarray.to_gpu(y)
>>> h = cublasCreate()
>>> cublasZrot(h, x.size, x_gpu.gpudata, 1, y_gpu.gpudata, 1, c, s)
>>> cublasDestroy(h)
>>> np.allclose(x_gpu.get(), c*x+s*y)
True
>>> np.allclose(y_gpu.get(), -s.conj()*x+c*y)
True
```

References

cublas<t>rot

skcuda.cublas.cublasZrotg

skcuda.cublas.**cublasZrotg**(*handle*, *a*, *b*)

Construct a double precision complex Givens rotation matrix.

Constructs the double precision complex Givens rotation matrix *G = [[c, s], [-s.conj(), c]]* such that *dot(G, [[a], [b]] == [[r], [0]]*, where *c**2+s**2 == 1* and *r == a**2+b**2* for real numbers and *c**2+(conj(s)*s) == 1* and *r == (a/abs(a))*sqrt(abs(a)**2+abs(b)**2)* for *a != 0* and *r == b* for *a == 0*.

Parameters

- **handle** (*int*) – CUBLAS context.
- **b** (*a,*) – Entries of vector whose second entry should be zeroed out by the rotation.

Returns

- **r** (*numpy.complex128*) – Defined above.
- **c** (*numpy.float64*) – Cosine component of rotation matrix.
- **s** (*numpy.complex128*) – Sine component of rotation matrix.

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> a = np.complex128(np.random.rand()+1j*np.random.rand())
>>> b = np.complex128(np.random.rand()+1j*np.random.rand())
>>> h = cublasCreate()
>>> r, c, s = cublasZrotg(h, a, b)
>>> cublasDestroy(h)
>>> np.allclose(np.dot(np.array([[c, s], [-np.conj(s), c]]), np.array([[a],␣
↪[b]])), np.array([[r], [0.0]]), atol=1e-6)
True
```

References

cublas<t>rotg

skcuda.cublas.cublasZscal

skcuda.cublas.**cublasZscal**(*handle*, *n*, *alpha*, *x*, *incx*)

Scale a double precision complex vector by a double precision complex scalar.

Replaces a double precision complex vector *x* with *alpha * x*.

Parameters

- **handle** (*int*) – CUBLAS context.
- **n** (*int*) – Number of elements in input vectors.
- **alpha** (*numpy.complex128*) – Scalar multiplier.
- **x** (*ctypes.c_void_p*) – Pointer to double precision complex input/output vector.
- **incx** (*int*) – Storage spacing between elements of *x*.

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> x = (np.random.rand(5)+1j*np.random.rand(5)).astype(np.complex128)
>>> x_gpu = gpuarray.to_gpu(x)
>>> alpha = np.complex128(np.random.rand()+1j*np.random.rand())
>>> h = cublasCreate()
>>> cublasZscal(h, x.size, alpha, x_gpu.gpudata, 1)
>>> cublasDestroy(h)
>>> np.allclose(x_gpu.get(), alpha*x)
True
```

References

cublas<t>scal

skcuda.cublas.cublasZswap

skcuda.cublas.**cublasZswap**(*handle*, *n*, *x*, *incx*, *y*, *incy*)

Swap double precision complex vectors.

Swaps the contents of one double precision complex vector with those of another double precision complex vector.

Parameters

- **handle** (*int*) – CUBLAS context.
- **n** (*int*) – Number of elements in input vectors.
- **x** (*ctypes.c_void_p*) – Pointer to double precision complex input/output vector.

- **incx** (*int*) – Storage spacing between elements of *x*.
- **y** (*ctypes.c_void_p*) – Pointer to double precision complex input/output vector.
- **incy** (*int*) – Storage spacing between elements of *y*.

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> x = (np.random.rand(5)+1j*np.random.rand(5)).astype(np.complex128)
>>> y = (np.random.rand(5)+1j*np.random.rand(5)).astype(np.complex128)
>>> x_gpu = gpuarray.to_gpu(x)
>>> y_gpu = gpuarray.to_gpu(y)
>>> h = cublasCreate()
>>> cublasZswap(h, x.size, x_gpu.gpudata, 1, y_gpu.gpudata, 1)
>>> cublasDestroy(h)
>>> np.allclose(x_gpu.get(), y)
True
>>> np.allclose(y_gpu.get(), x)
True
```

Notes

Both *x* and *y* must contain *n* elements.

References

cublas<t>swap

Single Precision BLAS2 Routines

cublasSgbmv	Matrix-vector product for real single precision general banded matrix.
cublasSgemv	Matrix-vector product for real single precision general matrix.
cublasSger	Rank-1 operation on real single precision general matrix.
cublasSsbmv	Matrix-vector product for real single precision symmetric-banded matrix.
cublasSspmv	Matrix-vector product for real single precision symmetric packed matrix.
cublasSspr	Rank-1 operation on real single precision symmetric packed matrix.
cublasSspr2	Rank-2 operation on real single precision symmetric packed matrix.
cublasSsymv	Matrix-vector product for real symmetric matrix.

Continued on next page

Table 4 – continued from previous page

cublasSsyr	Rank-1 operation on real single precision symmetric matrix.
cublasSsyr2	Rank-2 operation on real single precision symmetric matrix.
cublasStbmv	Matrix-vector product for real single precision triangular banded matrix.
cublasStbsv	Solve real single precision triangular banded system with one right-hand side.
cublasStpmv	Matrix-vector product for real single precision triangular packed matrix.
cublasStpsv	Solve real triangular packed system with one right-hand side.
cublasStrmv	Matrix-vector product for real single precision triangular matrix.
cublasStrsv	Solve real triangular system with one right-hand side.
cublasCgbmv	Matrix-vector product for complex single precision general banded matrix.
cublasCgemv	Matrix-vector product for complex single precision general matrix.
cublasCgerc	Rank-1 operation on complex single precision general matrix.
cublasCgeru	Rank-1 operation on complex single precision general matrix.
cublasChbmv	Matrix-vector product for single precision Hermitian banded matrix.
cublasChemv	Matrix vector product for single precision Hermitian matrix.
cublasCher	Rank-1 operation on single precision Hermitian matrix.
cublasCher2	Rank-2 operation on single precision Hermitian matrix.
cublasChpmv	Matrix-vector product for single precision Hermitian packed matrix.
cublasChpr	Rank-1 operation on single precision Hermitian packed matrix.
cublasChpr2	Rank-2 operation on single precision Hermitian packed matrix.
cublasCtbmv	Matrix-vector product for complex single precision triangular banded matrix.
cublasCtbsv	Solve complex single precision triangular banded system with one right-hand side.
cublasCtpmv	Matrix-vector product for complex single precision triangular packed matrix.
cublasCtpsv	Solve complex single precision triangular packed system with one right-hand side.
cublasCtrmv	Matrix-vector product for complex single precision triangular matrix.
cublasCtrsv	Solve complex single precision triangular system with one right-hand side.

skcuda.cublas.cublasSgbmv

skcuda.cublas.**cublasSgbmv**(*handle*, *trans*, *m*, *n*, *kl*, *ku*, *alpha*, *A*, *lda*, *x*, *incx*, *beta*, *y*, *incy*)
Matrix-vector product for real single precision general banded matrix.

References

cublas<t>gbmv

skcuda.cublas.cublasSgemv

skcuda.cublas.**cublasSgemv**(*handle*, *trans*, *m*, *n*, *alpha*, *A*, *lda*, *x*, *incx*, *beta*, *y*, *incy*)
Matrix-vector product for real single precision general matrix.

References

cublas<t>gemv

skcuda.cublas.cublasSger

skcuda.cublas.**cublasSger**(*handle*, *m*, *n*, *alpha*, *x*, *incx*, *y*, *incy*, *A*, *lda*)
Rank-1 operation on real single precision general matrix.

References

cublas<t>ger

skcuda.cublas.cublasSsbmv

skcuda.cublas.**cublasSsbmv**(*handle*, *uplo*, *n*, *k*, *alpha*, *A*, *lda*, *x*, *incx*, *beta*, *y*, *incy*)
Matrix-vector product for real single precision symmetric-banded matrix.

References

cublas<t>sbmv

skcuda.cublas.cublasSspmv

skcuda.cublas.**cublasSspmv**(*handle*, *uplo*, *n*, *alpha*, *AP*, *x*, *incx*, *beta*, *y*, *incy*)
Matrix-vector product for real single precision symmetric packed matrix.

References

cublas<t>spmv

skcuda.cublas.cublasSspr

skcuda.cublas.**cublasSspr**(*handle*, *uplo*, *n*, *alpha*, *x*, *incx*, *AP*)
Rank-1 operation on real single precision symmetric packed matrix.

References

cublas<t>spr

skcuda.cublas.cublasSspr2

skcuda.cublas.**cublasSspr2**(*handle*, *uplo*, *n*, *alpha*, *x*, *incx*, *y*, *incy*, *AP*)
Rank-2 operation on real single precision symmetric packed matrix.

References

cublas<t>spr2

skcuda.cublas.cublasSsymv

skcuda.cublas.**cublasSsymv**(*handle*, *uplo*, *n*, *alpha*, *A*, *lda*, *x*, *incx*, *beta*, *y*, *incy*)
Matrix-vector product for real symmetric matrix.

References

cublas<t>symv

skcuda.cublas.cublasSsyr

skcuda.cublas.**cublasSsyr**(*handle*, *uplo*, *n*, *alpha*, *x*, *incx*, *A*, *lda*)
Rank-1 operation on real single precision symmetric matrix.

References

cublas<t>syr

skcuda.cublas.cublasSsyr2

skcuda.cublas.**cublasSsyr2**(*handle*, *uplo*, *n*, *alpha*, *x*, *incx*, *y*, *incy*, *A*, *lda*)
Rank-2 operation on real single precision symmetric matrix.

References

cublas<t>syr2

skcuda.cublas.cublasStbmv

skcuda.cublas.**cublasStbmv**(*handle*, *uplo*, *trans*, *diag*, *n*, *k*, *A*, *lda*, *x*, *incx*)
Matrix-vector product for real single precision triangular banded matrix.

References

cublas<t>tbmv

skcuda.cublas.cublasStbsv

skcuda.cublas.**cublasStbsv**(*handle*, *uplo*, *trans*, *diag*, *n*, *k*, *A*, *lda*, *x*, *incx*)
Solve real single precision triangular banded system with one right-hand side.

References

cublas<t>tbsv

skcuda.cublas.cublasStpmv

skcuda.cublas.**cublasStpmv**(*handle*, *uplo*, *trans*, *diag*, *n*, *AP*, *x*, *incx*)
Matrix-vector product for real single precision triangular packed matrix.

References

cublas<t>tpmv

skcuda.cublas.cublasStpsv

skcuda.cublas.**cublasStpsv**(*handle*, *uplo*, *trans*, *diag*, *n*, *AP*, *x*, *incx*)
Solve real triangular packed system with one right-hand side.

References

cublas<t>tpsv

skcuda.cublas.cublasStrmv

skcuda.cublas.**cublasStrmv**(*handle*, *uplo*, *trans*, *diag*, *n*, *A*, *lda*, *x*, *inx*)
Matrix-vector product for real single precision triangular matrix.

References

cublas<t>trmv

skcuda.cublas.cublasStrsv

skcuda.cublas.**cublasStrsv**(*handle*, *uplo*, *trans*, *diag*, *n*, *A*, *lda*, *x*, *incx*)
Solve real triangular system with one right-hand side.

References

cublas<t>trsv

skcuda.cublas.cublasCgbmv

skcuda.cublas.**cublasCgbmv**(*handle*, *trans*, *m*, *n*, *kl*, *ku*, *alpha*, *A*, *lda*, *x*, *incx*, *beta*, *y*, *incy*)
Matrix-vector product for complex single precision general banded matrix.

References

cublas<t>gbmv

skcuda.cublas.cublasCgemv

skcuda.cublas.**cublasCgemv**(*handle*, *trans*, *m*, *n*, *alpha*, *A*, *lda*, *x*, *incx*, *beta*, *y*, *incy*)
Matrix-vector product for complex single precision general matrix.

References

cublas<t>gemv

skcuda.cublas.cublasCgerc

skcuda.cublas.**cublasCgerc**(*handle*, *m*, *n*, *alpha*, *x*, *incx*, *y*, *incy*, *A*, *lda*)
Rank-1 operation on complex single precision general matrix.

References

cublas<t>ger

skcuda.cublas.cublasCgeru

skcuda.cublas.**cublasCgeru**(*handle*, *m*, *n*, *alpha*, *x*, *incx*, *y*, *incy*, *A*, *lda*)
Rank-1 operation on complex single precision general matrix.

References

cublas<t>ger

skcuda.cublas.cublasChbmv

skcuda.cublas.**cublasChbmv**(*handle*, *uplo*, *n*, *k*, *alpha*, *A*, *lda*, *x*, *incx*, *beta*, *y*, *incy*)
Matrix-vector product for single precision Hermitian banded matrix.

References

cublas<t>hbmv

skcuda.cublas.cublasChemv

skcuda.cublas.**cublasChemv**(*handle*, *uplo*, *n*, *alpha*, *A*, *lda*, *x*, *incx*, *beta*, *y*, *incy*)
Matrix vector product for single precision Hermitian matrix.

References

cublas<t>hemv

skcuda.cublas.cublasCher

skcuda.cublas.**cublasCher**(*handle*, *uplo*, *n*, *alpha*, *x*, *incx*, *A*, *lda*)
Rank-1 operation on single precision Hermitian matrix.

References

cublas<t>her

skcuda.cublas.cublasCher2

skcuda.cublas.**cublasCher2**(*handle*, *uplo*, *n*, *alpha*, *x*, *incx*, *y*, *incy*, *A*, *lda*)
Rank-2 operation on single precision Hermitian matrix.

References

cublas<t>her2

skcuda.cublas.cublasChpmv

skcuda.cublas.**cublasChpmv**(*handle*, *uplo*, *n*, *alpha*, *AP*, *x*, *incx*, *beta*, *y*, *incy*)
Matrix-vector product for single precision Hermitian packed matrix.

References

cublas<t>hpmv

skcuda.cublas.cublasChpr

skcuda.cublas.**cublasChpr**(*handle*, *uplo*, *n*, *alpha*, *x*, *incx*, *AP*)
Rank-1 operation on single precision Hermitian packed matrix.

References

cublas<t>hpr

skcuda.cublas.cublasChpr2

skcuda.cublas.**cublasChpr2**(*handle*, *uplo*, *n*, *alpha*, *x*, *inx*, *y*, *incy*, *AP*)
Rank-2 operation on single precision Hermitian packed matrix.

References

cublas<t>hpr2

skcuda.cublas.cublasCtbmv

skcuda.cublas.**cublasCtbmv**(*handle*, *uplo*, *trans*, *diag*, *n*, *k*, *A*, *lda*, *x*, *incx*)
Matrix-vector product for complex single precision triangular banded matrix.

References

cublas<t>tbmv

skcuda.cublas.cublasCtbsv

skcuda.cublas.**cublasCtbsv**(*handle*, *uplo*, *trans*, *diag*, *n*, *k*, *A*, *lda*, *x*, *incx*)
Solve complex single precision triangular banded system with one right-hand side.

References

cublas<t>tbsv

skcuda.cublas.cublasCtpmv

skcuda.cublas.**cublasCtpmv**(*handle*, *uplo*, *trans*, *diag*, *n*, *AP*, *x*, *incx*)
Matrix-vector product for complex single precision triangular packed matrix.

References

cublas<t>tpmv

skcuda.cublas.cublasCtpsv

skcuda.cublas.**cublasCtpsv**(*handle*, *uplo*, *trans*, *diag*, *n*, *AP*, *x*, *incx*)
Solve complex single precision triangular packed system with one right-hand side.

References

cublas<t>tpsv

skcuda.cublas.cublasCtrmv

skcuda.cublas.**cublasCtrmv**(*handle*, *uplo*, *trans*, *diag*, *n*, *A*, *lda*, *x*, *incx*)
Matrix-vector product for complex single precision triangular matrix.

References

cublas<t>trmv

skcuda.cublas.cublasCtrsv

skcuda.cublas.**cublasCtrsv**(*handle*, *uplo*, *trans*, *diag*, *n*, *A*, *lda*, *x*, *incx*)
Solve complex single precision triangular system with one right-hand side.

References

cublas<t>trsv

Double Precision BLAS2 Routines

cublasDgbmv	Matrix-vector product for real double precision general banded matrix.
cublasDgemv	Matrix-vector product for real double precision general matrix.
cublasDger	Rank-1 operation on real double precision general matrix.
cublasDsbmv	Matrix-vector product for real double precision symmetric-banded matrix.
cublasDspmv	Matrix-vector product for real double precision symmetric packed matrix.
cublasDspr	Rank-1 operation on real double precision symmetric packed matrix.
cublasDspr2	Rank-2 operation on real double precision symmetric packed matrix.
cublasDsymv	Matrix-vector product for real double precision symmetric matrix.

Continued on next page

Table 5 – continued from previous page

cublasDsyr	Rank-1 operation on real double precision symmetric matrix.
cublasDsyr2	Rank-2 operation on real double precision symmetric matrix.
cublasDtbmv	Matrix-vector product for real double precision triangular banded matrix.
cublasDtbsv	Solve real double precision triangular banded system with one right-hand side.
cublasDtpmv	Matrix-vector product for real double precision triangular packed matrix.
cublasDtpsv	Solve real double precision triangular packed system with one right-hand side.
cublasDtrmv	Matrix-vector product for real double precision triangular matrix.
cublasDtrsv	Solve real double precision triangular system with one right-hand side.
cublasZgbmv	Matrix-vector product for complex double precision general banded matrix.
cublasZgemv	Matrix-vector product for complex double precision general matrix.
cublasZgerc	Rank-1 operation on complex double precision general matrix.
cublasZgeru	Rank-1 operation on complex double precision general matrix.
cublasZhbmv	Matrix-vector product for double precision Hermitian banded matrix.
cublasZhemv	Matrix-vector product for double precision Hermitian matrix.
cublasZher	Rank-1 operation on double precision Hermitian matrix.
cublasZher2	Rank-2 operation on double precision Hermitian matrix.
cublasZhpmv	Matrix-vector product for double precision Hermitian packed matrix.
cublasZhpr	Rank-1 operation on double precision Hermitian packed matrix.
cublasZhpr2	Rank-2 operation on double precision Hermitian packed matrix.
cublasZtbmv	Matrix-vector product for complex double triangular banded matrix.
cublasZtbsv	Solve complex double precision triangular banded system with one right-hand side.
cublasZtpmv	Matrix-vector product for complex double precision triangular packed matrix.
cublasZtpsv	Solve complex double precision triangular packed system with one right-hand size.
cublasZtrmv	Matrix-vector product for complex double precision triangular matrix.
cublasZtrsv	Solve complex double precision triangular system with one right-hand side.

skcuda.cublas.cublasDgbmv

skcuda.cublas.**cublasDgbmv**(*handle*, *trans*, *m*, *n*, *kl*, *ku*, *alpha*, *A*, *lda*, *x*, *incx*, *beta*, *y*, *incy*)
Matrix-vector product for real double precision general banded matrix.

References

cublas<t>gbmv

skcuda.cublas.cublasDgemv

skcuda.cublas.**cublasDgemv**(*handle*, *trans*, *m*, *n*, *alpha*, *A*, *lda*, *x*, *incx*, *beta*, *y*, *incy*)
Matrix-vector product for real double precision general matrix.

References

cublas<t>gemv

skcuda.cublas.cublasDger

skcuda.cublas.**cublasDger**(*handle*, *m*, *n*, *alpha*, *x*, *incx*, *y*, *incy*, *A*, *lda*)
Rank-1 operation on real double precision general matrix.

References

cublas<t>ger

skcuda.cublas.cublasDsbmv

skcuda.cublas.**cublasDsbmv**(*handle*, *uplo*, *n*, *k*, *alpha*, *A*, *lda*, *x*, *incx*, *beta*, *y*, *incy*)
Matrix-vector product for real double precision symmetric-banded matrix.

References

cublas<t>ger

skcuda.cublas.cublasDspmv

skcuda.cublas.**cublasDspmv**(*handle*, *uplo*, *n*, *alpha*, *AP*, *x*, *incx*, *beta*, *y*, *incy*)
Matrix-vector product for real double precision symmetric packed matrix.

References

cublas<t>spmv

skcuda.cublas.cublasDspr

skcuda.cublas.**cublasDspr**(*handle*, *uplo*, *n*, *alpha*, *x*, *incx*, *AP*)
Rank-1 operation on real double precision symmetric packed matrix.

References

cublas<t>spr

skcuda.cublas.cublasDspr2

skcuda.cublas.**cublasDspr2**(*handle*, *uplo*, *n*, *alpha*, *x*, *incx*, *y*, *incy*, *AP*)
Rank-2 operation on real double precision symmetric packed matrix.

References

cublas<t>spr2

skcuda.cublas.cublasDsymv

skcuda.cublas.**cublasDsymv**(*handle*, *uplo*, *n*, *alpha*, *A*, *lda*, *x*, *incx*, *beta*, *y*, *incy*)
Matrix-vector product for real double precision symmetric matrix.

References

cublas<t>symv

skcuda.cublas.cublasDsyr

skcuda.cublas.**cublasDsyr**(*handle*, *uplo*, *n*, *alpha*, *x*, *incx*, *A*, *lda*)
Rank-1 operation on real double precision symmetric matrix.

References

cublas<t>syr

skcuda.cublas.cublasDsyr2

skcuda.cublas.**cublasDsyr2**(*handle*, *uplo*, *n*, *alpha*, *x*, *incx*, *y*, *incy*, *A*, *lda*)
Rank-2 operation on real double precision symmetric matrix.

References

cublas<t>syr2

skcuda.cublas.cublasDtbmv

skcuda.cublas.**cublasDtbmv**(*handle*, *uplo*, *trans*, *diag*, *n*, *k*, *A*, *lda*, *x*, *incx*)
Matrix-vector product for real double precision triangular banded matrix.

References

cublas<t>tbmv

skcuda.cublas.cublasDtbsv

skcuda.cublas.**cublasDtbsv**(*handle*, *uplo*, *trans*, *diag*, *n*, *k*, *A*, *lda*, *x*, *incx*)
Solve real double precision triangular banded system with one right-hand side.

References

cublas<t>tbsv

skcuda.cublas.cublasDtpmv

skcuda.cublas.**cublasDtpmv**(*handle*, *uplo*, *trans*, *diag*, *n*, *AP*, *x*, *incx*)
Matrix-vector product for real double precision triangular packed matrix.

References

cublas<t>tpmv

skcuda.cublas.cublasDtpsv

skcuda.cublas.**cublasDtpsv**(*handle*, *uplo*, *trans*, *diag*, *n*, *AP*, *x*, *incx*)
Solve real double precision triangular packed system with one right-hand side.

References

cublas<t>tpsv

skcuda.cublas.cublasDtrmv

skcuda.cublas.**cublasDtrmv**(*handle*, *uplo*, *trans*, *diag*, *n*, *A*, *lda*, *x*, *inx*)
Matrix-vector product for real double precision triangular matrix.

References

cublas<t>trmv

skcuda.cublas.cublasDtrsv

skcuda.cublas.**cublasDtrsv**(*handle*, *uplo*, *trans*, *diag*, *n*, *A*, *lda*, *x*, *incx*)
Solve real double precision triangular system with one right-hand side.

References

cublas<t>trsv

skcuda.cublas.cublasZgbmv

skcuda.cublas.**cublasZgbmv**(*handle*, *trans*, *m*, *n*, *kl*, *ku*, *alpha*, *A*, *lda*, *x*, *incx*, *beta*, *y*, *incy*)
Matrix-vector product for complex double precision general banded matrix.

References

cublas<t>gbmv

skcuda.cublas.cublasZgemv

skcuda.cublas.**cublasZgemv**(*handle*, *trans*, *m*, *n*, *alpha*, *A*, *lda*, *x*, *incx*, *beta*, *y*, *incy*)
Matrix-vector product for complex double precision general matrix.

References

cublas<t>gemv

skcuda.cublas.cublasZgerc

skcuda.cublas.**cublasZgerc**(*handle*, *m*, *n*, *alpha*, *x*, *incx*, *y*, *incy*, *A*, *lda*)
Rank-1 operation on complex double precision general matrix.

References

cublas<t>ger

skcuda.cublas.cublasZgeru

skcuda.cublas.**cublasZgeru**(*handle*, *m*, *n*, *alpha*, *x*, *incx*, *y*, *incy*, *A*, *lda*)
Rank-1 operation on complex double precision general matrix.

References

cublas<t>ger

skcuda.cublas.cublasZhbmv

skcuda.cublas.**cublasZhbmv**(*handle*, *uplo*, *n*, *k*, *alpha*, *A*, *lda*, *x*, *incx*, *beta*, *y*, *incy*)
Matrix-vector product for double precision Hermitian banded matrix.

References

cublas<t>hbmv

skcuda.cublas.cublasZhemv

skcuda.cublas.**cublasZhemv**(*handle*, *uplo*, *n*, *alpha*, *A*, *lda*, *x*, *incx*, *beta*, *y*, *incy*)
Matrix-vector product for double precision Hermitian matrix.

References

cublas<t>hemv

skcuda.cublas.cublasZher

skcuda.cublas.**cublasZher**(*handle*, *uplo*, *n*, *alpha*, *x*, *incx*, *A*, *lda*)
Rank-1 operation on double precision Hermitian matrix.

References

cublas<t>her

skcuda.cublas.cublasZher2

skcuda.cublas.**cublasZher2**(*handle*, *uplo*, *n*, *alpha*, *x*, *incx*, *y*, *incy*, *A*, *lda*)
Rank-2 operation on double precision Hermitian matrix.

References

cublas<t>her2

skcuda.cublas.cublasZhpmv

skcuda.cublas.**cublasZhpmv**(*handle*, *uplo*, *n*, *alpha*, *AP*, *x*, *incx*, *beta*, *y*, *incy*)
Matrix-vector product for double precision Hermitian packed matrix.

References

cublas<t>hpmv

skcuda.cublas.cublasZhpr

skcuda.cublas.**cublasZhpr**(*handle*, *uplo*, *n*, *alpha*, *x*, *incx*, *AP*)
Rank-1 operation on double precision Hermitian packed matrix.

References

cublas<t>hpr

skcuda.cublas.cublasZhpr2

skcuda.cublas.**cublasZhpr2**(*handle*, *uplo*, *n*, *alpha*, *x*, *inx*, *y*, *incy*, *AP*)
Rank-2 operation on double precision Hermitian packed matrix.

References

cublas<t>hpr2

skcuda.cublas.cublasZtbmv

skcuda.cublas.**cublasZtbmv**(*handle*, *uplo*, *trans*, *diag*, *n*, *k*, *A*, *lda*, *x*, *incx*)
Matrix-vector product for complex double triangular banded matrix.

References

cublas<t>tbmv

skcuda.cublas.cublasZtbsv

skcuda.cublas.**cublasZtbsv**(*handle*, *uplo*, *trans*, *diag*, *n*, *k*, *A*, *lda*, *x*, *incx*)
Solve complex double precision triangular banded system with one right-hand side.

References

cublas<t>tbsv

skcuda.cublas.cublasZtpmv

skcuda.cublas.**cublasZtpmv**(*handle*, *uplo*, *trans*, *diag*, *n*, *AP*, *x*, *incx*)
Matrix-vector product for complex double precision triangular packed matrix.

References

cublas<t>tpmv

skcuda.cublas.cublasZtpsv

skcuda.cublas.**cublasZtpsv**(*handle*, *uplo*, *trans*, *diag*, *n*, *AP*, *x*, *incx*)
Solve complex double precision triangular packed system with one right-hand size.

References

cublas<t>tpsv

skcuda.cublas.cublasZtrmv

skcuda.cublas.**cublasZtrmv**(*handle*, *uplo*, *trans*, *diag*, *n*, *A*, *lda*, *x*, *incx*)
Matrix-vector product for complex double precision triangular matrix.

References

cublas<t>trmv

skcuda.cublas.cublasZtrsv

skcuda.cublas.**cublasZtrsv**(*handle*, *uplo*, *trans*, *diag*, *n*, *A*, *lda*, *x*, *incx*)
Solve complex double precision triangular system with one right-hand side.

References

cublas<t>trsv

Single Precision BLAS3 Routines

cublasSgemm	Matrix-matrix product for real single precision general matrix.
cublasSsymm	Matrix-matrix product for real single precision symmetric matrix.
cublasSsyrk	Rank-k operation on real single precision symmetric matrix.
cublasSsyr2k	Rank-2k operation on real single precision symmetric matrix.
cublasStrmm	Matrix-matrix product for real single precision triangular matrix.
cublasStrsm	Solve a real single precision triangular system with multiple right-hand sides.
cublasCgemm	Matrix-matrix product for complex single precision general matrix.
cublasChemm	Matrix-matrix product for single precision Hermitian matrix.

Continued on next page

Table 6 – continued from previous page

cublasCherk	Rank-k operation on single precision Hermitian matrix.
cublasCher2k	Rank-2k operation on single precision Hermitian matrix.
cublasCsymm	Matrix-matrix product for complex single precision symmetric matrix.
cublasCsyrk	Rank-k operation on complex single precision symmetric matrix.
cublasCsyr2k	Rank-2k operation on complex single precision symmetric matrix.
cublasCtrmm	Matrix-matrix product for complex single precision triangular matrix.
cublasCtrsm	Solve a complex single precision triangular system with multiple right-hand sides.

skcuda.cublas.cublasSgemm

skcuda.cublas.**cublasSgemm**(*handle*, *transa*, *transb*, *m*, *n*, *k*, *alpha*, *A*, *lda*, *B*, *ldb*, *beta*, *C*, *ldc*)
Matrix-matrix product for real single precision general matrix.

References

cublas<t>gemm

skcuda.cublas.cublasSsymm

skcuda.cublas.**cublasSsymm**(*handle*, *side*, *uplo*, *m*, *n*, *alpha*, *A*, *lda*, *B*, *ldb*, *beta*, *C*, *ldc*)
Matrix-matrix product for real single precision symmetric matrix.

References

cublas<t>symm

skcuda.cublas.cublasSsyrk

skcuda.cublas.**cublasSsyrk**(*handle*, *uplo*, *trans*, *n*, *k*, *alpha*, *A*, *lda*, *beta*, *C*, *ldc*)
Rank-k operation on real single precision symmetric matrix.

References

cublas<t>syrk

skcuda.cublas.cublasSsyr2k

skcuda.cublas.**cublasSsyr2k**(*handle*, *uplo*, *trans*, *n*, *k*, *alpha*, *A*, *lda*, *B*, *ldb*, *beta*, *C*, *ldc*)
Rank-2k operation on real single precision symmetric matrix.

References

cublas<t>syr2k

skcuda.cublas.cublasStrmm

skcuda.cublas.**cublasStrmm**(*handle*, *side*, *uplo*, *trans*, *diag*, *m*, *n*, *alpha*, *A*, *lda*, *B*, *ldb*, *C*, *ldc*)
Matrix-matrix product for real single precision triangular matrix.

References

cublas<t>trmm

skcuda.cublas.cublasStrsm

skcuda.cublas.**cublasStrsm**(*handle*, *side*, *uplo*, *trans*, *diag*, *m*, *n*, *alpha*, *A*, *lda*, *B*, *ldb*)
Solve a real single precision triangular system with multiple right-hand sides.

References

cublas<t>trsm

skcuda.cublas.cublasCgemm

skcuda.cublas.**cublasCgemm**(*handle*, *transa*, *transb*, *m*, *n*, *k*, *alpha*, *A*, *lda*, *B*, *ldb*, *beta*, *C*, *ldc*)
Matrix-matrix product for complex single precision general matrix.

References

cublas<t>gemm

skcuda.cublas.cublasChemm

skcuda.cublas.**cublasChemm**(*handle*, *side*, *uplo*, *m*, *n*, *alpha*, *A*, *lda*, *B*, *ldb*, *beta*, *C*, *ldc*)
Matrix-matrix product for single precision Hermitian matrix.

References

cublas<t>hemm

skcuda.cublas.cublasCherk

skcuda.cublas.**cublasCherk**(*handle*, *uplo*, *trans*, *n*, *k*, *alpha*, *A*, *lda*, *beta*, *C*, *ldc*)
Rank-k operation on single precision Hermitian matrix.

References

cublas<t>herk

skcuda.cublas.cublasCher2k

skcuda.cublas.**cublasCher2k**(*handle*, *uplo*, *trans*, *n*, *k*, *alpha*, *A*, *lda*, *B*, *ldb*, *beta*, *C*, *ldc*)
Rank-2k operation on single precision Hermitian matrix.

References

cublas<t>her2k

skcuda.cublas.cublasCsymm

skcuda.cublas.**cublasCsymm**(*handle*, *side*, *uplo*, *m*, *n*, *alpha*, *A*, *lda*, *B*, *ldb*, *beta*, *C*, *ldc*)
Matrix-matrix product for complex single precision symmetric matrix.

References

cublas<t>symm

skcuda.cublas.cublasCsyrk

skcuda.cublas.**cublasCsyrk**(*handle*, *uplo*, *trans*, *n*, *k*, *alpha*, *A*, *lda*, *beta*, *C*, *ldc*)
Rank-k operation on complex single precision symmetric matrix.

References

cublas<t>syrk

skcuda.cublas.cublasCsyr2k

skcuda.cublas.**cublasCsyr2k**(*handle*, *uplo*, *trans*, *n*, *k*, *alpha*, *A*, *lda*, *B*, *ldb*, *beta*, *C*, *ldc*)
Rank-2k operation on complex single precision symmetric matrix.

References

cublas<t>syr2k

skcuda.cublas.cublasCtrmm

skcuda.cublas.**cublasCtrmm**(*handle*, *side*, *uplo*, *trans*, *diag*, *m*, *n*, *alpha*, *A*, *lda*, *B*, *ldb*, *C*, *ldc*)
Matrix-matrix product for complex single precision triangular matrix.

References

cublas<t>trmm

skcuda.cublas.cublasCtrsm

skcuda.cublas.**cublasCtrsm**(*handle*, *side*, *uplo*, *trans*, *diag*, *m*, *n*, *alpha*, *A*, *lda*, *B*, *ldb*)
Solve a complex single precision triangular system with multiple right-hand sides.

References

cublas<t>trsm

Double Precision BLAS3 Routines

cublasDgemm	Matrix-matrix product for real double precision general matrix.
cublasDsymm	Matrix-matrix product for real double precision symmetric matrix.
cublasDsyrk	Rank-k operation on real double precision symmetric matrix.
cublasDsyr2k	Rank-2k operation on real double precision symmetric matrix.
cublasDtrmm	Matrix-matrix product for real double precision triangular matrix.
cublasDtrsm	Solve a real double precision triangular system with multiple right-hand sides.
cublasZgemm	Matrix-matrix product for complex double precision general matrix.
cublasZhemm	Matrix-matrix product for double precision Hermitian matrix.
cublasZherk	Rank-k operation on double precision Hermitian matrix.
cublasZher2k	Rank-2k operation on double precision Hermitian matrix.
cublasZsymm	Matrix-matrix product for complex double precision symmetric matrix.
cublasZsyrk	Rank-k operation on complex double precision symmetric matrix.
cublasZsyr2k	Rank-2k operation on complex double precision symmetric matrix.
cublasZtrmm	Matrix-matrix product for complex double precision triangular matrix.
cublasZtrsm	Solve complex double precision triangular system with multiple right-hand sides.

skcuda.cublas.cublasDgemm

skcuda.cublas.**cublasDgemm**(*handle*, *transa*, *transb*, *m*, *n*, *k*, *alpha*, *A*, *lda*, *B*, *ldb*, *beta*, *C*, *ldc*)
Matrix-matrix product for real double precision general matrix.

References

cublas<t>gemm

skcuda.cublas.cublasDsymm

skcuda.cublas.**cublasDsymm**(*handle*, *side*, *uplo*, *m*, *n*, *alpha*, *A*, *lda*, *B*, *ldb*, *beta*, *C*, *ldc*)
Matrix-matrix product for real double precision symmetric matrix.

References

cublas<t>symm

skcuda.cublas.cublasDsyrk

skcuda.cublas.**cublasDsyrk**(*handle*, *uplo*, *trans*, *n*, *k*, *alpha*, *A*, *lda*, *beta*, *C*, *ldc*)
Rank-k operation on real double precision symmetric matrix.

References

cublas<t>syrk

skcuda.cublas.cublasDsyr2k

skcuda.cublas.**cublasDsyr2k**(*handle*, *uplo*, *trans*, *n*, *k*, *alpha*, *A*, *lda*, *B*, *ldb*, *beta*, *C*, *ldc*)
Rank-2k operation on real double precision symmetric matrix.

References

cublas<t>syr2k

skcuda.cublas.cublasDtrmm

skcuda.cublas.**cublasDtrmm**(*handle*, *side*, *uplo*, *trans*, *diag*, *m*, *n*, *alpha*, *A*, *lda*, *B*, *ldb*, *C*, *ldc*)
Matrix-matrix product for real double precision triangular matrix.

References

cublas<t>trmm

skcuda.cublas.cublasDtrsm

skcuda.cublas.**cublasDtrsm**(*handle*, *side*, *uplo*, *trans*, *diag*, *m*, *n*, *alpha*, *A*, *lda*, *B*, *ldb*)
Solve a real double precision triangular system with multiple right-hand sides.

References

cublas<t>trsm

skcuda.cublas.cublasZgemm

skcuda.cublas.**cublasZgemm**(*handle*, *transa*, *transb*, *m*, *n*, *k*, *alpha*, *A*, *lda*, *B*, *ldb*, *beta*, *C*, *ldc*)
Matrix-matrix product for complex double precision general matrix.

References

cublas<t>gemm

skcuda.cublas.cublasZhemm

skcuda.cublas.**cublasZhemm**(*handle*, *side*, *uplo*, *m*, *n*, *alpha*, *A*, *lda*, *B*, *ldb*, *beta*, *C*, *ldc*)
Matrix-matrix product for double precision Hermitian matrix.

References

cublas<t>hemm

skcuda.cublas.cublasZherk

skcuda.cublas.**cublasZherk**(*handle*, *uplo*, *trans*, *n*, *k*, *alpha*, *A*, *lda*, *beta*, *C*, *ldc*)
Rank-k operation on double precision Hermitian matrix.

References

cublas<t>herk

skcuda.cublas.cublasZher2k

skcuda.cublas.**cublasZher2k**(*handle*, *uplo*, *trans*, *n*, *k*, *alpha*, *A*, *lda*, *B*, *ldb*, *beta*, *C*, *ldc*)
Rank-2k operation on double precision Hermitian matrix.

References

cublas<t>her2k

skcuda.cublas.cublasZsymm

skcuda.cublas.**cublasZsymm**(*handle*, *side*, *uplo*, *m*, *n*, *alpha*, *A*, *lda*, *B*, *ldb*, *beta*, *C*, *ldc*)
Matrix-matrix product for complex double precision symmetric matrix.

References

cublas<t>symm

skcuda.cublas.cublasZsyrk

skcuda.cublas.**cublasZsyrk**(*handle*, *uplo*, *trans*, *n*, *k*, *alpha*, *A*, *lda*, *beta*, *C*, *ldc*)
Rank-k operation on complex double precision symmetric matrix.

References

cublas<t>syrk

skcuda.cublas.cublasZsyr2k

skcuda.cublas.**cublasZsyr2k**(*handle*, *uplo*, *trans*, *n*, *k*, *alpha*, *A*, *lda*, *B*, *ldb*, *beta*, *C*, *ldc*)
Rank-2k operation on complex double precision symmetric matrix.

References

cublas<t>syr2k

skcuda.cublas.cublasZtrmm

skcuda.cublas.**cublasZtrmm**(*handle*, *side*, *uplo*, *trans*, *diag*, *m*, *n*, *alpha*, *A*, *lda*, *B*, *ldb*, *C*, *ldc*)
Matrix-matrix product for complex double precision triangular matrix.

References

cublas<t>trmm

skcuda.cublas.cublasZtrsm

skcuda.cublas.**cublasZtrsm**(*handle*, *side*, *uplo*, *transa*, *diag*, *m*, *n*, *alpha*, *A*, *lda*, *B*, *ldb*)
Solve complex double precision triangular system with multiple right-hand sides.

References

cublas<t>trsm

Single-Precision BLAS-like Extension Routines

cublasSdgmm	Multiplies a matrix with a diagonal matrix.
cublasSgeam	Matrix-matrix addition/transposition (single precision real).
cublasSgemmBatched	Matrix-matrix product for arrays of real single precision general matrices.
cublasCgemmBatched	Matrix-matrix product for arrays of complex single precision general matrices.
cublasStrsmBatched	This function solves an array of triangular linear systems with multiple right-hand-sides.
cublasSgetrfBatched	This function performs the LU factorization of an array of n x n matrices.
cublasCdgmm	Multiplies a matrix with a diagonal matrix.
cublasCgeam	Matrix-matrix addition/transposition (single precision complex).

skcuda.cublas.cublasSdgmm

skcuda.cublas.**cublasSdgmm**(*handle*, *side*, *m*, *n*, *A*, *lda*, *x*, *incx*, *C*, *ldc*)

Multiplies a matrix with a diagonal matrix.

References

cublas<t>dgmm

skcuda.cublas.cublasSgeam

skcuda.cublas.**cublasSgeam**(*handle*, *transa*, *transb*, *m*, *n*, *alpha*, *A*, *lda*, *beta*, *B*, *ldb*, *C*, *ldc*)

Matrix-matrix addition/transposition (single precision real).

Computes the sum of two single precision real scaled and possibly (conjugate) transposed matrices.

Parameters

- **handle** (*int*) – CUBLAS context
- **transb** (*transa,*) – ‘t’ if they are transposed, ‘c’ if they are conjugate transposed, ‘n’ if otherwise.
- **m** (*int*) – Number of rows in *A* and *C*.
- **n** (*int*) – Number of columns in *B* and *C*.
- **alpha** (*numpy.float32*) – Constant by which to scale *A*.
- **A** (*ctypes.c_void_p*) – Pointer to first matrix operand (*A*).
- **lda** (*int*) – Leading dimension of *A*.
- **beta** (*numpy.float32*) – Constant by which to scale *B*.
- **B** (*ctypes.c_void_p*) – Pointer to second matrix operand (*B*).
- **ldb** (*int*) – Leading dimension of *A*.

- **C** (*ctypes.c_void_p*) – Pointer to result matrix (*C*).
- **ldc** (*int*) – Leading dimension of *C*.

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> alpha = np.float32(np.random.rand())
>>> beta = np.float32(np.random.rand())
>>> a = np.random.rand(2, 3).astype(np.float32)
>>> b = np.random.rand(2, 3).astype(np.float32)
>>> c = alpha*a+beta*b
>>> a_gpu = gpuarray.to_gpu(a)
>>> b_gpu = gpuarray.to_gpu(b)
>>> c_gpu = gpuarray.empty(c.shape, c.dtype)
>>> h = cublasCreate()
>>> cublasSgeam(h, 'n', 'n', c.shape[0], c.shape[1], alpha, a_gpu.gpudata, a.
↪shape[0], beta, b_gpu.gpudata, b.shape[0], c_gpu.gpudata, c.shape[0])
>>> np.allclose(c_gpu.get(), c)
True
>>> a = np.random.rand(2, 3).astype(np.float32)
>>> b = np.random.rand(3, 2).astype(np.float32)
>>> c = alpha*a.T+beta*b
>>> a_gpu = gpuarray.to_gpu(a.T.copy())
>>> b_gpu = gpuarray.to_gpu(b.T.copy())
>>> c_gpu = gpuarray.empty(c.T.shape, c.dtype)
>>> transa = 'c' if np.iscomplexobj(a) else 't'
>>> cublasSgeam(h, transa, 'n', c.shape[0], c.shape[1], alpha, a_gpu.gpudata, a.
↪shape[0], beta, b_gpu.gpudata, b.shape[0], c_gpu.gpudata, c.shape[0])
>>> np.allclose(c_gpu.get().T, c)
True
>>> cublasDestroy(h)
```

References

cublas<t>geam

skcuda.cublas.cublasSgemmBatched

skcuda.cublas.**cublasSgemmBatched**(*handle*, *transa*, *transb*, *m*, *n*, *k*, *alpha*, *A*, *lda*, *B*, *ldb*, *beta*, *C*, *ldc*, *batchCount*)

Matrix-matrix product for arrays of real single precision general matrices.

References

cublas<t>gemmBatched

skcuda.cublas.cublasCgemmBatched

skcuda.cublas.**cublasCgemmBatched**(*handle*, *transa*, *transb*, *m*, *n*, *k*, *alpha*, *A*, *lda*, *B*, *ldb*, *beta*, *C*, *ldc*, *batchCount*)

Matrix-matrix product for arrays of complex single precision general matrices.

References

cublas<t>gemmBatched

skcuda.cublas.cublasStrsmBatched

skcuda.cublas.**cublasStrsmBatched**(*handle*, *side*, *uplo*, *trans*, *diag*, *m*, *n*, *alpha*, *A*, *lda*, *B*, *ldb*, *batchCount*)

This function solves an array of triangular linear systems with multiple right-hand-sides.

References

cublas<t>trsmBatched

skcuda.cublas.cublasSgetrfBatched

skcuda.cublas.**cublasSgetrfBatched**(*handle*, *n*, *A*, *lda*, *P*, *info*, *batchSize*)

This function performs the LU factorization of an array of n x n matrices.

References

cublas<t>getrfBatched

skcuda.cublas.cublasCdgmm

skcuda.cublas.**cublasCdgmm**(*handle*, *side*, *m*, *n*, *A*, *lda*, *x*, *incx*, *C*, *ldc*)

Multiplies a matrix with a diagonal matrix.

References

cublas<t>dgmm

skcuda.cublas.cublasCgeam

skcuda.cublas.**cublasCgeam**(*handle*, *transa*, *transb*, *m*, *n*, *alpha*, *A*, *lda*, *beta*, *B*, *ldb*, *C*, *ldc*)

Matrix-matrix addition/transposition (single precision complex).

Computes the sum of two single precision complex scaled and possibly (conjugate) transposed matrices.

Parameters

- **handle** (*int*) – CUBLAS context
- **transb** (*transa,*) – 't' if they are transposed, 'c' if they are conjugate transposed, 'n' if otherwise.
- **m** (*int*) – Number of rows in *A* and *C*.
- **n** (*int*) – Number of columns in *B* and *C*.
- **alpha** (*numpy.complex64*) – Constant by which to scale *A*.
- **A** (*ctypes.c_void_p*) – Pointer to first matrix operand (*A*).
- **lda** (*int*) – Leading dimension of *A*.
- **beta** (*numpy.complex64*) – Constant by which to scale *B*.
- **B** (*ctypes.c_void_p*) – Pointer to second matrix operand (*B*).
- **ldb** (*int*) – Leading dimension of *A*.
- **C** (*ctypes.c_void_p*) – Pointer to result matrix (*C*).
- **ldc** (*int*) – Leading dimension of *C*.

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> alpha = np.complex64(np.random.rand()+1j*np.random.rand())
>>> beta = np.complex64(np.random.rand()+1j*np.random.rand())
>>> a = (np.random.rand(2, 3)+1j*np.random.rand(2, 3)).astype(np.complex64)
>>> b = (np.random.rand(2, 3)+1j*np.random.rand(2, 3)).astype(np.complex64)
>>> c = alpha*a+beta*b
>>> a_gpu = gpuarray.to_gpu(a)
>>> b_gpu = gpuarray.to_gpu(b)
>>> c_gpu = gpuarray.empty(c.shape, c.dtype)
>>> h = cublasCreate()
>>> cublasCgeam(h, 'n', 'n', c.shape[0], c.shape[1], alpha, a_gpu.gpudata, a.
↪shape[0], beta, b_gpu.gpudata, b.shape[0], c_gpu.gpudata, c.shape[0])
>>> np.allclose(c_gpu.get(), c)
True
>>> a = (np.random.rand(2, 3)+1j*np.random.rand(2, 3)).astype(np.complex64)
>>> b = (np.random.rand(3, 2)+1j*np.random.rand(3, 2)).astype(np.complex64)
>>> c = alpha*np.conj(a).T+beta*b
>>> a_gpu = gpuarray.to_gpu(a.T.copy())
>>> b_gpu = gpuarray.to_gpu(b.T.copy())
>>> c_gpu = gpuarray.empty(c.T.shape, c.dtype)
>>> transa = 'c' if np.iscomplexobj(a) else 't'
>>> cublasCgeam(h, transa, 'n', c.shape[0], c.shape[1], alpha, a_gpu.gpudata, a.
↪shape[0], beta, b_gpu.gpudata, b.shape[0], c_gpu.gpudata, c.shape[0])
>>> np.allclose(c_gpu.get().T, c)
True
>>> cublasDestroy(h)
```

References

cublas<t>geam

Double-Precision BLAS-like Extension Routines

cublasDdgmm	Multiplies a matrix with a diagonal matrix.
cublasDgeam	Matrix-matrix addition/transposition (double precision real).
cublasDgemmBatched	Matrix-matrix product for arrays of real double precision general matrices.
cublasZgemmBatched	Matrix-matrix product for arrays of complex double precision general matrices.
cublasDtrsmBatched	This function solves an array of triangular linear systems with multiple right-hand-sides.
cublasDgetrfBatched	This function performs the LU factorization of an array of n x n matrices.
cublasZdgmm	Multiplies a matrix with a diagonal matrix.
cublasZgeam	Matrix-matrix addition/transposition (double precision complex).

skcuda.cublas.cublasDdgmm

skcuda.cublas.**cublasDdgmm**(*handle*, *side*, *m*, *n*, *A*, *lda*, *x*, *incx*, *C*, *ldc*)

Multiplies a matrix with a diagonal matrix.

References

cublas<t>dgmm

skcuda.cublas.cublasDgeam

skcuda.cublas.**cublasDgeam**(*handle*, *transa*, *transb*, *m*, *n*, *alpha*, *A*, *lda*, *beta*, *B*, *ldb*, *C*, *ldc*)

Matrix-matrix addition/transposition (double precision real).

Computes the sum of two double precision real scaled and possibly (conjugate) transposed matrices.

Parameters

- **handle** (*int*) – CUBLAS context
- **transb** (*transa,*) – ‘t’ if they are transposed, ‘c’ if they are conjugate transposed, ‘n’ if otherwise.
- **m** (*int*) – Number of rows in *A* and *C*.
- **n** (*int*) – Number of columns in *B* and *C*.
- **alpha** (*numpy.float64*) – Constant by which to scale *A*.
- **A** (*ctypes.c_void_p*) – Pointer to first matrix operand (*A*).
- **lda** (*int*) – Leading dimension of *A*.
- **beta** (*numpy.float64*) – Constant by which to scale *B*.
- **B** (*ctypes.c_void_p*) – Pointer to second matrix operand (*B*).
- **ldb** (*int*) – Leading dimension of *A*.

- **C** (*ctypes.c_void_p*) – Pointer to result matrix (*C*).
- **ldc** (*int*) – Leading dimension of *C*.

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> alpha = np.float64(np.random.rand())
>>> beta = np.float64(np.random.rand())
>>> a = np.random.rand(2, 3).astype(np.float64)
>>> b = np.random.rand(2, 3).astype(np.float64)
>>> c = alpha*a+beta*b
>>> a_gpu = gpuarray.to_gpu(a)
>>> b_gpu = gpuarray.to_gpu(b)
>>> c_gpu = gpuarray.empty(c.shape, c.dtype)
>>> h = cublasCreate()
>>> cublasDgeam(h, 'n', 'n', c.shape[0], c.shape[1], alpha, a_gpu.gpudata, a.
↪shape[0], beta, b_gpu.gpudata, b.shape[0], c_gpu.gpudata, c.shape[0])
>>> np.allclose(c_gpu.get(), c)
True
>>> a = np.random.rand(2, 3).astype(np.float64)
>>> b = np.random.rand(3, 2).astype(np.float64)
>>> c = alpha*a.T+beta*b
>>> a_gpu = gpuarray.to_gpu(a.T.copy())
>>> b_gpu = gpuarray.to_gpu(b.T.copy())
>>> c_gpu = gpuarray.empty(c.T.shape, c.dtype)
>>> transa = 'c' if np.iscomplexobj(a) else 't'
>>> cublasDgeam(h, transa, 'n', c.shape[0], c.shape[1], alpha, a_gpu.gpudata, a.
↪shape[0], beta, b_gpu.gpudata, b.shape[0], c_gpu.gpudata, c.shape[0])
>>> np.allclose(c_gpu.get().T, c)
True
>>> cublasDestroy(h)
```

References

cublas<t>geam

skcuda.cublas.cublasDgemmBatched

skcuda.cublas.**cublasDgemmBatched**(*handle*, *transa*, *transb*, *m*, *n*, *k*, *alpha*, *A*, *lda*, *B*, *ldb*, *beta*, *C*, *ldc*, *batchCount*)

Matrix-matrix product for arrays of real double precision general matrices.

References

cublas<t>gemmBatched

skcuda.cublas.cublasZgemmBatched

skcuda.cublas.**cublasZgemmBatched**(*handle*, *transa*, *transb*, *m*, *n*, *k*, *alpha*, *A*, *lda*, *B*, *ldb*, *beta*, *C*, *ldc*, *batchCount*)

Matrix-matrix product for arrays of complex double precision general matrices.

References

cublas<t>gemmBatched

skcuda.cublas.cublasDtrsmBatched

skcuda.cublas.**cublasDtrsmBatched**(*handle*, *side*, *uplo*, *trans*, *diag*, *m*, *n*, *alpha*, *A*, *lda*, *B*, *ldb*, *batchCount*)

This function solves an array of triangular linear systems with multiple right-hand-sides.

References

cublas<t>trsmBatched

skcuda.cublas.cublasDgetrfBatched

skcuda.cublas.**cublasDgetrfBatched**(*handle*, *n*, *A*, *lda*, *P*, *info*, *batchSize*)

This function performs the LU factorization of an array of n x n matrices.

References

cublas<t>getrfBatched

skcuda.cublas.cublasZdgmm

skcuda.cublas.**cublasZdgmm**(*handle*, *side*, *m*, *n*, *A*, *lda*, *x*, *incx*, *C*, *ldc*)

Multiplies a matrix with a diagonal matrix.

References

cublas<t>dgmm

skcuda.cublas.cublasZgeam

skcuda.cublas.**cublasZgeam**(*handle*, *transa*, *transb*, *m*, *n*, *alpha*, *A*, *lda*, *beta*, *B*, *ldb*, *C*, *ldc*)

Matrix-matrix addition/transposition (double precision complex).

Computes the sum of two double precision complex scaled and possibly (conjugate) transposed matrices.

Parameters

- **handle** (*int*) – CUBLAS context
- **transb** (*transa,*) – 't' if they are transposed, 'c' if they are conjugate transposed, 'n' if otherwise.
- **m** (*int*) – Number of rows in *A* and *C*.
- **n** (*int*) – Number of columns in *B* and *C*.
- **alpha** (*numpy.complex128*) – Constant by which to scale *A*.
- **A** (*ctypes.c_void_p*) – Pointer to first matrix operand (*A*).
- **lda** (*int*) – Leading dimension of *A*.
- **beta** (*numpy.complex128*) – Constant by which to scale *B*.
- **B** (*ctypes.c_void_p*) – Pointer to second matrix operand (*B*).
- **ldb** (*int*) – Leading dimension of *A*.
- **C** (*ctypes.c_void_p*) – Pointer to result matrix (*C*).
- **ldc** (*int*) – Leading dimension of *C*.

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> alpha = np.complex128(np.random.rand()+1j*np.random.rand())
>>> beta = np.complex128(np.random.rand()+1j*np.random.rand())
>>> a = (np.random.rand(2, 3)+1j*np.random.rand(2, 3)).astype(np.complex128)
>>> b = (np.random.rand(2, 3)+1j*np.random.rand(2, 3)).astype(np.complex128)
>>> c = alpha*a+beta*b
>>> a_gpu = gpuarray.to_gpu(a)
>>> b_gpu = gpuarray.to_gpu(b)
>>> c_gpu = gpuarray.empty(c.shape, c.dtype)
>>> h = cublasCreate()
>>> cublasZgeam(h, 'n', 'n', c.shape[0], c.shape[1], alpha, a_gpu.gpudata, a.
↪shape[0], beta, b_gpu.gpudata, b.shape[0], c_gpu.gpudata, c.shape[0])
>>> np.allclose(c_gpu.get(), c)
True
>>> a = (np.random.rand(2, 3)+1j*np.random.rand(2, 3)).astype(np.complex128)
>>> b = (np.random.rand(3, 2)+1j*np.random.rand(3, 2)).astype(np.complex128)
>>> c = alpha*np.conj(a).T+beta*b
>>> a_gpu = gpuarray.to_gpu(a.T.copy())
>>> b_gpu = gpuarray.to_gpu(b.T.copy())
>>> c_gpu = gpuarray.empty(c.T.shape, c.dtype)
>>> transa = 'c' if np.iscomplexobj(a) else 't'
>>> cublasZgeam(h, transa, 'n', c.shape[0], c.shape[1], alpha, a_gpu.gpudata, a.
↪shape[0], beta, b_gpu.gpudata, b.shape[0], c_gpu.gpudata, c.shape[0])
>>> np.allclose(c_gpu.get().T, c)
True
>>> cublasDestroy(h)
```

References

cublas<t>geam

CUFFT Routines

Helper Routines

cufftCheckStatus
cufftCreate
cufftDestroy
cufftSetAutoAllocation
cufftSetCompatibilityMode
cufftSetStream
cufftSetWorkArea

Wrapper Routines

cufftPlan1d
cufftPlan2d
cufftPlan3d
cufftPlanMany
cufftDestroy
cufftExecC2C
cufftExecR2C
cufftExecC2R
cufftExecZ2Z
cufftExecD2Z
cufftExecZ2D
cufftEstimate1d
cufftEstimate2d
cufftEstimate3d
cufftEstimateMany
cufftGetSize1d
cufftGetSize2d
cufftGetSize3d
cufftGetSizeMany
cufftGetSize
cufftMakePlan1d
cufftMakePlan2d
cufftMakePlan3d
cufftMakePlanMany

CUSOLVER Routines

These routines are only available in CUDA 7.0 and later.

Helper Routines

cusolverDnCreate	Create cuSolverDn context.
cusolverDnCreateSyevjInfo	

Continued on next page

Table 12 – continued from previous page

cusolverDnGetStream	Get stream used by cuSolverDN library.
cusolverDnDestroy	Destroy cuSolverDn context.
cusolverDnDestroySyevjInfo	
cusolverDnSetStream	Set stream used by cuSolverDN library.
cusolverDnXsyevjGetResidual	
cusolverDnXsyevjGetSweeps	
cusolverDnXsyevjSetMaxSweeps	
cusolverDnXsyevjSetSortEig	
cusolverDnXsyevjSetTolerance	

skcuda.cusolver.cusolverDnCreate

skcuda.cusolver.**cusolverDnCreate**()
Create cuSolverDn context.

Returns **handle** – cuSolverDn context.

Return type int

References

cusolverDnCreate

skcuda.cusolver.cusolverDnCreateSyevjInfo

skcuda.cusolver.**cusolverDnCreateSyevjInfo**()

skcuda.cusolver.cusolverDnGetStream

skcuda.cusolver.**cusolverDnGetStream**(*handle*)
Get stream used by cuSolverDN library.

Parameters **handle** (*int*) – cuSolverDN context.

Returns **stream** – Stream used by context.

Return type int

References

cusolverDnGetStream

skcuda.cusolver.cusolverDnDestroy

skcuda.cusolver.**cusolverDnDestroy**(*handle*)
Destroy cuSolverDn context.

Parameters **handle** (*int*) – cuSolverDn context.

References

cusolverDnDestroy

skcuda.cusolver.cusolverDnDestroySyevjInfo

skcuda.cusolver.**cusolverDnDestroySyevjInfo**(*info*)

skcuda.cusolver.cusolverDnSetStream

skcuda.cusolver.**cusolverDnSetStream**(*handle*, *stream*)

Set stream used by cuSolverDN library.

Parameters

- **handle** (*int*) – cuSolverDN context.
- **stream** (*int*) – Stream to be used.

References

cusolverDnSetStream

skcuda.cusolver.cusolverDnXsyevjGetResidual

skcuda.cusolver.**cusolverDnXsyevjGetResidual**(*handle*, *info*)

skcuda.cusolver.cusolverDnXsyevjGetSweeps

skcuda.cusolver.**cusolverDnXsyevjGetSweeps**(*handle*, *info*)

skcuda.cusolver.cusolverDnXsyevjSetMaxSweeps

skcuda.cusolver.**cusolverDnXsyevjSetMaxSweeps**(*info*, *max_sweeps*)

skcuda.cusolver.cusolverDnXsyevjSetSortEig

skcuda.cusolver.**cusolverDnXsyevjSetSortEig**(*info*, *sort_eig*)

skcuda.cusolver.cusolverDnXsyevjSetTolerance

skcuda.cusolver.**cusolverDnXsyevjSetTolerance**(*info*, *tolerance*)

Wrapper Routines

Single Precision Routines

cusolverDnSgeqrf_bufferSize	Calculate size of work buffer used by cusolverDnSgeqrf.
cusolverDnSgeqrf	Compute QR factorization of a real single precision m x n matrix.
cusolverDnSgesvd_bufferSize	Calculate size of work buffer used by cusolverDnSgesvd.
cusolverDnSgesvd	Compute real single precision singular value decomposition.
cusolverDnSgetrf_bufferSize	Calculate size of work buffer used by cusolverDnSgetrf.
cusolverDnSgetrf	Compute LU factorization of a real single precision m x n matrix.
cusolverDnSgetrs	Solve real single precision linear system.
cusolverDnSorgqr_bufferSize	Calculate size of work buffer used by cusolverDnSorgqr.
cusolverDnSorgqr	Create unitary m x n matrix from single precision real reflection vectors.
cusolverDnSpotrf_bufferSize	Calculate size of work buffer used by cusolverDnSpotrf.
cusolverDnSpotrf	Compute Cholesky factorization of a real single precision Hermitian positive-definite matrix.
cusolverDnSsyevd_bufferSize	Calculate size of work buffer used by culsolverDnSsyevd.
cusolverDnSsyevd	
cusolverDnSsyevj_bufferSize	
cusolverDnSsyevj	
cusolverDnSsyevjBatched_bufferSize	
cusolverDnSsyevjBatched	
cusolverDnCgeqrf_bufferSize	Calculate size of work buffer used by cusolverDnCgeqrf.
cusolverDnCgeqrf	Compute QR factorization of a complex single precision m x n matrix.
cusolverDnCgesvd_bufferSize	Calculate size of work buffer used by cusolverDnCgesvd.
cusolverDnCgesvd	Compute complex single precision singular value decomposition.
cusolverDnCgetrf_bufferSize	Calculate size of work buffer used by cusolverDnCgetrf.
cusolverDnCgetrf	Compute LU factorization of a complex single precision m x n matrix.
cusolverDnCgetrs	Solve complex single precision linear system.
cusolverDnCheevd_bufferSize	Calculate size of work buffer used by culsolverDnCheevd.
cusolverDnCheevd	
cusolverDnCheevj_bufferSize	
cusolverDnCheevj	

Continued on next page

Table 13 – continued from previous page

cusolverDnCheevjBatched_bufferSize	
cusolverDnCheevjBatched	
cusolverDnCpotrf_bufferSize	Calculate size of work buffer used by cusolverDnCpotrf.
cusolverDnCpotrf	Compute Cholesky factorization of a complex single precision Hermitian positive-definite matrix.
cusolverDnCungqr_bufferSize	Calculate size of work buffer used by cusolverDnCungqr.
cusolverDnCungqr	Create unitary m x n matrix from single precision complex reflection vectors.

skcuda.cusolver.cusolverDnSgeqrf_bufferSize

skcuda.cusolver.**cusolverDnSgeqrf_bufferSize**(*handle*, *m*, *n*, *a*, *lda*)

Calculate size of work buffer used by cusolverDnSgeqrf.

References

cusolverDn<t>geqrf

skcuda.cusolver.cusolverDnSgeqrf

skcuda.cusolver.**cusolverDnSgeqrf**(*handle*, *m*, *n*, *a*, *lda*, *tau*, *workspace*, *lwork*, *devInfo*)

Compute QR factorization of a real single precision m x n matrix.

References

cusolverDn<t>geqrf

skcuda.cusolver.cusolverDnSgesvd_bufferSize

skcuda.cusolver.**cusolverDnSgesvd_bufferSize**(*handle*, *m*, *n*)

Calculate size of work buffer used by cusolverDnSgesvd.

References

cusolverDn<t>gesvd

skcuda.cusolver.cusolverDnSgesvd

skcuda.cusolver.**cusolverDnSgesvd**(*handle*, *jobu*, *jobvt*, *m*, *n*, *a*, *lda*, *s*, *U*, *ldu*, *vt*, *ldvt*, *work*, *lwork*, *rwork*, *devInfo*)

Compute real single precision singular value decomposition.

References

cusolverDn<t>gesvd

skcuda.cusolver.cusolverDnSgetrf_bufferSize

skcuda.cusolver.**cusolverDnSgetrf_bufferSize**(*handle*, *m*, *n*, *a*, *lda*)
Calculate size of work buffer used by cusolverDnSgetrf.

References

cusolverDn<t>getrf

skcuda.cusolver.cusolverDnSgetrf

skcuda.cusolver.**cusolverDnSgetrf**(*handle*, *m*, *n*, *a*, *lda*, *workspace*, *devIpiv*, *devInfo*)
Compute LU factorization of a real single precision m x n matrix.

References

cusolverDn<t>getrf

skcuda.cusolver.cusolverDnSgetrs

skcuda.cusolver.**cusolverDnSgetrs**(*handle*, *trans*, *n*, *nrhs*, *a*, *lda*, *devIpiv*, *B*, *ldb*, *devInfo*)
Solve real single precision linear system.

References

cusolverDn<t>getrs

skcuda.cusolver.cusolverDnSorgqr_bufferSize

skcuda.cusolver.**cusolverDnSorgqr_bufferSize**(*handle*, *m*, *n*, *k*, *a*, *lda*, *tau*)
Calculate size of work buffer used by cusolverDnSorgqr.

References

cusolverDn<t>orgqr

skcuda.cusolver.cusolverDnSorgqr

skcuda.cusolver.**cusolverDnSorgqr**(*handle*, *m*, *n*, *k*, *a*, *lda*, *tau*, *work*, *lwork*, *devInfo*)
Create unitary m x n matrix from single precision real reflection vectors.

References

cusolverDn<t>orgqr

skcuda.cusolver.cusolverDnSpotrf_bufferSize

skcuda.cusolver.**cusolverDnSpotrf_bufferSize**(*handle*, *uplo*, *n*, *a*, *lda*)
Calculate size of work buffer used by cusolverDnSpotrf.

References

cusolverDn<t>potrf

skcuda.cusolver.cusolverDnSpotrf

skcuda.cusolver.**cusolverDnSpotrf**(*handle*, *uplo*, *n*, *a*, *lda*, *workspace*, *devIpiv*, *devInfo*)
Compute Cholesky factorization of a real single precision Hermitian positive-definite matrix.

References

cusolverDn<t>potrf

skcuda.cusolver.cusolverDnSsyevd_bufferSize

skcuda.cusolver.**cusolverDnSsyevd_bufferSize**(*handle*, *jobz*, *uplo*, *n*, *a*, *lda*, *w*)
Calculate size of work buffer used by culsolverDnSsyevd.

References

cusolverDn<t>gebrd

skcuda.cusolver.cusolverDnSsyevd

skcuda.cusolver.**cusolverDnSsyevd**(*handle*, *jobz*, *uplo*, *n*, *a*, *lda*, *w*, *workspace*, *lwork*, *devInfo*)

skcuda.cusolver.cusolverDnSsyevj_bufferSize

skcuda.cusolver.**cusolverDnSsyevj_bufferSize**(*handle*, *jobz*, *uplo*, *n*, *a*, *lda*, *w*, *params*)

skcuda.cusolver.cusolverDnSsyevj

skcuda.cusolver.**cusolverDnSsyevj**(*handle*, *jobz*, *uplo*, *n*, *a*, *lda*, *w*, *work*, *lwork*, *info*, *params*)

skcuda.cusolver.cusolverDnSsyevjBatched_bufferSize

skcuda.cusolver.**cusolverDnSsyevjBatched_bufferSize**(*handle*, *jobz*, *uplo*, *n*, *a*, *lda*, *w*, *params*, *batchSize*)

skcuda.cusolver.cusolverDnSsyevjBatched

skcuda.cusolver.**cusolverDnSsyevjBatched**(*handle*, *jobz*, *uplo*, *n*, *a*, *lda*, *w*, *work*, *lwork*, *params*, *batchSize*)

skcuda.cusolver.cusolverDnCgeqrf_bufferSize

skcuda.cusolver.**cusolverDnCgeqrf_bufferSize**(*handle*, *m*, *n*, *a*, *lda*)

Calculate size of work buffer used by cusolverDnCgeqrf.

References

cusolverDn<t>geqrf

skcuda.cusolver.cusolverDnCgeqrf

skcuda.cusolver.**cusolverDnCgeqrf**(*handle*, *m*, *n*, *a*, *lda*, *tau*, *workspace*, *lwork*, *devInfo*)

Compute QR factorization of a complex single precision m x n matrix.

References

cusolverDn<t>geqrf

skcuda.cusolver.cusolverDnCgesvd_bufferSize

skcuda.cusolver.**cusolverDnCgesvd_bufferSize**(*handle*, *m*, *n*)

Calculate size of work buffer used by cusolverDnCgesvd.

References

cusolverDn<t>gesvd

skcuda.cusolver.cusolverDnCgesvd

skcuda.cusolver.**cusolverDnCgesvd**(*handle*, *jobu*, *jobvt*, *m*, *n*, *a*, *lda*, *s*, *U*, *ldu*, *vt*, *ldvt*, *work*, *lwork*, *rwork*, *devInfo*)

Compute complex single precision singular value decomposition.

References

cusolverDn<t>gesvd

skcuda.cusolver.cusolverDnCgetrf_bufferSize

skcuda.cusolver.**cusolverDnCgetrf_bufferSize**(*handle*, *m*, *n*, *a*, *lda*)
Calculate size of work buffer used by cusolverDnCgetrf.

References

cusolverDn<t>getrf

skcuda.cusolver.cusolverDnCgetrf

skcuda.cusolver.**cusolverDnCgetrf**(*handle*, *m*, *n*, *a*, *lda*, *workspace*, *devIpiv*, *devInfo*)
Compute LU factorization of a complex single precision m x n matrix.

References

cusolverDn<t>getrf

skcuda.cusolver.cusolverDnCgetrs

skcuda.cusolver.**cusolverDnCgetrs**(*handle*, *trans*, *n*, *nrhs*, *a*, *lda*, *devIpiv*, *B*, *ldb*, *devInfo*)
Solve complex single precision linear system.

References

cusolverDn<t>getrs

skcuda.cusolver.cusolverDnCheevd_bufferSize

skcuda.cusolver.**cusolverDnCheevd_bufferSize**(*handle*, *jobz*, *uplo*, *n*, *a*, *lda*, *w*)
Calculate size of work buffer used by culsolverDnCheevd.

References

cusolverDn<t>gebrd

skcuda.cusolver.cusolverDnCheevd

skcuda.cusolver.**cusolverDnCheevd**(*handle*, *jobz*, *uplo*, *n*, *a*, *lda*, *w*, *workspace*, *lwork*, *devInfo*)

skcuda.cusolver.cusolverDnCheevj_bufferSize

skcuda.cusolver.**cusolverDnCheevj_bufferSize**(*handle*, *jobz*, *uplo*, *n*, *a*, *lda*, *w*, *params*)

skcuda.cusolver.cusolverDnCheevj

skcuda.cusolver.**cusolverDnCheevj**(*handle*, *jobz*, *uplo*, *n*, *a*, *lda*, *w*, *work*, *lwork*, *info*, *params*)

skcuda.cusolver.cusolverDnCheevjBatched_bufferSize

skcuda.cusolver.**cusolverDnCheevjBatched_bufferSize**(*handle*, *jobz*, *uplo*, *n*, *a*, *lda*, *w*, *params*, *batchSize*)

skcuda.cusolver.cusolverDnCheevjBatched

skcuda.cusolver.**cusolverDnCheevjBatched**(*handle*, *jobz*, *uplo*, *n*, *a*, *lda*, *w*, *work*, *lwork*, *info*, *params*, *batchSize*)

skcuda.cusolver.cusolverDnCpotrf_bufferSize

skcuda.cusolver.**cusolverDnCpotrf_bufferSize**(*handle*, *uplo*, *n*, *a*, *lda*)

Calculate size of work buffer used by cusolverDnCpotrf.

References

cusolverDn<t>potrf

skcuda.cusolver.cusolverDnCpotrf

skcuda.cusolver.**cusolverDnCpotrf**(*handle*, *uplo*, *n*, *a*, *lda*, *workspace*, *devIpiv*, *devInfo*)

Compute Cholesky factorization of a complex single precision Hermitian positive-definite matrix.

References

cusolverDn<t>potrf

skcuda.cusolver.cusolverDnCungqr_bufferSize

skcuda.cusolver.**cusolverDnCungqr_bufferSize**(*handle*, *m*, *n*, *k*, *a*, *lda*, *tau*)

Calculate size of work buffer used by cusolverDnCungqr.

References

cusolverDn<t>orgqr

skcuda.cusolver.cusolverDnCungqr

skcuda.cusolver.**cusolverDnCungqr**(*handle*, *m*, *n*, *k*, *a*, *lda*, *tau*, *work*, *lwork*, *devInfo*)
Create unitary m x n matrix from single precision complex reflection vectors.

References

cusolverDn<t>orgqr

Double Precision Routines

cusolverDnDgeqrf_bufferSize	Calculate size of work buffer used by cusolverDnDgeqrf.
cusolverDnDgeqrf	Compute QR factorization of a real double precision m x n matrix.
cusolverDnDgesvd_bufferSize	Calculate size of work buffer used by cusolverDnDgesvd.
cusolverDnDgesvd	Compute real double precision singular value decomposition.
cusolverDnDgetrf_bufferSize	Calculate size of work buffer used by cusolverDnDgetrf.
cusolverDnDgetrf	Compute LU factorization of a real double precision m x n matrix.
cusolverDnDgetrs	Solve real double precision linear system.
cusolverDnDorgqr_bufferSize	Calculate size of work buffer used by cusolverDnDorgqr.
cusolverDnDorgqr	Create unitary m x n matrix from double precision real reflection vectors.
cusolverDnDpotrf_bufferSize	Calculate size of work buffer used by cusolverDnDpotrf.
cusolverDnDpotrf	Compute Cholesky factorization of a real double precision Hermitian positive-definite matrix.
cusolverDnDsyevd_bufferSize	Calculate size of work buffer used by culsolverDnDsyevd.
cusolverDnDsyevd	
cusolverDnDsyevj_bufferSize	
cusolverDnDsyevj	
cusolverDnDsyevjBatched_bufferSize	
cusolverDnDsyevjBatched	
cusolverDnZgeqrf_bufferSize	Calculate size of work buffer used by cusolverDnZgeqrf.
cusolverDnZgeqrf	Compute QR factorization of a complex double precision m x n matrix.
cusolverDnZgesvd_bufferSize	Calculate size of work buffer used by cusolverDnZgesvd.
cusolverDnZgesvd	Compute complex double precision singular value decomposition.
cusolverDnZgetrf_bufferSize	Calculate size of work buffer used by cusolverDnZgetrf.

Continued on next page

Table 14 – continued from previous page

cusolverDnZgetrf	Compute LU factorization of a complex double precision m x n matrix.
cusolverDnZgetrs	Solve complex double precision linear system.
cusolverDnZheevd_bufferSize	Calculate size of work buffer used by cusolverDnZheevd.
cusolverDnZheevd	
cusolverDnZheevj_bufferSize	
cusolverDnZheevj	
cusolverDnZheevjBatched_bufferSize	
cusolverDnZheevjBatched	
cusolverDnZpotrf_bufferSize	Calculate size of work buffer used by cusolverDnZpotrf.
cusolverDnZpotrf	Compute Cholesky factorization of a complex double precision Hermitian positive-definite matrix.
cusolverDnZungqr_bufferSize	Calculate size of work buffer used by cusolverDnZungqr.
cusolverDnZungqr	Create unitary m x n matrix from double precision complex reflection vectors.

skcuda.cusolver.cusolverDnDgeqrf_bufferSize

skcuda.cusolver.**cusolverDnDgeqrf_bufferSize**(*handle*, *m*, *n*, *a*, *lda*)

Calculate size of work buffer used by cusolverDnDgeqrf.

References

cusolverDn<t>geqrf

skcuda.cusolver.cusolverDnDgeqrf

skcuda.cusolver.**cusolverDnDgeqrf**(*handle*, *m*, *n*, *a*, *lda*, *tau*, *workspace*, *lwork*, *devInfo*)

Compute QR factorization of a real double precision m x n matrix.

References

cusolverDn<t>geqrf

skcuda.cusolver.cusolverDnDgesvd_bufferSize

skcuda.cusolver.**cusolverDnDgesvd_bufferSize**(*handle*, *m*, *n*)

Calculate size of work buffer used by cusolverDnDgesvd.

References

cusolverDn<t>gesvd

skcuda.cusolver.cusolverDnDgesvd

skcuda.cusolver.**cusolverDnDgesvd**(*handle*, *jobu*, *jobvt*, *m*, *n*, *a*, *lda*, *s*, *U*, *ldu*, *vt*, *ldvt*, *work*, *lwork*, *rwork*, *devInfo*)

Compute real double precision singular value decomposition.

References

cusolverDn<t>gesvd

skcuda.cusolver.cusolverDnDgetrf_bufferSize

skcuda.cusolver.**cusolverDnDgetrf_bufferSize**(*handle*, *m*, *n*, *a*, *lda*)

Calculate size of work buffer used by cusolverDnDgetrf.

References

cusolverDn<t>getrf

skcuda.cusolver.cusolverDnDgetrf

skcuda.cusolver.**cusolverDnDgetrf**(*handle*, *m*, *n*, *a*, *lda*, *workspace*, *devIpiv*, *devInfo*)

Compute LU factorization of a real double precision m x n matrix.

References

cusolverDn<t>getrf

skcuda.cusolver.cusolverDnDgetrs

skcuda.cusolver.**cusolverDnDgetrs**(*handle*, *trans*, *n*, *nrhs*, *a*, *lda*, *devIpiv*, *B*, *ldb*, *devInfo*)

Solve real double precision linear system.

References

cusolverDn<t>getrs

skcuda.cusolver.cusolverDnDorgqr_bufferSize

skcuda.cusolver.**cusolverDnDorgqr_bufferSize**(*handle*, *m*, *n*, *k*, *a*, *lda*, *tau*)

Calculate size of work buffer used by cusolverDnDorgqr.

References

cusolverDn<t>orgqr

skcuda.cusolver.cusolverDnDorgqr

skcuda.cusolver.**cusolverDnDorgqr**(*handle*, *m*, *n*, *k*, *a*, *lda*, *tau*, *work*, *lwork*, *devInfo*)
Create unitary m x n matrix from double precision real reflection vectors.

References

cusolverDn<t>orgqr

skcuda.cusolver.cusolverDnDpotrf_bufferSize

skcuda.cusolver.**cusolverDnDpotrf_bufferSize**(*handle*, *uplo*, *n*, *a*, *lda*)
Calculate size of work buffer used by cusolverDnDpotrf.

References

cusolverDn<t>potrf

skcuda.cusolver.cusolverDnDpotrf

skcuda.cusolver.**cusolverDnDpotrf**(*handle*, *uplo*, *n*, *a*, *lda*, *workspace*, *devIpiv*, *devInfo*)
Compute Cholesky factorization of a real double precision Hermitian positive-definite matrix.

References

cusolverDn<t>potrf

skcuda.cusolver.cusolverDnDsyevd_bufferSize

skcuda.cusolver.**cusolverDnDsyevd_bufferSize**(*handle*, *jobz*, *uplo*, *n*, *a*, *lda*, *w*)
Calculate size of work buffer used by culsolverDnDsyevd.

References

cusolverDn<t>gebrd

skcuda.cusolver.cusolverDnDsyevd

skcuda.cusolver.**cusolverDnDsyevd**(*handle*, *jobz*, *uplo*, *n*, *a*, *lda*, *w*, *workspace*, *lwork*, *devInfo*)

skcuda.cusolver.cusolverDnDsyevj_bufferSize

skcuda.cusolver.**cusolverDnDsyevj_bufferSize**(*handle*, *jobz*, *uplo*, *n*, *a*, *lda*, *w*, *params*)

skcuda.cusolver.cusolverDnDsyevj

skcuda.cusolver.**cusolverDnDsyevj**(*handle*, *jobz*, *uplo*, *n*, *a*, *lda*, *w*, *work*, *lwork*, *info*, *params*)

skcuda.cusolver.cusolverDnDsyevjBatched_bufferSize

skcuda.cusolver.**cusolverDnDsyevjBatched_bufferSize**(*handle*, *jobz*, *uplo*, *n*, *a*, *lda*, *w*, *params*, *batchSize*)

skcuda.cusolver.cusolverDnDsyevjBatched

skcuda.cusolver.**cusolverDnDsyevjBatched**(*handle*, *jobz*, *uplo*, *n*, *a*, *lda*, *w*, *work*, *lwork*, *info*, *params*, *batchSize*)

skcuda.cusolver.cusolverDnZgeqrf_bufferSize

skcuda.cusolver.**cusolverDnZgeqrf_bufferSize**(*handle*, *m*, *n*, *a*, *lda*)

Calculate size of work buffer used by cusolverDnZgeqrf.

References

cusolverDn<t>geqrf

skcuda.cusolver.cusolverDnZgeqrf

skcuda.cusolver.**cusolverDnZgeqrf**(*handle*, *m*, *n*, *a*, *lda*, *tau*, *workspace*, *lwork*, *devInfo*)

Compute QR factorization of a complex double precision m x n matrix.

References

cusolverDn<t>geqrf

skcuda.cusolver.cusolverDnZgesvd_bufferSize

skcuda.cusolver.**cusolverDnZgesvd_bufferSize**(*handle*, *m*, *n*)

Calculate size of work buffer used by cusolverDnZgesvd.

References

cusolverDn<t>gesvd

skcuda.cusolver.cusolverDnZgesvd

skcuda.cusolver.**cusolverDnZgesvd**(*handle*, *jobu*, *jobvt*, *m*, *n*, *a*, *lda*, *s*, *U*, *ldu*, *vt*, *ldvt*, *work*, *lwork*, *rwork*, *devInfo*)

Compute complex double precision singular value decomposition.

References

cusolverDn<t>gesvd

skcuda.cusolver.cusolverDnZgetrf_bufferSize

skcuda.cusolver.**cusolverDnZgetrf_bufferSize**(*handle*, *m*, *n*, *a*, *lda*)

Calculate size of work buffer used by cusolverDnZgetrf.

References

cusolverDn<t>getrf

skcuda.cusolver.cusolverDnZgetrf

skcuda.cusolver.**cusolverDnZgetrf**(*handle*, *m*, *n*, *a*, *lda*, *workspace*, *devIpiv*, *devInfo*)

Compute LU factorization of a complex double precision m x n matrix.

References

cusolverDn<t>getrf

skcuda.cusolver.cusolverDnZgetrs

skcuda.cusolver.**cusolverDnZgetrs**(*handle*, *trans*, *n*, *nrhs*, *a*, *lda*, *devIpiv*, *B*, *ldb*, *devInfo*)

Solve complex double precision linear system.

References

cusolverDn<t>getrs

skcuda.cusolver.cusolverDnZheevd_bufferSize

skcuda.cusolver.**cusolverDnZheevd_bufferSize**(*handle*, *jobz*, *uplo*, *n*, *a*, *lda*, *w*)

Calculate size of work buffer used by culsolverDnZheevd.

References

cusolverDn<t>gebrd

skcuda.cusolver.cusolverDnZheevd

skcuda.cusolver.**cusolverDnZheevd**(*handle*, *jobz*, *uplo*, *n*, *a*, *lda*, *w*, *workspace*, *lwork*, *devInfo*)

skcuda.cusolver.cusolverDnZheevj_bufferSize

skcuda.cusolver.**cusolverDnZheevj_bufferSize**(*handle*, *jobz*, *uplo*, *n*, *a*, *lda*, *w*, *params*)

skcuda.cusolver.cusolverDnZheevj

skcuda.cusolver.**cusolverDnZheevj**(*handle*, *jobz*, *uplo*, *n*, *a*, *lda*, *w*, *work*, *lwork*, *info*, *params*)

skcuda.cusolver.cusolverDnZheevjBatched_bufferSize

skcuda.cusolver.**cusolverDnZheevjBatched_bufferSize**(*handle*, *jobz*, *uplo*, *n*, *a*, *lda*, *w*, *params*, *batchSize*)

skcuda.cusolver.cusolverDnZheevjBatched

skcuda.cusolver.**cusolverDnZheevjBatched**(*handle*, *jobz*, *uplo*, *n*, *a*, *lda*, *w*, *work*, *lwork*, *info*, *params*, *batchSize*)

skcuda.cusolver.cusolverDnZpotrf_bufferSize

skcuda.cusolver.**cusolverDnZpotrf_bufferSize**(*handle*, *uplo*, *n*, *a*, *lda*)

Calculate size of work buffer used by cusolverDnZpotrf.

References

cusolverDn<t>potrf

skcuda.cusolver.cusolverDnZpotrf

skcuda.cusolver.**cusolverDnZpotrf**(*handle*, *uplo*, *n*, *a*, *lda*, *workspace*, *devIpiv*, *devInfo*)

Compute Cholesky factorization of a complex double precision Hermitian positive-definite matrix.

References

cusolverDn<t>potrf

skcuda.cusolver.cusolverDnZungqr_bufferSize

skcuda.cusolver.**cusolverDnZungqr_bufferSize**(*handle*, *m*, *n*, *k*, *a*, *lda*, *tau*)
Calculate size of work buffer used by cusolverDnZungqr.

References

cusolverDn<t>orgqr

skcuda.cusolver.cusolverDnZungqr

skcuda.cusolver.**cusolverDnZungqr**(*handle*, *m*, *n*, *k*, *a*, *lda*, *tau*, *work*, *lwork*, *devInfo*)
Create unitary m x n matrix from double precision complex reflection vectors.

References

cusolverDn<t>orgqr

CULA Routines

Framework Routines

culaCheckStatus	Raise an exception corresponding to the specified CULA status code.
culaFreeBuffers	Releases any memory buffers stored internally by CULA.
culaGetCublasMinimumVersion	Report the version of CUBLAS required by CULA.
culaGetCublasRuntimeVersion	Report the version of CUBLAS linked to by CULA.
culaGetCudaDriverVersion	Report the version of the CUDA driver installed on the system.
culaGetCudaMinimumVersion	Report the minimum version of CUDA required by CULA.
culaGetCudaRuntimeVersion	Report the version of the CUDA runtime linked to by the CULA library.
culaGetDeviceCount	Report the number of available GPU devices.
culaGetErrorInfo	Returns extended information code for the last CULA error.
culaGetErrorInfoString	Returns a readable CULA error string.
culaGetExecutingDevice	Reports the id of the GPU device used by CULA.
culaGetLastStatus	Returns the last status code returned from a CULA function.
culaGetStatusString	Get string associated with the specified CULA status code.
culaGetVersion	Report the version number of CULA.
culaInitialize	Initialize CULA.
culaSelectDevice	Selects a device with which CULA will operate.
culaShutdown	Shuts down CULA.

skcuda.cula.culaCheckStatus

skcuda.cula.**culaCheckStatus**(*status*)
Raise an exception corresponding to the specified CULA status code.

Parameters **status** (*int*) – CULA status code.

skcuda.cula.culaFreeBuffers

skcuda.cula.**culaFreeBuffers**()
Releases any memory buffers stored internally by CULA.

skcuda.cula.culaGetCublasMinimumVersion

skcuda.cula.**culaGetCublasMinimumVersion**()
Report the version of CUBLAS required by CULA.

skcuda.cula.culaGetCublasRuntimeVersion

skcuda.cula.**culaGetCublasRuntimeVersion**()
Report the version of CUBLAS linked to by CULA.

skcuda.cula.culaGetCudaDriverVersion

skcuda.cula.**culaGetCudaDriverVersion**()
Report the version of the CUDA driver installed on the system.

skcuda.cula.culaGetCudaMinimumVersion

skcuda.cula.**culaGetCudaMinimumVersion**()
Report the minimum version of CUDA required by CULA.

skcuda.cula.culaGetCudaRuntimeVersion

skcuda.cula.**culaGetCudaRuntimeVersion**()
Report the version of the CUDA runtime linked to by the CULA library.

skcuda.cula.culaGetDeviceCount

skcuda.cula.**culaGetDeviceCount**()
Report the number of available GPU devices.

skcuda.cula.culaGetErrorInfo

skcuda.cula.**culaGetErrorInfo**()
Returns extended information code for the last CULA error.

Returns **err** – Extended information code.

Return type int

skcuda.cula.culaGetErrorInfoString

skcuda.cula.**culaGetErrorInfoString**(*e*, *i*, *bufsize=100*)
Returns a readable CULA error string.

Returns a readable error string corresponding to a given CULA error code and extended error information code.

Parameters

- **e** (*int*) – CULA error code.
- **i** (*int*) – Extended information code.
- **bufsize** (*int*) – Length of string to return.

Returns **s** – Error string.

Return type str

skcuda.cula.culaGetExecutingDevice

skcuda.cula.**culaGetExecutingDevice**()
Reports the id of the GPU device used by CULA.

Returns **dev** – Device id.

Return type int

skcuda.cula.culaGetLastStatus

skcuda.cula.**culaGetLastStatus**()
Returns the last status code returned from a CULA function.

Returns **s** – Status code.

Return type int

skcuda.cula.culaGetStatusString

skcuda.cula.**culaGetStatusString**(*e*)
Get string associated with the specified CULA status code.

Parameters **e** (*int*) – Status code.

Returns **s** – Status string.

Return type str

skcuda.cula.culaGetVersion

skcuda.cula.**culaGetVersion**()
Report the version number of CULA.

skcuda.cula.culaInitialize

skcuda.cula.**culaInitialize**()
Initialize CULA.

Notes

Must be called before using any other CULA functions.

skcuda.cula.culaSelectDevice

skcuda.cula.**culaSelectDevice**(*dev*)
Selects a device with which CULA will operate.

Parameters **dev** (*int*) – GPU device number.

Notes

Must be called before *culaInitialize*.

skcuda.cula.culaShutdown

skcuda.cula.**culaShutdown**()
Shuts down CULA.

Auxiliary Routines

Single Precision Real

culaDeviceSgeNancheck	Check a real general matrix for invalid entries
culaDeviceSgeTranspose	Transpose of real general matrix.
culaDeviceSgeTransposeInplace	Inplace transpose of real square matrix.

skcuda.cula.culaDeviceSgeNancheck

skcuda.cula.**culaDeviceSgeNancheck**(*m*, *n*, *A*, *lda*)
Check a real general matrix for invalid entries

skcuda.cula.culaDeviceSgeTranspose

skcuda.cula.**culaDeviceSgeTranspose**(*m*, *n*, *A*, *lda*, *B*, *ldb*)
Transpose of real general matrix.

skcuda.cula.culaDeviceSgeTransposeInplace

skcuda.cula.**culaDeviceSgeTransposeInplace**(*n*, *A*, *lda*)
Inplace transpose of real square matrix.

Single Precision Complex

culaDeviceCgeConjugate	Conjugate of complex general matrix.
culaDeviceCgeNancheck	Check a complex general matrix for invalid entries
culaDeviceCgeTranspose	Transpose of complex general matrix.
culaDeviceCgeTransposeConjugate	Conjugate transpose of complex general matrix.
culaDeviceCgeTransposeInplace	Inplace transpose of complex square matrix.
culaDeviceCgeTransposeConjugateInplace	Inplace conjugate transpose of complex square matrix.

skcuda.cula.culaDeviceCgeConjugate

skcuda.cula.**culaDeviceCgeConjugate**(*m*, *n*, *A*, *lda*)
Conjugate of complex general matrix.

skcuda.cula.culaDeviceCgeNancheck

skcuda.cula.**culaDeviceCgeNancheck**(*m*, *n*, *A*, *lda*)
Check a complex general matrix for invalid entries

skcuda.cula.culaDeviceCgeTranspose

skcuda.cula.**culaDeviceCgeTranspose**(*m*, *n*, *A*, *lda*, *B*, *ldb*)
Transpose of complex general matrix.

skcuda.cula.culaDeviceCgeTransposeConjugate

skcuda.cula.**culaDeviceCgeTransposeConjugate**(*m*, *n*, *A*, *lda*, *B*, *ldb*)
Conjugate transpose of complex general matrix.

skcuda.cula.culaDeviceCgeTransposeInplace

skcuda.cula.**culaDeviceCgeTransposeInplace**(*n*, *A*, *lda*)
Inplace transpose of complex square matrix.

skcuda.cula.culaDeviceCgeTransposeConjugateInplace

skcuda.cula.**culaDeviceCgeTransposeConjugateInplace**(*n*, *A*, *lda*)
Inplace conjugate transpose of complex square matrix.

Double Precision Real

culaDeviceDgeNancheck	Check a real general matrix for invalid entries
culaDeviceDgeTranspose	Transpose of real general matrix.
culaDeviceDgeTransposeInplace	Inplace transpose of real square matrix.

skcuda.cula.culaDeviceDgeNancheck

skcuda.cula.**culaDeviceDgeNancheck**(*m*, *n*, *A*, *lda*)
Check a real general matrix for invalid entries

skcuda.cula.culaDeviceDgeTranspose

skcuda.cula.**culaDeviceDgeTranspose**(*m*, *n*, *A*, *lda*, *B*, *ldb*)
Transpose of real general matrix.

skcuda.cula.culaDeviceDgeTransposeInplace

skcuda.cula.**culaDeviceDgeTransposeInplace**(*n*, *A*, *lda*)
Inplace transpose of real square matrix.

Double Precision Complex

culaDeviceZgeConjugate	Conjugate of complex general matrix.
culaDeviceZgeNancheck	Check a complex general matrix for invalid entries
culaDeviceZgeTranspose	Transpose of complex general matrix.
culaDeviceZgeTransposeConjugate	Conjugate transpose of complex general matrix.
culaDeviceZgeTransposeInplace	Inplace transpose of complex square matrix.
culaDeviceZgeTransposeConjugateInplace	Inplace conjugate transpose of complex square matrix.

skcuda.cula.culaDeviceZgeConjugate

skcuda.cula.**culaDeviceZgeConjugate**(*m*, *n*, *A*, *lda*)
Conjugate of complex general matrix.

skcuda.cula.culaDeviceZgeNancheck

skcuda.cula.**culaDeviceZgeNancheck**(*m*, *n*, *A*, *lda*)
Check a complex general matrix for invalid entries

skcuda.cula.culaDeviceZgeTranspose

skcuda.cula.**culaDeviceZgeTranspose**(*m*, *n*, *A*, *lda*, *B*, *ldb*)
Transpose of complex general matrix.

skcuda.cula.culaDeviceZgeTransposeConjugate

skcuda.cula.**culaDeviceZgeTransposeConjugate**(*m*, *n*, *A*, *lda*, *B*, *ldb*)
Conjugate transpose of complex general matrix.

skcuda.cula.culaDeviceZgeTransposeInplace

skcuda.cula.**culaDeviceZgeTransposeInplace**(*n*, *A*, *lda*)
Inplace transpose of complex square matrix.

skcuda.cula.culaDeviceZgeTransposeConjugateInplace

skcuda.cula.**culaDeviceZgeTransposeConjugateInplace**(*n*, *A*, *lda*)
Inplace conjugate transpose of complex square matrix.

BLAS Routines

Single Precision Real

culaDeviceSgemm	Matrix-matrix product for general matrix.
culaDeviceSgemv	Matrix-vector product for real general matrix.

skcuda.cula.culaDeviceSgemm

skcuda.cula.**culaDeviceSgemm**(*transa*, *transb*, *m*, *n*, *k*, *alpha*, *A*, *lda*, *B*, *ldb*, *beta*, *C*, *ldc*)
Matrix-matrix product for general matrix.

skcuda.cula.culaDeviceSgemv

skcuda.cula.**culaDeviceSgemv**(*trans*, *m*, *n*, *alpha*, *A*, *lda*, *x*, *incx*, *beta*, *y*, *incy*)
Matrix-vector product for real general matrix.

Single Precision Complex

culaDeviceCgemm	Matrix-matrix product for complex general matrix.
culaDeviceCgemv	Matrix-vector product for complex general matrix.

skcuda.cula.culaDeviceCgemm

skcuda.cula.**culaDeviceCgemm**(*transa*, *transb*, *m*, *n*, *k*, *alpha*, *A*, *lda*, *B*, *ldb*, *beta*, *C*, *ldc*)
Matrix-matrix product for complex general matrix.

skcuda.cula.culaDeviceCgemv

skcuda.cula.**culaDeviceCgemv**(*trans*, *m*, *n*, *alpha*, *A*, *lda*, *x*, *incx*, *beta*, *y*, *incy*)
Matrix-vector product for complex general matrix.

Double Precision Real

culaDeviceDgemm	Matrix-matrix product for general matrix.
culaDeviceDgemv	Matrix-vector product for real general matrix.

skcuda.cula.culaDeviceDgemm

skcuda.cula.**culaDeviceDgemm**(*transa*, *transb*, *m*, *n*, *k*, *alpha*, *A*, *lda*, *B*, *ldb*, *beta*, *C*, *ldc*)
Matrix-matrix product for general matrix.

skcuda.cula.culaDeviceDgemv

skcuda.cula.**culaDeviceDgemv**(*trans*, *m*, *n*, *alpha*, *A*, *lda*, *x*, *incx*, *beta*, *y*, *incy*)
Matrix-vector product for real general matrix.

Double Precision Complex

culaDeviceZgemm	Matrix-matrix product for complex general matrix.
culaDeviceZgemv	Matrix-vector product for complex general matrix.

skcuda.cula.culaDeviceZgemm

skcuda.cula.**culaDeviceZgemm**(*transa*, *transb*, *m*, *n*, *k*, *alpha*, *A*, *lda*, *B*, *ldb*, *beta*, *C*, *ldc*)
Matrix-matrix product for complex general matrix.

skcuda.cula.culaDeviceZgemv

skcuda.cula.**culaDeviceZgemv**(*trans*, *m*, *n*, *alpha*, *A*, *lda*, *x*, *incx*, *beta*, *y*, *incy*)
Matrix-vector product for complex general matrix.

LAPACK Routines

Single Precision Real

culaDeviceSgels	Solve linear system with QR or LQ factorization.
culaDeviceSgeqrf	QR factorization.
culaDeviceSgesv	Solve linear system with LU factorization.
culaDeviceSgesvd	SVD decomposition.
culaDeviceSgetrf	LU factorization.
culaDeviceSgglse	Solve linear equality-constrained least squares problem.
culaDeviceSposv	Solve positive definite linear system with Cholesky factorization.
culaDeviceSpotrf	Cholesky factorization.

skcuda.cula.culaDeviceSgels

skcuda.cula.**culaDeviceSgels**(*trans*, *m*, *n*, *nrhs*, *a*, *lda*, *b*, *ldb*)
Solve linear system with QR or LQ factorization.

skcuda.cula.culaDeviceSgeqrf

skcuda.cula.**culaDeviceSgeqrf**(*m*, *n*, *a*, *lda*, *tau*)
QR factorization.

skcuda.cula.culaDeviceSgesv

skcuda.cula.**culaDeviceSgesv**(*n*, *nrhs*, *a*, *lda*, *ipiv*, *b*, *ldb*)
Solve linear system with LU factorization.

skcuda.cula.culaDeviceSgesvd

skcuda.cula.**culaDeviceSgesvd**(*jobu*, *jobvt*, *m*, *n*, *a*, *lda*, *s*, *u*, *ldu*, *vt*, *ldvt*)
SVD decomposition.

skcuda.cula.culaDeviceSgetrf

skcuda.cula.**culaDeviceSgetrf**(*m*, *n*, *a*, *lda*, *ipiv*)
LU factorization.

skcuda.cula.culaDeviceSgglse

skcuda.cula.**culaDeviceSgglse**(*m*, *n*, *p*, *a*, *lda*, *b*, *ldb*, *c*, *d*, *x*)
Solve linear equality-constrained least squares problem.

skcuda.cula.culaDeviceSposv

skcuda.cula.**culaDeviceSposv**(*upio*, *n*, *nrhs*, *a*, *lda*, *b*, *ldb*)
Solve positive definite linear system with Cholesky factorization.

skcuda.cula.culaDeviceSpotrf

skcuda.cula.**culaDeviceSpotrf**(*uplo*, *n*, *a*, *lda*)
Cholesky factorization.

Single Precision Complex

culaDeviceCgels	Solve linear system with QR or LQ factorization.
culaDeviceCgeqrf	QR factorization.
culaDeviceCgesv	Solve linear system with LU factorization.
culaDeviceCgesvd	SVD decomposition.
culaDeviceCgetrf	LU factorization.
culaDeviceCgglse	Solve linear equality-constrained least squares problem.
culaDeviceCposv	Solve positive definite linear system with Cholesky factorization.
culaDeviceCpotrf	Cholesky factorization.

skcuda.cula.culaDeviceCgels

skcuda.cula.**culaDeviceCgels**(*trans*, *m*, *n*, *nrhs*, *a*, *lda*, *b*, *ldb*)
Solve linear system with QR or LQ factorization.

skcuda.cula.culaDeviceCgeqrf

skcuda.cula.**culaDeviceCgeqrf**(*m*, *n*, *a*, *lda*, *tau*)
QR factorization.

skcuda.cula.culaDeviceCgesv

skcuda.cula.**culaDeviceCgesv**(*n*, *nrhs*, *a*, *lda*, *ipiv*, *b*, *ldb*)
Solve linear system with LU factorization.

skcuda.cula.culaDeviceCgesvd

skcuda.cula.**culaDeviceCgesvd**(*jobu*, *jobvt*, *m*, *n*, *a*, *lda*, *s*, *u*, *ldu*, *vt*, *ldvt*)
SVD decomposition.

skcuda.cula.culaDeviceCgetrf

skcuda.cula.**culaDeviceCgetrf**(*m*, *n*, *a*, *lda*, *ipiv*)
LU factorization.

skcuda.cula.culaDeviceCgglse

skcuda.cula.**culaDeviceCgglse**(*m*, *n*, *p*, *a*, *lda*, *b*, *ldb*, *c*, *d*, *x*)
Solve linear equality-constrained least squares problem.

skcuda.cula.culaDeviceCposv

skcuda.cula.**culaDeviceCposv**(*upio*, *n*, *nrhs*, *a*, *lda*, *b*, *ldb*)
Solve positive definite linear system with Cholesky factorization.

skcuda.cula.culaDeviceCpotrf

skcuda.cula.**culaDeviceCpotrf**(*uplo*, *n*, *a*, *lda*)
Cholesky factorization.

Double Precision Real

culaDeviceDgels	Solve linear system with QR or LQ factorization.
culaDeviceDgeqrf	QR factorization.
culaDeviceDgesv	Solve linear system with LU factorization.
culaDeviceDgesvd	SVD decomposition.
culaDeviceDgetrf	LU factorization.
culaDeviceDgglse	Solve linear equality-constrained least squares problem.
culaDeviceDposv	Solve positive definite linear system with Cholesky factorization.
culaDeviceDpotrf	Cholesky factorization.

skcuda.cula.culaDeviceDgels

skcuda.cula.**culaDeviceDgels**(*trans*, *m*, *n*, *nrhs*, *a*, *lda*, *b*, *ldb*)
Solve linear system with QR or LQ factorization.

skcuda.cula.culaDeviceDgeqrf

skcuda.cula.**culaDeviceDgeqrf**(*m*, *n*, *a*, *lda*, *tau*)
QR factorization.

skcuda.cula.culaDeviceDgesv

skcuda.cula.**culaDeviceDgesv**(*n*, *nrhs*, *a*, *lda*, *ipiv*, *b*, *ldb*)
Solve linear system with LU factorization.

skcuda.cula.culaDeviceDgesvd

skcuda.cula.**culaDeviceDgesvd**(*jobu*, *jobvt*, *m*, *n*, *a*, *lda*, *s*, *u*, *ldu*, *vt*, *ldvt*)
SVD decomposition.

skcuda.cula.culaDeviceDgetrf

skcuda.cula.**culaDeviceDgetrf**(*m*, *n*, *a*, *lda*, *ipiv*)
LU factorization.

skcuda.cula.culaDeviceDgglse

skcuda.cula.**culaDeviceDgglse**(*m*, *n*, *p*, *a*, *lda*, *b*, *ldb*, *c*, *d*, *x*)
Solve linear equality-constrained least squares problem.

skcuda.cula.culaDeviceDposv

skcuda.cula.**culaDeviceDposv**(*upio*, *n*, *nrhs*, *a*, *lda*, *b*, *ldb*)
Solve positive definite linear system with Cholesky factorization.

skcuda.cula.culaDeviceDpotrf

skcuda.cula.**culaDeviceDpotrf**(*uplo*, *n*, *a*, *lda*)
Cholesky factorization.

Double Precision Complex

culaDeviceZgels	Solve linear system with QR or LQ factorization.
culaDeviceZgeqrf	QR factorization.
culaDeviceZgesv	Solve linear system with LU factorization.
culaDeviceZgesvd	SVD decomposition.
culaDeviceZgetrf	LU factorization.
culaDeviceZgglse	Solve linear equality-constrained least squares problem.
culaDeviceZposv	Solve positive definite linear system with Cholesky factorization.
culaDeviceZpotrf	Cholesky factorization.

skcuda.cula.culaDeviceZgels

skcuda.cula.**culaDeviceZgels**(*trans*, *m*, *n*, *nrhs*, *a*, *lda*, *b*, *ldb*)
Solve linear system with QR or LQ factorization.

skcuda.cula.culaDeviceZgeqrf

skcuda.cula.**culaDeviceZgeqrf**(*m*, *n*, *a*, *lda*, *tau*)
QR factorization.

skcuda.cula.culaDeviceZgesv

skcuda.cula.**culaDeviceZgesv**(*n*, *nrhs*, *a*, *lda*, *ipiv*, *b*, *ldb*)
Solve linear system with LU factorization.

skcuda.cula.culaDeviceZgesvd

skcuda.cula.**culaDeviceZgesvd**(*jobu*, *jobvt*, *m*, *n*, *a*, *lda*, *s*, *u*, *ldu*, *vt*, *ldvt*)
SVD decomposition.

skcuda.cula.culaDeviceZgetrf

skcuda.cula.**culaDeviceZgetrf**(*m*, *n*, *a*, *lda*, *ipiv*)
LU factorization.

skcuda.cula.culaDeviceZgglse

skcuda.cula.**culaDeviceZgglse**(*m*, *n*, *p*, *a*, *lda*, *b*, *ldb*, *c*, *d*, *x*)
Solve linear equality-constrained least squares problem.

skcuda.cula.culaDeviceZposv

skcuda.cula.**culaDeviceZposv**(*upio*, *n*, *nrhs*, *a*, *lda*, *b*, *ldb*)
Solve positive definite linear system with Cholesky factorization.

skcuda.cula.culaDeviceZpotrf

skcuda.cula.**culaDeviceZpotrf**(*uplo*, *n*, *a*, *lda*)
Cholesky factorization.

Multi-GPU CULA Routines

Framework Routines

pculaConfigInit	Initialize pCULA configuration structure to sensible defaults.

skcuda.pcula.pculaConfigInit

skcuda.pcula.**pculaConfigInit**(*config*)
Initialize pCULA configuration structure to sensible defaults.

BLAS Routines

Single Precision Real

pculaSgemm	Matrix-matrix product for general matrix.
pculaStrsm	Triangular system solve.

skcuda.pcula.pculaSgemm

skcuda.pcula.**pculaSgemm**(*config*, *transa*, *transb*, *m*, *n*, *k*, *alpha*, *A*, *lda*, *B*, *ldb*, *beta*, *C*, *ldc*)
Matrix-matrix product for general matrix.

skcuda.pcula.pculaStrsm

skcuda.pcula.**pculaStrsm**(*config*, *side*, *uplo*, *transa*, *diag*, *m*, *n*, *alpha*, *a*, *lda*, *b*, *ldb*)
Triangular system solve.

Single Precision Complex

pculaCgemm	Matrix-matrix product for general matrix.
pculaCtrsm	Triangular system solve.

skcuda.pcula.pculaCgemm

skcuda.pcula.**pculaCgemm**(*config*, *transa*, *transb*, *m*, *n*, *k*, *alpha*, *A*, *lda*, *B*, *ldb*, *beta*, *C*, *ldc*)
Matrix-matrix product for general matrix.

skcuda.pcula.pculaCtrsm

skcuda.pcula.**pculaCtrsm**(*config*, *side*, *uplo*, *transa*, *diag*, *m*, *n*, *alpha*, *a*, *lda*, *b*, *ldb*)
Triangular system solve.

Double Precision Real

pculaDgemm	Matrix-matrix product for general matrix.
pculaDtrsm	Triangular system solve.

skcuda.pculaDgemm

skcuda.pcula.**pculaDgemm**(*config*, *transa*, *transb*, *m*, *n*, *k*, *alpha*, *A*, *lda*, *B*, *ldb*, *beta*, *C*, *ldc*)
Matrix-matrix product for general matrix.

skcuda.pculaDtrsm

skcuda.pcula.**pculaDtrsm**(*config*, *side*, *uplo*, *transa*, *diag*, *m*, *n*, *alpha*, *a*, *lda*, *b*, *ldb*)
Triangular system solve.

Double Precision Complex

pculaZgemm	Matrix-matrix product for general matrix.
pculaZtrsm	Triangular system solve.

skcuda.pculaZgemm

skcuda.pcula.**pculaZgemm**(*config*, *transa*, *transb*, *m*, *n*, *k*, *alpha*, *A*, *lda*, *B*, *ldb*, *beta*, *C*, *ldc*)
Matrix-matrix product for general matrix.

skcuda.pculaZtrsm

skcuda.pcula.**pculaZtrsm**(*config*, *side*, *uplo*, *transa*, *diag*, *m*, *n*, *alpha*, *a*, *lda*, *b*, *ldb*)
Triangular system solve.

LAPACK Routines

Single Precision Real

pculaSgesv	General system solve using LU decomposition.
pculaSgetrf	LU decomposition.
pculaSgetrs	LU solve.
pculaSposv	QR factorization.
pculaSpotrf	Cholesky decomposition.
pculaSpotrs	Cholesky solve.

skcuda.pculaSgesv

skcuda.pcula.**pculaSgesv**(*config*, *n*, *nrhs*, *a*, *lda*, *ipiv*, *b*, *ldb*)
General system solve using LU decomposition.

skcuda.pculaSgetrf

skcuda.pcula.**pculaSgetrf**(*config*, *m*, *n*, *a*, *lda*, *ipiv*)
LU decomposition.

skcuda.pcula.pculaSgetrs

skcuda.pcula.**pculaSgetrs**(*config*, *trans*, *n*, *nrhs*, *a*, *lda*, *ipiv*, *b*, *ldb*)
LU solve.

skcuda.pcula.pculaSposv

skcuda.pcula.**pculaSposv**(*config*, *uplo*, *n*, *nrhs*, *a*, *lda*, *b*, *ldb*)
QR factorization.

skcuda.pcula.pculaSpotrf

skcuda.pcula.**pculaSpotrf**(*config*, *uplo*, *n*, *a*, *lda*)
Cholesky decomposition.

skcuda.pcula.pculaSpotrs

skcuda.pcula.**pculaSpotrs**(*config*, *uplo*, *n*, *nrhs*, *a*, *lda*, *b*, *ldb*)
Cholesky solve.

Single Precision Complex

pculaCgesv	General system solve using LU decomposition.
pculaCgetrf	LU decomposition.
pculaCgetrs	LU solve.
pculaCposv	QR factorization.
pculaCpotrf	Cholesky decomposition.
pculaCpotrs	Cholesky solve.

skcuda.pcula.pculaCgesv

skcuda.pcula.**pculaCgesv**(*config*, *n*, *nrhs*, *a*, *lda*, *ipiv*, *b*, *ldb*)
General system solve using LU decomposition.

skcuda.pcula.pculaCgetrf

skcuda.pcula.**pculaCgetrf**(*config*, *m*, *n*, *a*, *lda*, *ipiv*)
LU decomposition.

skcuda.pcula.pculaCgetrs

skcuda.pcula.**pculaCgetrs**(*config*, *trans*, *n*, *nrhs*, *a*, *lda*, *ipiv*, *b*, *ldb*)
LU solve.

skcuda.pcula.pculaCposv

skcuda.pcula.**pculaCposv**(*config*, *uplo*, *n*, *nrhs*, *a*, *lda*, *b*, *ldb*)
QR factorization.

skcuda.pcula.pculaCpotrf

skcuda.pcula.**pculaCpotrf**(*config*, *uplo*, *n*, *a*, *lda*)
Cholesky decomposition.

skcuda.pcula.pculaCpotrs

skcuda.pcula.**pculaCpotrs**(*config*, *uplo*, *n*, *nrhs*, *a*, *lda*, *b*, *ldb*)
Cholesky solve.

Double Precision Real

pculaDgesv	General system solve using LU decomposition.
pculaDgetrf	LU decomposition.
pculaDgetrs	LU solve.
pculaDposv	QR factorization.
pculaDpotrf	Cholesky decomposition.
pculaDpotrs	Cholesky solve.

skcuda.pcula.pculaDgesv

skcuda.pcula.**pculaDgesv**(*config*, *n*, *nrhs*, *a*, *lda*, *ipiv*, *b*, *ldb*)
General system solve using LU decomposition.

skcuda.pcula.pculaDgetrf

skcuda.pcula.**pculaDgetrf**(*config*, *m*, *n*, *a*, *lda*, *ipiv*)
LU decomposition.

skcuda.pcula.pculaDgetrs

skcuda.pcula.**pculaDgetrs**(*config*, *trans*, *n*, *nrhs*, *a*, *lda*, *ipiv*, *b*, *ldb*)
LU solve.

skcuda.pcula.pculaDposv

skcuda.pcula.**pculaDposv**(*config*, *uplo*, *n*, *nrhs*, *a*, *lda*, *b*, *ldb*)
QR factorization.

skcuda.pcula.pculaDpotrf

skcuda.pcula.**pculaDpotrf**(*config*, *uplo*, *n*, *a*, *lda*)
Cholesky decomposition.

skcuda.pcula.pculaDpotrs

skcuda.pcula.**pculaDpotrs**(*config*, *uplo*, *n*, *nrhs*, *a*, *lda*, *b*, *ldb*)
Cholesky solve.

Double Precision Complex

pculaZgesv	General system solve using LU decomposition.
pculaZgetrf	LU decomposition.
pculaZgetrs	LU solve.
pculaZposv	QR factorization.
pculaZpotrf	Cholesky decomposition.
pculaZpotrs	Cholesky solve.

skcuda.pcula.pculaZgesv

skcuda.pcula.**pculaZgesv**(*config*, *n*, *nrhs*, *a*, *lda*, *ipiv*, *b*, *ldb*)
General system solve using LU decomposition.

skcuda.pcula.pculaZgetrf

skcuda.pcula.**pculaZgetrf**(*config*, *m*, *n*, *a*, *lda*, *ipiv*)
LU decomposition.

skcuda.pcula.pculaZgetrs

skcuda.pcula.**pculaZgetrs**(*config*, *trans*, *n*, *nrhs*, *a*, *lda*, *ipiv*, *b*, *ldb*)
LU solve.

skcuda.pcula.pculaZposv

skcuda.pcula.**pculaZposv**(*config*, *uplo*, *n*, *nrhs*, *a*, *lda*, *b*, *ldb*)
QR factorization.

skcuda.pcula.pculaZpotrf

skcuda.pcula.**pculaZpotrf**(*config*, *uplo*, *n*, *a*, *lda*)
Cholesky decomposition.

skcuda.pculaZpotrs

skcuda.pcula.**pculaZpotrs**(*config*, *uplo*, *n*, *nrhs*, *a*, *lda*, *b*, *ldb*)
Cholesky solve.

1.2.2 High-Level Routines

Fast Fourier Transform

fft
ifft
Plan

Integration Routines

simps	Implementation of composite Simpson's rule similar to scipy.integrate.simps.
trapz	1D trapezoidal integration.
trapz2d	2D trapezoidal integration.

skcuda.integrate.simps

skcuda.integrate.**simps**(*x_gpu*, *dx=1.0*, *even='avg'*, *handle=None*)
Implementation of composite Simpson's rule similar to scipy.integrate.simps.

Integrate x_gpu with spacing dx using composite Simpson's rule. If there are an even number of samples, N, then there are an odd number of intervals (N-1), but Simpson's rule requires an even number of intervals. The parameter 'even' controls how this is handled.

Parameters

- **x_gpu** (*pycuda.gpuarray.GPUArray*) – Input array to integrate.
- **dx** (*scalar*) – Spacing.
- **even** (*str {'avg', 'first', 'last'}, optional*) –

  'avg' [Average two results:1) use the first N-2 intervals with] a trapezoidal rule on the last interval and 2) use the last N-2 intervals with a trapezoidal rule on the first interval.

  'first' [Use Simpson's rule for the first N-2 intervals with] a trapezoidal rule on the last interval.

  'last' [Use Simpson's rule for the last N-2 intervals with a] trapezoidal rule on the first interval.
- **handle** (*int*) – CUBLAS context. If no context is specified, the default handle from *skcuda.misc._global_cublas_handle* is used.

Returns **result** – Definite integral as approximated by the composite Simpson's rule.

Return type float

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray
>>> import numpy as np
>>> import integrate
>>> integrate.init()
>>> x_gpu = gpuarray.arange(0,10,dtype=np.float64)
>>> integrate.simps(x_gpu)
40.5
>>> x_gpu**=3
>>> integrate.simps(x_gpu)
1642.5
>>> integrate.simps(x_gpu, even='first')
1644.5
```

skcuda.integrate.trapz

skcuda.integrate.**trapz**(*x_gpu*, *dx=1.0*, *handle=None*)

1D trapezoidal integration.

Parameters

- **x_gpu** (*pycuda.gpuarray.GPUArray*) – Input array to integrate.
- **dx** (*scalar*) – Spacing.
- **handle** (*int*) – CUBLAS context. If no context is specified, the default handle from *skcuda.misc._global_cublas_handle* is used.

Returns **result** – Definite integral as approximated by the trapezoidal rule.

Return type float

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray
>>> import numpy as np
>>> import integrate
>>> integrate.init()
>>> x = np.asarray(np.random.rand(10), np.float32)
>>> x_gpu = gpuarray.to_gpu(x)
>>> z = integrate.trapz(x_gpu)
>>> np.allclose(np.trapz(x), z)
True
```

skcuda.integrate.trapz2d

skcuda.integrate.**trapz2d**(*x_gpu*, *dx=1.0*, *dy=1.0*, *handle=None*)

2D trapezoidal integration.

Parameters

- **x_gpu** (*pycuda.gpuarray.GPUArray*) – Input matrix to integrate.

- **dx** (*float*) – X-axis spacing.
- **dy** (*float*) – Y-axis spacing
- **handle** (*int*) – CUBLAS context. If no context is specified, the default handle from *skcuda.misc._global_cublas_handle* is used.

Returns **result** – Definite double integral as approximated by the trapezoidal rule.

Return type float

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray
>>> import numpy as np
>>> import integrate
>>> integrate.init()
>>> x = np.asarray(np.random.rand(10, 10), np.float32)
>>> x_gpu = gpuarray.to_gpu(x)
>>> z = integrate.trapz2d(x_gpu)
>>> np.allclose(np.trapz(np.trapz(x)), z)
True
```

Linear Algebra Routines and Classes

add_diag	Adds a vector to the diagonal of an array.
add_dot	Calculates the dot product of two arrays and adds it to a third matrix.
cho_factor	Cholesky factorization.
cho_solve	Cholesky solver.
cholesky	Cholesky factorization.
conj	Complex conjugate.
det	Compute the determinant of a square matrix.
diag	Construct a diagonal matrix if input array is one-dimensional, or extracts diagonal entries of a two-dimensional array.
dmd	Dynamic Mode Decomposition.
dot_diag	Dot product of diagonal and non-diagonal arrays.
dot	Dot product of two arrays.
eig	Eigendecomposition of a matrix.
eye	Construct a 2D matrix with ones on the diagonal and zeros elsewhere.
hermitian	Hermitian (conjugate) matrix transpose.
inv	Compute the inverse of a matrix.
mdot	Product of several matrices.
multiply	Element-wise array multiplication (Hadamard product).
norm	Euclidean norm (2-norm) of real vector.
pinv	Moore-Penrose pseudoinverse.
qr	QR Decomposition.
scale	Scale a vector by a factor alpha.

Continued on next page

Table 39 – continued from previous page

svd	Singular Value Decomposition.
PCA	Principal Component Analysis with similar API to sklearn.decomposition.PCA
trace	Return the sum along the main diagonal of the array.
transpose	Matrix transpose.
tril	Lower triangle of a matrix.
triu	Upper triangle of a matrix.
vander	Generate a Vandermonde matrix.

skcuda.linalg.add_diag

skcuda.linalg.**add_diag**(*d_gpu*, *a_gpu*, *overwrite=False*, *handle=None*)

Adds a vector to the diagonal of an array.

This is the same as A + diag(D), but faster.

Parameters

- **d_gpu** (*pycuda.gpuarray.GPUArray*) – Array of length *N* corresponding to the vector to be added to the diagonal.
- **a_gpu** (*pycuda.gpuarray.GPUArray*) – Summand array with shape *(N, N)*.
- **overwrite** (*bool (default: False)*) – If true, save the result in *a_gpu*.
- **handle** (*int*) – CUBLAS context. If no context is specified, the default handle from *skcuda.misc._global_cublas_handle* is used.

Returns **r_gpu** – The computed sum product.

Return type pycuda.gpuarray.GPUArray

Notes

d_gpu and *a_gpu* must have the same precision data type.

skcuda.linalg.add_dot

skcuda.linalg.**add_dot**(*a_gpu*, *b_gpu*, *c_gpu*, *transa='N'*, *transb='N'*, *alpha=1.0*, *beta=1.0*, *handle=None*)

Calculates the dot product of two arrays and adds it to a third matrix.

In essence, this computes

C = alpha * (A B) + beta * C

For 2D arrays of shapes *(m, k)* and *(k, n)*, it computes the matrix product; the result has shape *(m, n)*.

Parameters

- **a_gpu** (*pycuda.gpuarray.GPUArray*) – Input array.
- **b_gpu** (*pycuda.gpuarray.GPUArray*) – Input array.
- **c_gpu** (*pycuda.gpuarray.GPUArray*) – Cumulative array.

- **transa** (*char*) – If 'T', compute the product of the transpose of *a_gpu*. If 'C', compute the product of the Hermitian of *a_gpu*.
- **transb** (*char*) – If 'T', compute the product of the transpose of *b_gpu*. If 'C', compute the product of the Hermitian of *b_gpu*.
- **handle** (*int (optional)*) – CUBLAS context. If no context is specified, the default handle from *skcuda.misc._global_cublas_handle* is used.

Returns **c_gpu**

Return type pycuda.gpuarray.GPUArray

Notes

The matrices must all contain elements of the same data type.

skcuda.linalg.cho_factor

skcuda.linalg.**cho_factor**(*a_gpu*, *uplo='L'*, *lib='cusolver'*)

Cholesky factorization.

Performs an in-place Cholesky factorization on the matrix *a* such that *a = x*x.T* or *x.T*x*, if the lower='L' or upper='U' triangle of *a* is used, respectively.

Parameters

- **a_gpu** (*pycuda.gpuarray.GPUArray*) – Input matrix of shape *(m, m)* to decompose.
- **uplo** (*{'U', 'L'}*) – Use upper or lower (default) triangle of 'a_gpu'
- **lib** (*str*) – Library to use. May be either 'cula' or 'cusolver'.

Notes

If using CULA, double precision is only supported if the standard version of the CULA Dense toolkit is installed.

Examples

```
>>> import pycuda.gpuarray as gpuarray
>>> import pycuda.autoinit
>>> import numpy as np
>>> import scipy.linalg
>>> import skcuda.linalg as linalg
>>> linalg.init()
>>> a = np.array([[3.0,0.0],[0.0,7.0]])
>>> a = np.asarray(a, np.float64)
>>> a_gpu = gpuarray.to_gpu(a)
>>> cho_factor(a_gpu)
>>> np.allclose(a_gpu.get(), scipy.linalg.cho_factor(a)[0])
True
```

skcuda.linalg.cho_solve

skcuda.linalg.**cho_solve**(*a_gpu*, *b_gpu*, *uplo='L'*, *lib='cusolver'*)

Cholesky solver.

Solve a system of equations via Cholesky factorization, i.e. *a*x = b*. Overwrites *b* to give *inv(a)*b*, and overwrites the chosen triangle of *a* with factorized triangle.

Parameters

- **a** (*pycuda.gpuarray.GPUArray*) – Input matrix of shape *(m, m)* to decompose.
- **b** (*pycuda.gpuarray.GPUArray*) – Input matrix of shape *(m, 1)* to decompose.
- **uplo** (*chr*) – Use the upper='U' or lower='L' (default) triangle of *a*.
- **lib** (*str*) – Library to use. May be either 'cula' or 'cusolver'.

Notes

If using CULA, double precision is only supported if the standard version of the CULA Dense toolkit is installed.

Examples

```
>>> import pycuda.gpuarray as gpuarray
>>> import pycuda.autoinit
>>> import numpy as np
>>> import scipy.linalg
>>> import skcuda.linalg as linalg
>>> linalg.init()
>>> a = np.array([[3, 0], [0, 7]]).asarray(np.float64)
>>> a_gpu = gpuarray.to_gpu(a)
>>> b = np.array([11, 19]).astype(np.float64)
>>> b_gpu  = gpuarray.to_gpu(b)
>>> cho_solve(a_gpu, b_gpu)
>>> np.allclose(b_gpu.get(), scipy.linalg.cho_solve(scipy.linalg.cho_factor(a),␣
↪b))
True
```

skcuda.linalg.cholesky

skcuda.linalg.**cholesky**(*a_gpu*, *uplo='L'*, *lib='cusolver'*)

Cholesky factorization.

Performs an in-place Cholesky factorization on the matrix *a* such that *a = x*x.T* or *x.T*x*, if the lower='L' or upper='U' triangle of *a* is used, respectively. All other entries in *a* are set to 0.

Parameters

- **a_gpu** (*pycuda.gpuarray.GPUArray*) – Input matrix of shape *(m, m)* to decompose.
- **uplo** (*{'U', 'L'}*) – Use upper or lower (default) triangle of 'a_gpu'
- **lib** (*str*) – Library to use. May be either 'cula' or 'cusolver'.

Notes

If using CULA, double precision is only supported if the standard version of the CULA Dense toolkit is installed.

Examples

```
>>> import pycuda.gpuarray as gpuarray
>>> import pycuda.autoinit
>>> import numpy as np
>>> import scipy.linalg
>>> import skcuda.linalg as linalg
>>> linalg.init()
>>> a = np.array([[3.0,0.0],[0.0,7.0]])
>>> a = np.asarray(a, np.float64)
>>> a_gpu = gpuarray.to_gpu(a)
>>> cholesky(a_gpu)
>>> np.allclose(a_gpu.get(), scipy.linalg.cholesky(a)[0])
True
```

skcuda.linalg.conj

skcuda.linalg.**conj**(*x_gpu*, *overwrite=False*)

Complex conjugate.

Compute the complex conjugate of the array in device memory.

Parameters

- **x_gpu** (*pycuda.gpuarray.GPUArray*) – Input array of shape *(m, n)*.
- **overwrite** (*bool (default: False)*) – If true, save the result in the specified array. If false, return the result in a newly allocated array.

Returns **xc_gpu** – Conjugate of the input array. If *overwrite* is true, the returned matrix is the same as the input array.

Return type pycuda.gpuarray.GPUArray

Examples

```
>>> import pycuda.driver as drv
>>> import pycuda.gpuarray as gpuarray
>>> import pycuda.autoinit
>>> import numpy as np
>>> import skcuda.linalg as linalg
>>> linalg.init()
>>> x = np.array([[1+1j, 2-2j, 3+3j, 4-4j], [5+5j, 6-6j, 7+7j, 8-8j]], np.
↪complex64)
>>> x_gpu = gpuarray.to_gpu(x)
>>> y_gpu = linalg.conj(x_gpu)
>>> np.all(x == np.conj(y_gpu.get()))
True
```

skcuda.linalg.det

skcuda.linalg.**det**(*a_gpu*, *overwrite=False*, *workspace_gpu=None*, *ipiv_gpu=None*, *handle=None*, *lib='cusolver'*)

Compute the determinant of a square matrix.

Parameters

- **a_gpu** (*pycuda.gpuarray.GPUArray*) – The square n*n matrix of which to calculate the determinant.
- **overwrite** (*bool (default: False)*) – Discard data in *a* (may improve performance).
- **workspace_gpu** (*pycuda.gpuarray.GPUArray (optional)*) – Temporary array of size Lwork (typically computed by CUSOLVER helper functions), can be supplied to save allocations. Only used if lib == 'cusolver'.
- **ipiv_gpu** (*pycuda.gpuarray.GPUArray (optional)*) – Temporary array of size n, can be supplied to save allocations.
- **handle** (*int*) – CUBLAS context. If no context is specified, the default handle from *skcuda.misc._global_cublas_handle* is used.
- **lib** (*str*) – Library to use. May be either 'cula' or 'cusolver'.

Returns **det** – determinant of a_gpu

Return type number

skcuda.linalg.diag

skcuda.linalg.**diag**(*v_gpu*)

Construct a diagonal matrix if input array is one-dimensional, or extracts diagonal entries of a two-dimensional array.

If input-array is one-dimensional, constructs a matrix in device memory whose diagonal elements correspond to the elements in the specified array; all non-diagonal elements are set to 0.

If input-array is two-dimensional, constructs an array in device memory whose elements correspond to the elements along the main-diagonal of the specified array.

Parameters **v_obj** (*pycuda.gpuarray.GPUArray*) – Input array of shape *(n,m)*.

Returns **d_gpu** – If v_obj has shape *(n,1)*, output is diagonal matrix of dimensions *[n, n]*. If v_obj has shape *(n,m)*, output is array of length *min(n,m)*.

Return type pycuda.gpuarray.GPUArray

Examples

```
>>> import pycuda.driver as drv
>>> import pycuda.gpuarray as gpuarray
>>> import pycuda.autoinit
>>> import numpy as np
>>> import skcuda.linalg as linalg
>>> linalg.init()
>>> v = np.array([1, 2, 3, 4, 5, 6], np.float32)
```

(continues on next page)

(continued from previous page)

```
>>> v_gpu = gpuarray.to_gpu(v)
>>> d_gpu = linalg.diag(v_gpu)
>>> np.all(d_gpu.get() == np.diag(v))
True
>>> v = np.array([1j, 2j, 3j, 4j, 5j, 6j], np.complex64)
>>> v_gpu = gpuarray.to_gpu(v)
>>> d_gpu = linalg.diag(v_gpu)
>>> np.all(d_gpu.get() == np.diag(v))
True
>>> v = np.array([[1., 2., 3.],[4., 5., 6.]], np.float64)
>>> v_gpu = gpuarray.to_gpu(v)
>>> d_gpu = linalg.diag(v_gpu)
>>> d_gpu
array([ 1.,  5.])
```

skcuda.linalg.dmd

skcuda.linalg.**dmd**(*a_gpu*, *k=None*, *modes='exact'*, *return_amplitudes=False*, *return_vandermonde=False*, *handle=None*)

Dynamic Mode Decomposition.

Dynamic Mode Decomposition (DMD) is a data processing algorithm which allows to decompose a matrix *a* in space and time. The matrix *a* is decomposed as *a = FBV*, where the columns of *F* contain the dynamic modes. The modes are ordered corresponding to the amplitudes stored in the diagonal matrix *B*. *V* is a Vandermonde matrix describing the temporal evolution.

Parameters

- **a_gpu** (*pycuda.gpuarray.GPUArray*) – Real/complex input matrix *a* with dimensions *(m, n)*.
- **k** (*int, optional*) – If *k < (n-1)* low-rank Dynamic Mode Decomposition is computed.
- **modes** (*{'standard', 'exact'}*) –

  'standard' [uses the standard definition to compute the dynamic modes,] *F = U * W*.

  'exact' : computes the exact dynamic modes, *F = Y * V * (S**-1) * W*.
- **return_amplitudes** (bool *{True, False}*) – True: return amplitudes in addition to dynamic modes.
- **return_vandermonde** (bool *{True, False}*) – True: return Vandermonde matrix in addition to dynamic modes and amplitudes.
- **handle** (*int*) – CUBLAS context. If no context is specified, the default handle from *skcuda.misc._global_cublas_handle* is used.

Returns

- **f_gpu** (*pycuda.gpuarray.GPUArray*) – Matrix containing the dynamic modes of shape *(m, n-1)* or *(m, k)*.
- **b_gpu** (*pycuda.gpuarray.GPUArray*) – 1-D array containing the amplitudes of length *min(n-1, k)*.
- **v_gpu** (*pycuda.gpuarray.GPUArray*) – Vandermonde matrix of shape *(n-1, n-1)* or *(k, n-1)*.

Notes

Double precision is only supported if the standard version of the CULA Dense toolkit is installed.

This function destroys the contents of the input matrix.

Arrays are assumed to be stored in column-major order, i.e., order='F'.

References

M. R. Jovanovic, P. J. Schmid, and J. W. Nichols. "Low-rank and sparse dynamic mode decomposition." Center for Turbulence Research Annual Research Briefs (2012): 139-152.

J. H. Tu, et al. "On dynamic mode decomposition: theory and applications." arXiv preprint arXiv:1312.0041 (2013).

```
>>> #Numpy
>>> import numpy as np
>>> #Plot libs
>>> import matplotlib.pyplot as plt
>>> from mpl_toolkits.mplot3d import Axes3D
>>> from matplotlib import cm
>>> #GPU DMD libs
>>> import pycuda.gpuarray as gpuarray
>>> import pycuda.autoinit
>>> from skcuda import linalg, rlinalg
>>> linalg.init()
```

```
>>> # Define time and space discretizations
>>> x=np.linspace( -15, 15, 200)
>>> t=np.linspace(0, 8*np.pi , 80)
>>> dt=t[2]-t[1]
>>> X, T = np.meshgrid(x,t)
>>> # Create two patio-temporal patterns
>>> F1 = 0.5* np.cos(X)*(1.+0.* T)
>>> F2 = ( (1./np.cosh(X)) * np.tanh(X)) *(2.*np.exp(1j*2.8*T))
>>> # Add both signals
>>> F = (F1+F2)
```

```
>>> #Plot dataset
>>> fig = plt.figure()
>>> ax = fig.add_subplot(231, projection='3d')
>>> ax = fig.gca(projection='3d')
>>> surf = ax.plot_surface(X, T, F, rstride=1, cstride=1, cmap=cm.
↪coolwarm, linewidth=0, antialiased=True)
>>> ax.set_zlim(-1, 1)
>>> plt.title('F')
>>> ax = fig.add_subplot(232, projection='3d')
>>> ax = fig.gca(projection='3d')
>>> surf = ax.plot_surface(X, T, F1, rstride=1, cstride=1, cmap=cm.
↪coolwarm, linewidth=0, antialiased=False)
>>> ax.set_zlim(-1, 1)
>>> plt.title('F1')
>>> ax = fig.add_subplot(233, projection='3d')
>>> ax = fig.gca(projection='3d')
>>> surf = ax.plot_surface(X, T, F2, rstride=1, cstride=1, cmap=cm.
↪coolwarm, linewidth=0, antialiased=False)
```

(continues on next page)

(continued from previous page)

```
>>> ax.set_zlim(-1, 1)
>>> plt.title('F2')
```

```
>>> #Dynamic Mode Decomposition
>>> F_gpu = np.array(F.T, np.complex64, order='F')
>>> F_gpu = gpuarray.to_gpu(F_gpu)
>>> Fmodes_gpu, b_gpu, V_gpu, omega_gpu = linalg.dmd(F_gpu, k=2, modes=
↪'exact', return_amplitudes=True, return_vandermonde=True)
>>> omega = omega_gpu.get()
>>> plt.scatter(omega.real, omega.imag, marker='o', c='r')
```

```
>>> #Recover original signal
>>> F1tilde = np.dot(Fmodes_gpu[:,0:1].get() , np.dot(b_gpu[0].get(), V_
↪gpu[0:1,:].get() ) )
>>> F2tilde = np.dot(Fmodes_gpu[:,1:2].get() , np.dot(b_gpu[1].get(), V_
↪gpu[1:2,:].get() ) )
```

```
>>> #Plot DMD modes
>>> #Mode 0
>>> ax = fig.add_subplot(235, projection='3d')
>>> ax = fig.gca(projection='3d')
>>> surf = ax.plot_surface(X[0:F1tilde.shape[1],:], T[0:F1tilde.shape[1],
↪:], F1tilde.T, rstride=1, cstride=1, cmap=cm.coolwarm, linewidth=0,␣
↪antialiased=False)
>>> ax.set_zlim(-1, 1)
>>> plt.title('F1_tilde')
>>> #Mode 1
>>> ax = fig.add_subplot(236, projection='3d')
>>> ax = fig.gca(projection='3d')
>>> surf = ax.plot_surface(X[0:F2tilde.shape[1],:], T[0:F2tilde.shape[1],
↪:], F2tilde.T, rstride=1, cstride=1, cmap=cm.coolwarm, linewidth=0,␣
↪antialiased=False)
>>> ax.set_zlim(-1, 1)
>>> plt.title('F2_tilde')
>>> plt.show()
```

skcuda.linalg.dot_diag

skcuda.linalg.**dot_diag**(*d_gpu*, *a_gpu*, *trans='N'*, *overwrite=False*, *handle=None*)

Dot product of diagonal and non-diagonal arrays.

Computes the matrix product of a diagonal array represented as a vector and a non-diagonal array.

Parameters

- **d_gpu** (*pycuda.gpuarray.GPUArray*) – Array of length *N* corresponding to the diagonal of the multiplier.
- **a_gpu** (*pycuda.gpuarray.GPUArray*) – Multiplicand array with shape *(N, M)*. Must have same data type as *d_gpu*.
- **trans** (*char*) – If 'T', compute the product of the transpose of *a_gpu*.
- **overwrite** (*bool (default: False)*) – If true, save the result in *a_gpu*.
- **handle** (*int*) – CUBLAS context. If no context is specified, the default handle from *skcuda.misc._global_cublas_handle* is used.

Returns **r_gpu** – The computed matrix product.

Return type pycuda.gpuarray.GPUArray

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> import skcuda.linalg as linalg
>>> linalg.init()
>>> d = np.random.rand(4)
>>> a = np.random.rand(4, 4)
>>> d_gpu = gpuarray.to_gpu(d)
>>> a_gpu = gpuarray.to_gpu(a)
>>> r_gpu = linalg.dot_diag(d_gpu, a_gpu)
>>> np.allclose(np.dot(np.diag(d), a), r_gpu.get())
True
```

skcuda.linalg.dot

skcuda.linalg.**dot**(*x_gpu*, *y_gpu*, *transa='N'*, *transb='N'*, *handle=None*, *out=None*)

Dot product of two arrays.

For 1D arrays, this function computes the inner product. For 2D arrays of shapes *(m, k)* and *(k, n)*, it computes the matrix product; the result has shape *(m, n)*.

Parameters

- **x_gpu** (*pycuda.gpuarray.GPUArray*) – Input array.
- **y_gpu** (*pycuda.gpuarray.GPUArray*) – Input array.
- **transa** (*char*) – If 'T', compute the product of the transpose of *x_gpu*. If 'C', compute the product of the Hermitian of *x_gpu*.
- **transb** (*char*) – If 'T', compute the product of the transpose of *y_gpu*. If 'C', compute the product of the Hermitian of *y_gpu*.
- **handle** (*int*) – CUBLAS context. If no context is specified, the default handle from *skcuda.misc._global_cublas_handle* is used.
- **out** (*pycuda.gpuarray.GPUArray, optional*) – Output argument. Will be used to store the result.

Returns **c_gpu** – Inner product of *x_gpu* and *y_gpu*. When the inputs are 1D arrays, the result will be returned as a scalar.

Return type pycuda.gpuarray.GPUArray, float{32,64}, or complex{64,128}

Notes

The input matrices must all contain elements of the same data type.

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> import skcuda.linalg as linalg
>>> import skcuda.misc as misc
>>> linalg.init()
>>> a = np.asarray(np.random.rand(4, 2), np.float32)
>>> b = np.asarray(np.random.rand(2, 2), np.float32)
>>> a_gpu = gpuarray.to_gpu(a)
>>> b_gpu = gpuarray.to_gpu(b)
>>> c_gpu = linalg.dot(a_gpu, b_gpu)
>>> np.allclose(np.dot(a, b), c_gpu.get())
True
>>> d = np.asarray(np.random.rand(5), np.float32)
>>> e = np.asarray(np.random.rand(5), np.float32)
>>> d_gpu = gpuarray.to_gpu(d)
>>> e_gpu = gpuarray.to_gpu(e)
>>> f = linalg.dot(d_gpu, e_gpu)
>>> np.allclose(np.dot(d, e), f)
True
```

skcuda.linalg.eig

skcuda.linalg.**eig**(*a_gpu*, *jobvl='N'*, *jobvr='V'*, *imag='F'*, *lib='cusolver'*)

Eigendecomposition of a matrix.

Compute the eigenvalues *w* for a real/complex square matrix *a* and (optionally) the real left and right eigenvectors *vl*, *vr*.

Parameters

- **a_gpu** (*pycuda.gpuarray.GPUArray*) – Real/complex input matrix *a* with dimensions *(m, n)*.
- **jobvl** (*{'V', 'N'}*) – 'V' : returns *vl*, the left eigenvectors of *a* with dimensions *(m, m)*. 'N' : left eigenvectors are not computed.
- **jobvr** (*{'V', 'N'}*) – 'V' : returns *vr*, the right eigenvectors of *a* with dimensions *(m, m)*, (default). 'N' : right eigenvectors are not computed.
- **imag** (*{'F', 'T'}*) – 'F' : imaginary parts of a real matrix are not returned (default). 'T' : returns the imaginary parts of a real matrix (only relevant in the case of single/double precision).
- **lib** (*str*) – Library to use. May be either 'cula' or 'cusolver'. If using 'cusolver', only symmetric/Hermitian matrices are supported.

Returns

- **vr_gpu** (*pycuda.gpuarray.GPUArray*) – The normalized (Euclidean norm equal to 1) right eigenvectors, such that the column *vr[:,i]* is the eigenvector corresponding to the eigenvalue *w[i]*.
- **w_gpu** (*pycuda.gpuarray.GPUArray*) – Array containing the real/complex eigenvalues, not necessarily ordered. *w* is of length *m*.

- **vl_gpu** (*pycuda.gpuarray.GPUArray*) – The normalized (Euclidean norm equal to 1) left eigenvectors, such that the column *vl[:,i]* is the eigenvector corresponding to the eigenvalue *w[i]*.

Notes

Double precision is only supported if the standard version of the CULA Dense toolkit is installed.

This function destroys the contents of the input matrix.

Arrays are expected to be stored in column-major order, i.e., order='F'.

Examples

```
>>> import pycuda.gpuarray as gpuarray
>>> import pycuda.autoinit
>>> import numpy as np
>>> from skcuda import linalg
>>> linalg.init()
>>> # Compute right eigenvectors of a symmetric matrix A and verify A*vr = vr*w
>>> a = np.array(([1,3],[3,5]), np.float32, order='F')
>>> a_gpu = gpuarray.to_gpu(a)
>>> vr_gpu, w_gpu = linalg.eig(a_gpu, 'N', 'V')
>>> np.allclose(np.dot(a, vr_gpu.get()), np.dot(vr_gpu.get(), np.diag(w_gpu.
↪get())), 1e-4)
True
>>> # Compute left eigenvectors of a symmetric matrix A and verify vl.T*A = w*vl.T
>>> a = np.array(([1,3],[3,5]), np.float32, order='F')
>>> a_gpu = gpuarray.to_gpu(a)
>>> w_gpu, vl_gpu = linalg.eig(a_gpu, 'V', 'N')
>>> np.allclose(np.dot(vl_gpu.get().T, a), np.dot(np.diag(w_gpu.get()), vl_gpu.
↪get().T), 1e-4)
True
>>> # Compute left/right eigenvectors of a symmetric matrix A and verify A =␣
↪vr*w*vl.T
>>> a = np.array(([1,3],[3,5]), np.float32, order='F')
>>> a_gpu = gpuarray.to_gpu(a)
>>> vr_gpu, w_gpu, vl_gpu = linalg.eig(a_gpu, 'V', 'V')
>>> np.allclose(a, np.dot(vr_gpu.get(), np.dot(np.diag(w_gpu.get()), vl_gpu.get().
↪T)), 1e-4)
True
>>> # Compute eigenvalues of a square matrix A and verify that trace(A)=sum(w)
>>> a = np.array(np.random.rand(9,9), np.float32, order='F')
>>> a_gpu = gpuarray.to_gpu(a)
>>> w_gpu = linalg.eig(a_gpu, 'N', 'N')
>>> np.allclose(np.trace(a), sum(w_gpu.get()), 1e-4)
True
>>> # Compute eigenvalues of a real valued matrix A possessing complex e-valuesand
>>> a = np.array(np.array(([1, -2], [1, 3])), np.float32, order='F')
>>> a_gpu = gpuarray.to_gpu(a)
>>> w_gpu = linalg.eig(a_gpu, 'N', 'N', imag='T')
True
>>> # Compute eigenvalues of a complex valued matrix A and verify that␣
↪trace(A)=sum(w)
>>> a = np.array(np.random.rand(2,2) + 1j*np.random.rand(2,2), np.complex64,␣
↪order='F')
```

(continues on next page)

(continued from previous page)

```
>>> a_gpu = gpuarray.to_gpu(a)
>>> w_gpu = linalg.eig(a_gpu, 'N', 'N')
>>> np.allclose(np.trace(a), sum(w_gpu.get()), 1e-4)
True
```

skcuda.linalg.eye

skcuda.linalg.**eye**(*N*, *dtype=<Mock object>*)

Construct a 2D matrix with ones on the diagonal and zeros elsewhere.

Constructs a matrix in device memory whose diagonal elements are set to 1 and non-diagonal elements are set to 0.

Parameters

- **N** (*int*) – Number of rows or columns in the output matrix.
- **dtype** (*type*) – Matrix data type.

Returns **e_gpu** – Diagonal matrix of dimensions *[N, N]* with diagonal values set to 1.

Return type pycuda.gpuarray.GPUArray

Examples

```
>>> import pycuda.driver as drv
>>> import pycuda.gpuarray as gpuarray
>>> import pycuda.autoinit
>>> import numpy as np
>>> import skcuda.linalg as linalg
>>> linalg.init()
>>> N = 5
>>> e_gpu = linalg.eye(N)
>>> np.all(e_gpu.get() == np.eye(N))
True
>>> e_gpu = linalg.eye(N, np.complex64)
>>> np.all(e_gpu.get() == np.eye(N, dtype=np.complex64))
True
```

skcuda.linalg.hermitian

skcuda.linalg.**hermitian**(*a_gpu*, *handle=None*)

Hermitian (conjugate) matrix transpose.

Conjugate transpose a matrix in device memory and return an object representing the transposed matrix.

Parameters

- **a_gpu** (*pycuda.gpuarray.GPUArray*) – Input matrix of shape *(m, n)*.
- **handle** (*int*) – CUBLAS context. If no context is specified, the default handle from *skcuda.misc._global_cublas_handle* is used.

Returns **at_gpu** – Transposed matrix of shape *(n, m)*.

Return type pycuda.gpuarray.GPUArray

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.driver as drv
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> import skcuda.linalg as linalg
>>> linalg.init()
>>> a = np.array([[1, 2, 3, 4, 5, 6], [7, 8, 9, 10, 11, 12]], np.float32)
>>> a_gpu = gpuarray.to_gpu(a)
>>> at_gpu = linalg.hermitian(a_gpu)
>>> np.all(a.T == at_gpu.get())
True
>>> b = np.array([[1j, 2j, 3j, 4j, 5j, 6j], [7j, 8j, 9j, 10j, 11j, 12j]], np.
↪complex64)
>>> b_gpu = gpuarray.to_gpu(b)
>>> bt_gpu = linalg.hermitian(b_gpu)
>>> np.all(np.conj(b.T) == bt_gpu.get())
True
```

skcuda.linalg.inv

skcuda.linalg.**inv**(*a_gpu*, *overwrite=False*, *ipiv_gpu=None*, *lib='cusolver'*)

Compute the inverse of a matrix.

Parameters

- **a_gpu** (*pycuda.gpuarray.GPUArray*) – Square (n, n) matrix to be inverted.
- **overwrite** (*bool (default: False)*) – Discard data in *a* (may improve performance).
- **ipiv_gpu** (*pycuda.gpuarray.GPUArray (optional)*) – Temporary array of size *n*, can be supplied to save allocations.
- **lib** (*str*) – Library to use. May be either 'cula' or 'cusolver'.

Returns **ainv_gpu** – Inverse of the matrix *a*.

Return type pycuda.gpuarray.GPUArray

Raises

- *LinAlgError :* – If *a* is singular.
- *ValueError :* – * If *a* is not square, or not 2-dimensional. * If ipiv was not None but had the wrong dtype or shape.

Notes

When the CUSOLVER backend is selected, an extra copy will be performed if *overwrite* is set to transfer the result back into the input matrix.

skcuda.linalg.mdot

skcuda.linalg.**mdot**(**args*, ***kwargs*)

Product of several matrices.

Computes the matrix product of several arrays of shapes.

Parameters

- **b_gpu, ..** (*a_gpu,*) – Arrays to multiply.
- **handle** (*int*) – CUBLAS context. If no context is specified, the default handle from *skcuda.misc._global_cublas_handle* is used.

Returns **c_gpu** – Matrix product of *a_gpu*, *b_gpu*, etc.

Return type pycuda.gpuarray.GPUArray

Notes

The input matrices must all contain elements of the same data type.

Examples

```
>>> import pycuda.gpuarray as gpuarray
>>> import pycuda.autoinit
>>> import numpy as np
>>> import skcuda.linalg as linalg
>>> linalg.init()
>>> a = np.asarray(np.random.rand(4, 2), np.float32)
>>> b = np.asarray(np.random.rand(2, 2), np.float32)
>>> c = np.asarray(np.random.rand(2, 2), np.float32)
>>> a_gpu = gpuarray.to_gpu(a)
>>> b_gpu = gpuarray.to_gpu(b)
>>> c_gpu = gpuarray.to_gpu(c)
>>> d_gpu = linalg.mdot(a_gpu, b_gpu, c_gpu)
>>> np.allclose(np.dot(a, np.dot(b, c)), d_gpu.get())
True
```

skcuda.linalg.multiply

skcuda.linalg.**multiply**(*x_gpu*, *y_gpu*, *overwrite=False*)

Element-wise array multiplication (Hadamard product).

Parameters

- **y_gpu** (*x_gpu,*) – Input arrays to be multiplied.
- **dev** (*pycuda.driver.Device*) – Device object to be used.
- **overwrite** (*bool (default: False)*) – If true, return the result in *y_gpu*. is false, return the result in a newly allocated array.

Returns **z_gpu** – The element-wise product of the input arrays.

Return type pycuda.gpuarray.GPUArray

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> import skcuda.linalg as linalg
>>> linalg.init()
>>> x = np.asarray(np.random.rand(4, 4), np.float32)
>>> y = np.asarray(np.random.rand(4, 4), np.float32)
>>> x_gpu = gpuarray.to_gpu(x)
>>> y_gpu = gpuarray.to_gpu(y)
>>> z_gpu = linalg.multiply(x_gpu, y_gpu)
>>> np.allclose(x*y, z_gpu.get())
True
```

skcuda.linalg.norm

skcuda.linalg.**norm**(*x_gpu*, *handle=None*)

Euclidean norm (2-norm) of real vector.

Computes the Euclidean norm of an array.

Parameters

- **x_gpu** (*pycuda.gpuarray.GPUArray*) – Input array.
- **handle** (*int*) – CUBLAS context. If no context is specified, the default handle from *skcuda.misc._global_cublas_handle* is used.

Returns **nrm** – Euclidean norm of *x*.

Return type real

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> import skcuda.linalg as linalg
>>> linalg.init()
>>> x = np.asarray(np.random.rand(4, 4), np.float32)
>>> x_gpu = gpuarray.to_gpu(x)
>>> nrm = linalg.norm(x_gpu)
>>> np.allclose(nrm, np.linalg.norm(x))
True
>>> x_gpu = gpuarray.to_gpu(np.array([3+4j, 12-84j]))
>>> linalg.norm(x_gpu)
85.0
```

skcuda.linalg.pinv

skcuda.linalg.**pinv**(*a_gpu*, *rcond=1e-15*, *lib='cusolver'*)

Moore-Penrose pseudoinverse.

Compute the Moore-Penrose pseudoinverse of the specified matrix.

Parameters

- **a_gpu** (*pycuda.gpuarray.GPUArray*) – Input matrix of shape *(m, n)*.
- **rcond** (*float*) – Singular values smaller than *rcond'*max(singular_values)* are set to zero.
- **lib** (*str*) – Library to use. May be either 'cula' or 'cusolver'.

Returns **a_inv_gpu** – Pseudoinverse of input matrix.

Return type pycuda.gpuarray.GPUArray

Notes

Double precision is only supported if the standard version of the CULA Dense toolkit is installed.

This function destroys the contents of the input matrix.

If the input matrix is square, the pseudoinverse uses less memory.

Examples

```
>>> import pycuda.driver as drv
>>> import pycuda.gpuarray as gpuarray
>>> import pycuda.autoinit
>>> import numpy as np
>>> import skcuda.linalg as linalg
>>> linalg.init()
>>> a = np.asarray(np.random.rand(8, 4), np.float32)
>>> a_gpu = gpuarray.to_gpu(a)
>>> a_inv_gpu = linalg.pinv(a_gpu)
>>> np.allclose(np.linalg.pinv(a), a_inv_gpu.get(), 1e-4)
True
>>> b = np.asarray(np.random.rand(8, 4)+1j*np.random.rand(8, 4), np.complex64)
>>> b_gpu = gpuarray.to_gpu(b)
>>> b_inv_gpu = linalg.pinv(b_gpu)
>>> np.allclose(np.linalg.pinv(b), b_inv_gpu.get(), 1e-4)
True
```

Notes

The CUSOLVER backend cannot be used with CUDA 7.0.

skcuda.linalg.qr

skcuda.linalg.**qr**(*a_gpu*, *mode='reduced'*, *handle=None*, *lib='cusolver'*)

QR Decomposition.

Factor the real/complex matrix *a* as *QR*, where *Q* is an orthonormal/unitary matrix and *R* is an upper triangular matrix.

Parameters

- **a_gpu** (*pycuda.gpuarray.GPUArray*) – Real/complex input matrix *a* with dimensions *(m, n)*. *a* is assumed to be *m*>='n'.

- **mode** (*{'reduced', 'economic', 'r'}*) – 'reduced' : returns *Q*, *R* with dimensions *(m, k)* and *(k, n)* (default). 'economic' : returns *Q* only with dimensions *(m, k)*. 'r' : returns *R* only with dimensions *(k, n)* with *k'=min'(m,n)*.
- **handle** (*int*) – CUBLAS context. If no context is specified, the default handle from *skcuda.misc._global_cublas_handle* is used.
- **lib** (*str*) – Library to use. May be either 'cula' or 'cusolver'.

Returns

- **q_gpu** (*pycuda.gpuarray.GPUArray*) – Orthonormal/unitary matrix (depending on whether or not *A* is real/complex).
- **r_gpu** (*pycuda.gpuarray.GPUArray*) – The upper-triangular matrix.

Notes

Double precision is only supported if the standard version of the CULA Dense toolkit is installed.

This function destroys the contents of the input matrix.

Arrays are assumed to be stored in column-major order, i.e., order='F'.

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> import skcuda.linalg as linalg
>>> linalg.init()
>>> # Rectangular matrix A, np.float32
>>> A = np.array(np.random.randn(9, 7), np.float32, order='F')
>>> A_gpu = gpuarray.to_gpu(A)
>>> Q_gpu, R_gpu = linalg.qr(A_gpu, 'reduced')
>>> np.allclose(A, np.dot(Q_gpu.get(), R_gpu.get()), 1e-4)
True
>>> # Square matrix A, np.complex128
>>> A = np.random.randn(9, 9) + 1j*np.random.randn(9, 9)
>>> A = np.asarray(A, np.complex128, order='F')
>>> A_gpu = gpuarray.to_gpu(A)
>>> Q_gpu, R_gpu = linalg.qr(A_gpu, 'reduced')
>>> np.allclose(A, np.dot(Q_gpu.get(), R_gpu.get()), 1e-4)
True
>>> np.allclose(np.identity(Q_gpu.shape[0]) + 1j*0, np.dot(Q_gpu.get().conj().T,␣
↪Q_gpu.get()), 1e-4)
True
>>> # Numpy QR and CULA QR
>>> A = np.array(np.random.randn(9, 7), np.float32, order='F')
>>> Q, R = np.linalg.qr(A, 'reduced')
>>> a_gpu = gpuarray.to_gpu(A)
>>> Q_gpu, R_gpu = linalg.qr(a_gpu, 'reduced')
>>> np.allclose(Q, Q_gpu.get(), 1e-4)
True
>>> np.allclose(R, R_gpu.get(), 1e-4)
True
```

skcuda.linalg.scale

skcuda.linalg.**scale**(*alpha*, *x_gpu*, *alpha_real=False*, *handle=None*)

Scale a vector by a factor alpha.

Parameters

- **alpha** (*scalar*) – Scale parameter
- **x_gpu** (*pycuda.gpuarray.GPUArray*) – Input array.
- **alpha_real** (*bool*) – If *True* and *x_gpu* is complex, then one of the specialized versions *cublasCsscal* or *cublasZdscal* is used which might improve performance for large arrays. (By default, *alpha* is coerced to the corresponding complex type.)
- **handle** (*int*) – CUBLAS context. If no context is specified, the default handle from *skcuda.misc._global_cublas_handle* is used.

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> import skcuda.linalg as linalg
>>> linalg.init()
>>> x = np.asarray(np.random.rand(4, 4), np.float32)
>>> x_gpu = gpuarray.to_gpu(x)
>>> alpha = 2.4
>>> linalg.scale(alpha, x_gpu)
>>> np.allclose(x_gpu.get(), alpha*x)
True
```

skcuda.linalg.svd

skcuda.linalg.**svd**(*a_gpu*, *jobu='A'*, *jobvt='A'*, *lib='cusolver'*)

Singular Value Decomposition.

Factors the matrix *a* into two unitary matrices, *u* and *vh*, and a 1-dimensional array of real, non-negative singular values, *s*, such that *a == dot(u.T, dot(diag(s), vh.T))*.

Parameters

- **a** (*pycuda.gpuarray.GPUArray*) – Input matrix of shape *(m, n)* to decompose.
- **jobu** (*{'A', 'S', 'O', 'N'}*) – If 'A', return the full *u* matrix with shape *(m, m)*. If 'S', return the *u* matrix with shape *(m, k)*. If 'O', return the *u* matrix with shape *(m, k) without allocating a new matrix. If 'N', don't return 'u*.
- **jobvt** (*{'A', 'S', 'O', 'N'}*) – If 'A', return the full *vh* matrix with shape *(n, n)*. If 'S', return the *vh* matrix with shape *(k, n)*. If 'O', return the *vh* matrix with shape *(k, n) without allocating a new matrix. If 'N', don't return 'vh*.
- **lib** (*str*) – Library to use. May be either 'cula' or 'cusolver'.

Returns

- **u** (*pycuda.gpuarray.GPUArray*) – Unitary matrix of shape *(m, m)* or *(m, k)* depending on value of *jobu*.

- **s** (*pycuda.gpuarray.GPUArray*) – Array containing the singular values, sorted such that *s[i] >= s[i+1]*. *s* is of length *min(m, n)*.
- **vh** (*pycuda.gpuarray.GPUArray*) – Unitary matrix of shape *(n, n)* or *(k, n)*, depending on *jobvt*.

Notes

If using CULA, double precision is only supported if the standard version of the CULA Dense toolkit is installed.

This function destroys the contents of the input matrix regardless of the values of *jobu* and *jobvt*.

Only one of *jobu* or *jobvt* may be set to *O*, and then only for a square matrix.

The CUSOLVER library in CUDA 7.0 only supports *jobu* == *jobvt* == 'A'.

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> import skcuda.linalg as linalg
>>> linalg.init()
>>> a = np.random.randn(9, 6) + 1j*np.random.randn(9, 6)
>>> a = np.asarray(a, np.complex64)
>>> a_gpu = gpuarray.to_gpu(a)
>>> u_gpu, s_gpu, vh_gpu = linalg.svd(a_gpu, 'S', 'S')
>>> np.allclose(a, np.dot(u_gpu.get(), np.dot(np.diag(s_gpu.get()), vh_gpu.
↪get())), 1e-4)
True
```

skcuda.linalg.PCA

class skcuda.linalg.**PCA**(*n_components=None*, *handle=None*, *epsilon=1e-07*, *max_iter=10000*)

Principal Component Analysis with similar API to sklearn.decomposition.PCA

The algorithm implemented here was first implemented with cuda in [Andrecut, 2008]. It performs nonlinear dimensionality reduction for a data matrix, mapping the data to a lower dimensional space of K. See references for more information.

Parameters

- **n_components** (*int, default=None*) – The number of principal component column vectors to compute in the output matrix.
- **epsilon** (*float, default=1e-7*) – The maximum error tolerance for eigen value approximation.
- **max_iter** (*int, default=10000*) – The maximum number of iterations in approximating each eigenvalue.

Notes

If n_components is None, then for a NxP data matrix *K = min(N, P)*. Otherwise, *K = min(n_components, N, P)*

References

[Andrecut, 2008]

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> import skcuda.linalg as linalg
>>> from skcuda.linalg import PCA as cuPCA
>>> pca = cuPCA(n_components=4) # map the data to 4 dimensions
>>> X = np.random.rand(1000,100) # 1000 samples of 100-dimensional data vectors
>>> X_gpu = gpuarray.GPUArray((1000,100), np.float64, order="F") # note that␣
↪order="F" or a transpose is necessary. fit_transform requires row-major␣
↪matrices, and column-major is the default
>>> X_gpu.set(X) # copy data to gpu
>>> T_gpu = pca.fit_transform(X_gpu) # calculate the principal components
>>> linalg.dot(T_gpu[:,0], T_gpu[:,1]) # show that the resulting eigenvectors are␣
↪orthogonal
0.0
```

__init__(*n_components=None*, *handle=None*, *epsilon=1e-07*, *max_iter=10000*)
x.__init__(...) initializes x; see help(type(x)) for signature

Methods

__init__([n_components, handle, epsilon, ...])	x.__init__(...) initializes x; see help(type(x)) for signature
fit_transform(X_gpu)	Fit the Principal Component Analysis model, and return the dimension-reduced matrix.
get_n_components()	n_components getter.
set_n_components(n_components)	n_components setter.

skcuda.linalg.trace

skcuda.linalg.**trace**(*x_gpu*, *handle=None*)
Return the sum along the main diagonal of the array.

Parameters

- **x_gpu** (*pycuda.gpuarray.GPUArray*) – Matrix to calculate the trace of.
- **handle** (*int*) – CUBLAS context. If no context is specified, the default handle from *skcuda.misc._global_cublas_handle* is used.

Returns **trace** – trace of x_gpu

Return type number

skcuda.linalg.transpose

skcuda.linalg.**transpose**(*a_gpu*, *handle=None*)

Matrix transpose.

Transpose a matrix in device memory and return an object representing the transposed matrix.

Parameters **a_gpu** (*pycuda.gpuarray.GPUArray*) – Input matrix of shape *(m, n)*.

Returns

- **at_gpu** (*pycuda.gpuarray.GPUArray*) – Transposed matrix of shape *(n, m)*.
- **handle** (*int*) – CUBLAS context. If no context is specified, the default handle from *skcuda.misc._global_cublas_handle* is used.

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.driver as drv
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> import skcuda.linalg as linalg
>>> linalg.init()
>>> a = np.array([[1, 2, 3, 4, 5, 6], [7, 8, 9, 10, 11, 12]], np.float32)
>>> a_gpu = gpuarray.to_gpu(a)
>>> at_gpu = linalg.transpose(a_gpu)
>>> np.all(a.T == at_gpu.get())
True
>>> b = np.array([[1j, 2j, 3j, 4j, 5j, 6j], [7j, 8j, 9j, 10j, 11j, 12j]], np.
↪complex64)
>>> b_gpu = gpuarray.to_gpu(b)
>>> bt_gpu = linalg.transpose(b_gpu)
>>> np.all(b.T == bt_gpu.get())
True
```

skcuda.linalg.tril

skcuda.linalg.**tril**(*a_gpu*, *overwrite=False*, *handle=None*)

Lower triangle of a matrix.

Return the lower triangle of a square matrix.

Parameters

- **a_gpu** (*pycuda.gpuarray.GPUArray*) – Input matrix of shape *(m, m)*
- **overwrite** (*bool (default: False)*) – If true, zero out the upper triangle of the matrix. If false, return the result in a newly allocated matrix.
- **handle** (*int*) – CUBLAS context. If no context is specified, the default handle from *skcuda.misc._global_cublas_handle* is used.

Returns **l_gpu** – The lower triangle of the original matrix.

Return type pycuda.gpuarray

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.driver as drv
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> import skcuda.linalg as linalg
>>> linalg.init()
>>> a = np.asarray(np.random.rand(4, 4), np.float32)
>>> a_gpu = gpuarray.to_gpu(a)
>>> l_gpu = linalg.tril(a_gpu, False)
>>> np.allclose(np.tril(a), l_gpu.get())
True
```

skcuda.linalg.triu

skcuda.linalg.**triu**(*a_gpu*, *k=0*, *overwrite=False*, *handle=None*)

Upper triangle of a matrix.

Return the upper triangle of a square matrix.

Parameters

- **a_gpu** (*pycuda.gpuarray.GPUArray*) – Input matrix of shape *(m, m)*
- **overwrite** (*bool (default: False)*) – If true, zero out the lower triangle of the matrix. If false, return the result in a newly allocated matrix.
- **handle** (*int*) – CUBLAS context. If no context is specified, the default handle from *skcuda.misc._global_cublas_handle* is used.

Returns **u_gpu** – The upper triangle of the original matrix.

Return type pycuda.gpuarray

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.driver as drv
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> import skcuda.linalg as linalg
>>> linalg.init()
>>> a = np.asarray(np.random.rand(4, 4), np.float32)
>>> a_gpu = gpuarray.to_gpu(a)
>>> u_gpu = linalg.triu(a_gpu, False)
>>> np.allclose(np.triu(a), u_gpu.get())
True
```

skcuda.linalg.vander

skcuda.linalg.**vander**(*a_gpu*, *n=None*, *handle=None*)

Generate a Vandermonde matrix.

A Vandermonde matrix (named for Alexandre- Theophile Vandermonde) is a matrix where the columns are powers of the input vector, i.e., the *i-th* column is the input vector raised element-wise to the power of *i*.

Parameters

- **a_gpu** (*pycuda.gpuarray.GPUArray*) – Real/complex 1-D input array of shape *(m, 1)*.
- **n** (*int, optional*) – Number of columns in the Vandermonde matrix. If *n* is not specified, a square array is returned *(m,m)*.

Returns **vander_gpu** – Vandermonde matrix of shape *(m,n)*.

Return type pycuda.gpuarray

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import numpy as np
>>> import skcuda.linalg as linalg
>>> a = np.array(np.array([1, 2, 3]), np.float32, order='F')
>>> a_gpu = gpuarray.to_gpu(a)
>>> v_gpu = linalg.vander(a_gpu, n=4)
>>> np.allclose(v_gpu.get(), np.fliplr(np.vander(a, 4)), atol=1e-6)
True
```

Randomized Linear Algebra Routines

rdmd	Randomized Dynamic Mode Decomposition.
cdmd	Compressed Dynamic Mode Decomposition.
rsvd	Randomized Singular Value Decomposition.

skcuda.rlinalg.rdmd

skcuda.rlinalg.**rdmd**(*a_gpu*, *k=None*, *p=5*, *q=1*, *modes='exact'*, *method_rsvd='standard'*, *return_amplitudes=False*, *return_vandermonde=False*, *handle=None*)

Randomized Dynamic Mode Decomposition.

Dynamic Mode Decomposition (DMD) is a data processing algorithm which allows to decompose a matrix *a* in space and time. The matrix *a* is decomposed as *a = FBV*, where the columns of *F* contain the dynamic modes. The modes are ordered corresponding to the amplitudes stored in the diagonal matrix *B*. *V* is a Vandermonde matrix describing the temporal evolution.

Parameters

- **a_gpu** (*pycuda.gpuarray.GPUArray*) – Real/complex input matrix *a* with dimensions *(m, n)*.
- **k** (*int, optional*) – If *k < (n-1)* low-rank Dynamic Mode Decomposition is computed.
- **p** (*int*) – *p* sets the oversampling parameter for rSVD (default k=5).
- **q** (*int*) – *q* sets the number of power iterations for rSVD (default=1).

- **modes** (*{'standard', 'exact'}*) –

  'standard' [uses the standard definition to compute the dynamic modes,] *F = U * W*.

  'exact' : computes the exact dynamic modes, *F = Y * V * (S**-1) * W*.

- **method_rsvd** (*{'standard', 'fast'}*) – 'standard' : (default) Standard algorithm as described in [1, 2] 'fast' : Version II algorithm as described in [2]
- **return_amplitudes** (bool *{True, False}*) – True: return amplitudes in addition to dynamic modes.
- **return_vandermonde** (bool *{True, False}*) – True: return Vandermonde matrix in addition to dynamic modes and amplitudes.
- **handle** (*int*) – CUBLAS context. If no context is specified, the default handle from *skcuda.misc._global_cublas_handle* is used.

Returns

- **f_gpu** (*pycuda.gpuarray.GPUArray*) – Matrix containing the dynamic modes of shape *(m, n-1)* or *(m, k)*.
- **b_gpu** (*pycuda.gpuarray.GPUArray*) – 1-D array containing the amplitudes of length *min(n-1, k)*.
- **v_gpu** (*pycuda.gpuarray.GPUArray*) – Vandermonde matrix of shape *(n-1, n-1)* or *(k, n-1)*.

Notes

Double precision is only supported if the standard version of the CULA Dense toolkit is installed.

This function destroys the contents of the input matrix.

Arrays are assumed to be stored in column-major order, i.e., order='F'.

References

N. B. Erichson and C. Donovan. "Randomized Low-Rank Dynamic Mode Decomposition for Motion Detection" Under Review.

N. Halko, P. Martinsson, and J. Tropp. "Finding structure with randomness: probabilistic algorithms for constructing approximate matrix decompositions" (2009). (available at arXiv).

J. H. Tu, et al. "On dynamic mode decomposition: theory and applications." arXiv preprint arXiv:1312.0041 (2013).

Examples

```
>>> #Numpy
>>> import numpy as np
>>> #Plot libs
>>> import matplotlib.pyplot as plt
>>> from mpl_toolkits.mplot3d import Axes3D
>>> from matplotlib import cm
```

(continues on next page)

(continued from previous page)

```
>>> #GPU DMD libs
>>> import pycuda.gpuarray as gpuarray
>>> import pycuda.autoinit
>>> from skcuda import linalg, rlinalg
>>> linalg.init()
>>> rlinalg.init()
```

```
>>> # Define time and space discretizations
>>> x=np.linspace( -15, 15, 200)
>>> t=np.linspace(0, 8*np.pi , 80)
>>> dt=t[2]-t[1]
>>> X, T = np.meshgrid(x,t)
>>> # Create two patio-temporal patterns
>>> F1 = 0.5* np.cos(X)*(1.+0.* T)
>>> F2 = ( (1./np.cosh(X)) * np.tanh(X)) *(2.*np.exp(1j*2.8*T))
>>> # Add both signals
>>> F = (F1+F2)
```

```
>>> #Plot dataset
>>> fig = plt.figure()
>>> ax = fig.add_subplot(231, projection='3d')
>>> ax = fig.gca(projection='3d')
>>> surf = ax.plot_surface(X, T, F, rstride=1, cstride=1, cmap=cm.coolwarm,␣
↪linewidth=0, antialiased=True)
>>> ax.set_zlim(-1, 1)
>>> plt.title('F')
>>> ax = fig.add_subplot(232, projection='3d')
>>> ax = fig.gca(projection='3d')
>>> surf = ax.plot_surface(X, T, F1, rstride=1, cstride=1, cmap=cm.coolwarm,␣
↪linewidth=0, antialiased=False)
>>> ax.set_zlim(-1, 1)
>>> plt.title('F1')
>>> ax = fig.add_subplot(233, projection='3d')
>>> ax = fig.gca(projection='3d')
>>> surf = ax.plot_surface(X, T, F2, rstride=1, cstride=1, cmap=cm.coolwarm,␣
↪linewidth=0, antialiased=False)
>>> ax.set_zlim(-1, 1)
>>> plt.title('F2')
```

```
>>> #Dynamic Mode Decomposition
>>> F_gpu = np.array(F.T, np.complex64, order='F')
>>> F_gpu = gpuarray.to_gpu(F_gpu)
>>> Fmodes_gpu, b_gpu, V_gpu, omega_gpu = rlinalg.rdmd(F_gpu, k=2, p=0, q=1,␣
↪modes='exact', return_amplitudes=True, return_vandermonde=True)
>>> omega = omega_gpu.get()
>>> plt.scatter(omega.real, omega.imag, marker='o', c='r')
```

```
>>> #Recover original signal
>>> F1tilde = np.dot(Fmodes_gpu[:,0:1].get() , np.dot(b_gpu[0].get(), V_gpu[0:1,
↪:].get() ) )
>>> F2tilde = np.dot(Fmodes_gpu[:,1:2].get() , np.dot(b_gpu[1].get(), V_gpu[1:2,
↪:].get() ) )
```

```
>>> #Plot DMD modes
>>> #Mode 0
```

(continues on next page)

(continued from previous page)

```
>>> ax = fig.add_subplot(235, projection='3d')
>>> ax = fig.gca(projection='3d')
>>> surf = ax.plot_surface(X[0:F1tilde.shape[1],:], T[0:F1tilde.shape[1],:],␣
↪F1tilde.T, rstride=1, cstride=1, cmap=cm.coolwarm, linewidth=0,␣
↪antialiased=False)
>>> ax.set_zlim(-1, 1)
>>> plt.title('F1_tilde')
>>> #Mode 1
>>> ax = fig.add_subplot(236, projection='3d')
>>> ax = fig.gca(projection='3d')
>>> surf = ax.plot_surface(X[0:F2tilde.shape[1],:], T[0:F2tilde.shape[1],:],␣
↪F2tilde.T, rstride=1, cstride=1, cmap=cm.coolwarm, linewidth=0,␣
↪antialiased=False)
>>> ax.set_zlim(-1, 1)
>>> plt.title('F2_tilde')
>>> plt.show()
```

skcuda.rlinalg.cdmd

skcuda.rlinalg.**cdmd**(*a_gpu*, *k=None*, *c=None*, *modes='exact'*, *return_amplitudes=False*, *return_vandermonde=False*, *handle=None*)

Compressed Dynamic Mode Decomposition.

Dynamic Mode Decomposition (DMD) is a data processing algorithm which allows to decompose a matrix *a* in space and time. The matrix *a* is decomposed as *a = FBV*, where the columns of *F* contain the dynamic modes. The modes are ordered corresponding to the amplitudes stored in the diagonal matrix *B*. *V* is a Vandermonde matrix describing the temporal evolution.

Parameters

- **a_gpu** (*pycuda.gpuarray.GPUArray*) – Real/complex input matrix *a* with dimensions *(m, n)*.
- **k** (*int, optional*) – If *k < (n-1)* low-rank Dynamic Mode Decomposition is computed.
- **c** (*int*) – *p* sets the number of measurements sensors.
- **modes** (*{'exact'}*) – 'exact' : computes the exact dynamic modes, *F = Y * V * (S**-1) * W*.
- **return_amplitudes** (bool *{True, False}*) – True: return amplitudes in addition to dynamic modes.
- **return_vandermonde** (bool *{True, False}*) – True: return Vandermonde matrix in addition to dynamic modes and amplitudes.
- **handle** (*int*) – CUBLAS context. If no context is specified, the default handle from *skcuda.misc._global_cublas_handle* is used.

Returns

- **f_gpu** (*pycuda.gpuarray.GPUArray*) – Matrix containing the dynamic modes of shape *(m, n-1)* or *(m, k)*.
- **b_gpu** (*pycuda.gpuarray.GPUArray*) – 1-D array containing the amplitudes of length *min(n-1, k)*.

- **v_gpu** (*pycuda.gpuarray.GPUArray*) – Vandermonde matrix of shape *(n-1, n-1)* or *(k, n-1)*.

Notes

Double precision is only supported if the standard version of the CULA Dense toolkit is installed.

This function destroys the contents of the input matrix.

Arrays are assumed to be stored in column-major order, i.e., order='F'.

References

S. L. Brunton, et al. "Compressed sampling and dynamic mode decomposition." arXiv preprint arXiv:1312.5186 (2013).

J. H. Tu, et al. "On dynamic mode decomposition: theory and applications." arXiv preprint arXiv:1312.0041 (2013).

Examples

```
>>> #Numpy
>>> import numpy as np
>>> #Plot libs
>>> import matplotlib.pyplot as plt
>>> from mpl_toolkits.mplot3d import Axes3D
>>> from matplotlib import cm
>>> #GPU DMD libs
>>> import pycuda.gpuarray as gpuarray
>>> import pycuda.autoinit
>>> from skcuda import linalg, rlinalg
>>> linalg.init()
>>> rlinalg.init()
```

```
>>> # Define time and space discretizations
>>> x=np.linspace( -15, 15, 200)
>>> t=np.linspace(0, 8*np.pi , 80)
>>> dt=t[2]-t[1]
>>> X, T = np.meshgrid(x,t)
>>> # Create two patio-temporal patterns
>>> F1 = 0.5* np.cos(X)*(1.+0.* T)
>>> F2 = ( (1./np.cosh(X)) * np.tanh(X)) *(2.*np.exp(1j*2.8*T))
>>> # Add both signals
>>> F = (F1+F2)
```

```
>>> #Plot dataset
>>> fig = plt.figure()
>>> ax = fig.add_subplot(231, projection='3d')
>>> ax = fig.gca(projection='3d')
>>> surf = ax.plot_surface(X, T, F, rstride=1, cstride=1, cmap=cm.coolwarm,␣
↪linewidth=0, antialiased=True)
>>> ax.set_zlim(-1, 1)
>>> plt.title('F')
```

(continues on next page)

(continued from previous page)

```
>>> ax = fig.add_subplot(232, projection='3d')
>>> ax = fig.gca(projection='3d')
>>> surf = ax.plot_surface(X, T, F1, rstride=1, cstride=1, cmap=cm.coolwarm,␣
↪linewidth=0, antialiased=False)
>>> ax.set_zlim(-1, 1)
>>> plt.title('F1')
>>> ax = fig.add_subplot(233, projection='3d')
>>> ax = fig.gca(projection='3d')
>>> surf = ax.plot_surface(X, T, F2, rstride=1, cstride=1, cmap=cm.coolwarm,␣
↪linewidth=0, antialiased=False)
>>> ax.set_zlim(-1, 1)
>>> plt.title('F2')
```

```
>>> #Dynamic Mode Decomposition
>>> F_gpu = np.array(F.T, np.complex64, order='F')
>>> F_gpu = gpuarray.to_gpu(F_gpu)
>>> Fmodes_gpu, b_gpu, V_gpu, omega_gpu = rlinalg.cdmd(F_gpu, k=2, c=20, modes=
↪'exact', return_amplitudes=True, return_vandermonde=True)
>>> omega = omega_gpu.get()
>>> plt.scatter(omega.real, omega.imag, marker='o', c='r')
```

```
>>> #Recover original signal
>>> F1tilde = np.dot(Fmodes_gpu[:,0:1].get() , np.dot(b_gpu[0].get(), V_gpu[0:1,
↪:].get() ) )
>>> F2tilde = np.dot(Fmodes_gpu[:,1:2].get() , np.dot(b_gpu[1].get(), V_gpu[1:2,
↪:].get() ) )
```

```
>>> # Plot DMD modes
>>> #Mode 0
>>> ax = fig.add_subplot(235, projection='3d')
>>> ax = fig.gca(projection='3d')
>>> surf = ax.plot_surface(X[0:F1tilde.shape[1],:], T[0:F1tilde.shape[1],:],␣
↪F1tilde.T, rstride=1, cstride=1, cmap=cm.coolwarm, linewidth=0,␣
↪antialiased=False)
>>> ax.set_zlim(-1, 1)
>>> plt.title('F1_tilde')
>>> #Mode 1
>>> ax = fig.add_subplot(236, projection='3d')
>>> ax = fig.gca(projection='3d')
>>> surf = ax.plot_surface(X[0:F2tilde.shape[1],:], T[0:F2tilde.shape[1],:],␣
↪F2tilde.T, rstride=1, cstride=1, cmap=cm.coolwarm, linewidth=0,␣
↪antialiased=False)
>>> ax.set_zlim(-1, 1)
>>> plt.title('F2_tilde')
>>> plt.show()
```

skcuda.rlinalg.rsvd

skcuda.rlinalg.**rsvd**(*a_gpu*, *k=None*, *p=0*, *q=0*, *method='standard'*, *handle=None*)

Randomized Singular Value Decomposition.

Randomized algorithm for computing the approximate low-rank singular value decomposition of a rectangular (m, n) matrix *a* with target rank *k << n*. The input matrix a is factored as *a = U * diag(s) * Vt*. The right singluar vectors are the columns of the real or complex unitary matrix *U*. The left

singular vectors are the columns of the real or complex unitary matrix *V*. The singular values *s* are non-negative and real numbers.

The paramter *p* is a oversampling parameter to improve the approximation. A value between 2 and 10 is recommended.

The paramter *q* specifies the number of normlized power iterations (subspace iterations) to reduce the approximation error. This is recommended if the the singular values decay slowly and in practice 1 or 2 iterations achive good results. However, computing power iterations is increasing the computational time.

If k > (n/1.5), partial SVD or trancated SVD might be faster.

Parameters

- **a_gpu** (*pycuda.gpuarray.GPUArray*) – Real/complex input matrix *a* with dimensions *(m, n)*.
- **k** (*int*) – *k* is the target rank of the low-rank decomposition, k << min(m,n).
- **p** (*int*) – *p* sets the oversampling parameter (default k=0).
- **q** (*int*) – *q* sets the number of power iterations (default=0).
- **method** (*{'standard', 'fast'}*) – 'standard' : Standard algorithm as described in [1, 2] 'fast' : Version II algorithm as described in [2]
- **handle** (*int*) – CUBLAS context. If no context is specified, the default handle from *skcuda.misc._global_cublas_handle* is used.

Returns

- **u_gpu** (*pycuda.gpuarray*) – Right singular values, array of shape *(m, k)*.
- **s_gpu** (*pycuda.gpuarray*) – Singular values, 1-d array of length *k*.
- **vt_gpu** (*pycuda.gpuarray*) – Left singular values, array of shape *(k, n)*.

Notes

Double precision is only supported if the standard version of the CULA Dense toolkit is installed.

This function destroys the contents of the input matrix.

Arrays are assumed to be stored in column-major order, i.e., order='F'.

Input matrix of shape *(m, n)*, where *n>m* is not supported yet.

References

N. Halko, P. Martinsson, and J. Tropp. "Finding structure with randomness: probabilistic algorithms for constructing approximate matrix decompositions" (2009). (available at arXiv).

S. Voronin and P.Martinsson. "RSVDPACK: Subroutines for computing partial singular value decompositions via randomized sampling on single core, multi core, and GPU architectures" (2015). (available at arXiv).

Examples

```
>>> import pycuda.gpuarray as gpuarray
>>> import pycuda.autoinit
>>> import numpy as np
>>> from skcuda import linalg, rlinalg
>>> linalg.init()
>>> rlinalg.init()
```

```
>>> #Randomized SVD decomposition of the square matrix `a` with single precision.
>>> #Note: There is no gain to use rsvd if k > int(n/1.5)
>>> a = np.array(np.random.randn(5, 5), np.float32, order='F')
>>> a_gpu = gpuarray.to_gpu(a)
>>> U, s, Vt = rlinalg.rsvd(a_gpu, k=5, method='standard')
>>> np.allclose(a, np.dot(U.get(), np.dot(np.diag(s.get()), Vt.get())), 1e-4)
True
```

```
>>> #Low-rank SVD decomposition with target rank k=2
>>> a = np.array(np.random.randn(5, 5), np.float32, order='F')
>>> a_gpu = gpuarray.to_gpu(a)
>>> U, s, Vt = rlinalg.rsvd(a_gpu, k=2, method='standard')
```

Special Math Functions

exp1	Exponential integral with *n = 1* of complex arguments.
expi	Exponential integral of complex arguments.
sici	Sine/Cosine integral.

skcuda.special.exp1

skcuda.special.**exp1**(*z_gpu*)

Exponential integral with *n = 1* of complex arguments.

Parameters **z_gpu** (*GPUArray*) – Input matrix of shape *(m, n)*.

Returns **e_gpu** – GPUarrays containing the exponential integrals of the entries of *z_gpu*.

Return type GPUArray

Examples

```
>>> import pycuda.gpuarray as gpuarray
>>> import pycuda.autoinit
>>> import numpy as np
>>> import scipy.special
>>> import special
>>> z = np.asarray(np.random.rand(4, 4)+1j*np.random.rand(4, 4), np.complex64)
>>> z_gpu = gpuarray.to_gpu(z)
>>> e_gpu = exp1(z_gpu)
>>> e_sp = scipy.special.exp1(z)
```

(continues on next page)

(continued from previous page)

```
>>> np.allclose(e_sp, e_gpu.get())
True
```

skcuda.special.expi

skcuda.special.**expi**(*z_gpu*)

Exponential integral of complex arguments.

Parameters **z_gpu** (*GPUArray*) – Input matrix of shape *(m, n)*.

Returns **e_gpu** – GPUarrays containing the exponential integrals of the entries of *z_gpu*.

Return type GPUArray

Examples

```
>>> import pycuda.gpuarray as gpuarray
>>> import pycuda.autoinit
>>> import numpy as np
>>> import scipy.special
>>> import special
>>> z = np.asarray(np.random.rand(4, 4)+1j*np.random.rand(4, 4), np.complex64)
>>> z_gpu = gpuarray.to_gpu(z)
>>> e_gpu = expi(z_gpu)
>>> e_sp = scipy.special.expi(z)
>>> np.allclose(e_sp, e_gpu.get())
True
```

skcuda.special.sici

skcuda.special.**sici**(*x_gpu*)

Sine/Cosine integral.

Computes the sine and cosine integral of every element in the input matrix.

Parameters **x_gpu** (*GPUArray*) – Input matrix of shape *(m, n)*.

Returns **(si_gpu, ci_gpu)** – Tuple of GPUarrays containing the sine integrals and cosine integrals of the entries of *x_gpu*.

Return type tuple of GPUArrays

Examples

```
>>> import pycuda.gpuarray as gpuarray
>>> import pycuda.autoinit
>>> import numpy as np
>>> import scipy.special
>>> import special
>>> x = np.array([[1, 2], [3, 4]], np.float32)
>>> x_gpu = gpuarray.to_gpu(x)
```

(continues on next page)

(continued from previous page)

```
>>> (si_gpu, ci_gpu) = sici(x_gpu)
>>> (si, ci) = scipy.special.sici(x)
>>> np.allclose(si, si_gpu.get())
True
>>> np.allclose(ci, ci_gpu.get())
True
```

1.2.3 Other Routines

Miscellaneous Routines

add	Adds two scalars, vectors, or matrices.
add_matvec	Adds a vector to each column/row of the matrix.
argmax	Indices of the maximum values along an axis.
argmin	Indices of the minimum values along an axis.
cumsum	Cumulative sum.
diff	Calculate the discrete difference.
div_matvec	Divides each column/row of a matrix by a vector.
divide	Divides two scalars, vectors, or matrices with broadcasting.
done_context	Detach from a context cleanly.
get_by_index	Get values in a GPUArray by index.
get_compute_capability	Get the compute capability of the specified device.
get_current_device	Get the device in use by the current context.
get_dev_attrs	
inf	Return an array of the given shape and dtype filled with infs.
init	Initialize libraries used by scikit-cuda.
init_context	Create a context that will be cleaned up properly.
init_device	Initialize a GPU device.
iscomplextype	Check whether a type is complex.
isdoubletype	Check whether a type has double precision.
max	Return the maximum of an array or maximum along an axis.
maxabs	Get maximum absolute value.
mean	Compute the arithmetic means along the specified axis.
min	Return the minimum of an array or minimum along an axis.
mult_matvec	Multiplies a vector elementwise with each column/row of the matrix.
multiply	Multiplies two scalars, vectors, or matrices with broadcasting.
ones	Return an array of the given shape and dtype filled with ones.
ones_like	Return an array of ones with the same shape and type as a given array.
select_block_grid_sizes	
set_by_index	Set values in a GPUArray by index.

Continued on next page

Table 43 – continued from previous page

set_realloc	Transfer data into a GPUArray instance.
shutdown	Shutdown libraries used by scikit-cuda.
std	Compute the standard deviation along the specified axis.
subtract	Subtracts two scalars, vectors, or matrices with broadcasting.
sum	Compute the sum along the specified axis.
var	Compute the variance along the specified axis.
zeros	Return an array of the given shape and dtype filled with zeros.
zeros_like	Return an array of zeros with the same shape and type as a given array.

skcuda.misc.add

skcuda.misc.**add**(*x_gpu*, *y_gpu*)

Adds two scalars, vectors, or matrices.

The numpy broadcasting rules apply so this would yield the same result as *x_gpu.get()* + *y_gpu.get()* in host code.

Parameters **y_gpu** (*x_gpu,*) – The arrays to be added.

Returns **out** – Equivalent to *x_gpu.get()* + *y_gpu.get()*.

Return type pycuda.gpuarray.GPUArray

Notes

The *out* and *stream* options are not supported because *GPUArray.__add__* doesn't provide them.

skcuda.misc.add_matvec

skcuda.misc.**add_matvec**(*x_gpu*, *a_gpu*, *axis=None*, *out=None*, *stream=None*)

Adds a vector to each column/row of the matrix.

The numpy broadcasting rules apply so this would yield the same result as *x_gpu.get()* + *a_gpu.get()* in host-code.

Parameters

- **x_gpu** (*pycuda.gpuarray.GPUArray*) – Matrix to which to add the vector.
- **a_gpu** (*pycuda.gpuarray.GPUArray*) – Vector to add to *x_gpu*.
- **axis** (*int (optional)*) – The axis onto which the vector is added. By default this is determined automatically by using the first axis with the correct dimensionality.
- **out** (*pycuda.gpuarray.GPUArray (optional)*) – Optional destination matrix.
- **stream** (*pycuda.driver.Stream (optional)*) – Optional Stream in which to perform this calculation.

Returns **out** – Result of *x_gpu* + *a_gpu*

Return type pycuda.gpuarray.GPUArray

skcuda.misc.argmax

skcuda.misc.**argmax**(*a_gpu*, *axis*, *keepdims=False*)

Indices of the maximum values along an axis.

Parameters

- **a_gpu** (*pycuda.gpuarray.GPUArray*) – Input array
- **axis** (*int*) – Axis along which the maxima are computed.
- **keepdims** (*bool (optional, default False)*) – If True, the axes which are reduced are left in the result as dimensions with size one.

Returns **out** – Array of indices into the array.

Return type pycuda.gpuarray.GPUArray

skcuda.misc.argmin

skcuda.misc.**argmin**(*a_gpu*, *axis*, *keepdims=False*)

Indices of the minimum values along an axis.

Parameters

- **a_gpu** (*pycuda.gpuarray.GPUArray*) – Input array
- **axis** (*int*) – Axis along which the minima are computed.
- **keepdims** (*bool (optional, default False)*) – If True, the axes which are reduced are left in the result as dimensions with size one.

Returns **out** – Array of indices into the array.

Return type pycuda.gpuarray.GPUArray

skcuda.misc.cumsum

skcuda.misc.**cumsum**(*x_gpu*)

Cumulative sum.

Return the cumulative sum of the elements in the specified array.

Parameters **x_gpu** (*pycuda.gpuarray.GPUArray*) – Input array.

Returns **c_gpu** – Output array containing cumulative sum of *x_gpu*.

Return type pycuda.gpuarray.GPUArray

Notes

Higher dimensional arrays are implicitly flattened row-wise by this function.

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import misc
>>> x_gpu = gpuarray.to_gpu(np.random.rand(5).astype(np.float32))
>>> c_gpu = misc.cumsum(x_gpu)
>>> np.allclose(c_gpu.get(), np.cumsum(x_gpu.get()))
True
```

skcuda.misc.diff

skcuda.misc.**diff**(*x_gpu*)

Calculate the discrete difference.

Calculates the first order difference between the successive entries of a vector.

Parameters **x_gpu** (*pycuda.gpuarray.GPUArray*) – Input vector.

Returns **y_gpu** – Discrete difference.

Return type pycuda.gpuarray.GPUArray

Examples

```
>>> import pycuda.driver as drv
>>> import pycuda.gpuarray as gpuarray
>>> import pycuda.autoinit
>>> import numpy as np
>>> import misc
>>> x = np.asarray(np.random.rand(5), np.float32)
>>> x_gpu = gpuarray.to_gpu(x)
>>> y_gpu = misc.diff(x_gpu)
>>> np.allclose(np.diff(x), y_gpu.get())
True
```

skcuda.misc.div_matvec

skcuda.misc.**div_matvec**(*x_gpu*, *a_gpu*, *axis=None*, *out=None*, *stream=None*)

Divides each column/row of a matrix by a vector.

The numpy broadcasting rules apply so this would yield the same result as *x_gpu.get()* / *a_gpu.get()* in host-code.

Parameters

- **x_gpu** (*pycuda.gpuarray.GPUArray*) – Matrix to divide by the vector *a_gpu*.
- **a_gpu** (*pycuda.gpuarray.GPUArray*) – The matrix *x_gpu* will be divided by this vector.
- **axis** (*int (optional)*) – The axis on which division occurs. By default this is determined automatically by using the first axis with the correct dimensionality.
- **out** (*pycuda.gpuarray.GPUArray (optional)*) – Optional destination matrix.

- **stream** (*pycuda.driver.Stream (optional)*) – Optional Stream in which to perform this calculation.

Returns **out** – result of *x_gpu* / *a_gpu*

Return type pycuda.gpuarray.GPUArray

skcuda.misc.divide

skcuda.misc.**divide**(*x_gpu*, *y_gpu*)

Divides two scalars, vectors, or matrices with broadcasting.

The numpy broadcasting rules apply so this would yield the same result as *x_gpu.get()* / *y_gpu.get()* in host code.

Parameters **y_gpu** (*x_gpu,*) – The arrays to be divided.

Returns **out** – Equivalent to *x_gpu.get()* / *y_gpu.get()*.

Return type pycuda.gpuarray.GPUArray

Notes

The *out* and *stream* options are not supported because *GPUArray.__div__* doesn't provide them.

skcuda.misc.done_context

skcuda.misc.**done_context**(*ctx*)

Detach from a context cleanly.

Detach from a context and remove its pop() from atexit.

Parameters **ctx** (*pycuda.driver.Context*) – Context from which to detach.

skcuda.misc.get_by_index

skcuda.misc.**get_by_index**(*src_gpu*, *ind*)

Get values in a GPUArray by index.

Parameters

- **src_gpu** (*pycuda.gpuarray.GPUArray*) – GPUArray instance from which to extract values.
- **ind** (*pycuda.gpuarray.GPUArray or numpy.ndarray*) – Array of element indices to set. Must have an integer dtype.

Returns **res_gpu** – GPUArray with length of *ind* and dtype of *src_gpu* containing selected values.

Return type pycuda.gpuarray.GPUArray

Examples

```
>>> import pycuda.gpuarray as gpuarray
>>> import pycuda.autoinit
>>> import numpy as np
>>> import misc
>>> src = np.random.rand(5).astype(np.float32)
>>> src_gpu = gpuarray.to_gpu(src)
>>> ind = gpuarray.to_gpu(np.array([0, 2, 4]))
>>> res_gpu = misc.get_by_index(src_gpu, ind)
>>> np.allclose(res_gpu.get(), src[[0, 2, 4]])
True
```

Notes

Only supports 1D index arrays.

May not be efficient for certain index patterns because of lack of inability to coalesce memory operations.

skcuda.misc.get_compute_capability

skcuda.misc.**get_compute_capability**(*dev*)

Get the compute capability of the specified device.

Retrieve the compute capability of the specified CUDA device and return it as a floating point value.

Parameters **d** (*pycuda.driver.Device*) – Device object to examine.

Returns **c** – Compute capability.

Return type float

skcuda.misc.get_current_device

skcuda.misc.**get_current_device**()

Get the device in use by the current context.

Returns **d** – Device in use by current context.

Return type pycuda.driver.Device

skcuda.misc.get_dev_attrs

misc.**get_dev_attrs**

skcuda.misc.inf

skcuda.misc.**inf**(*shape*, *dtype*, *order='C'*, *allocator=<Mock object>*)

Return an array of the given shape and dtype filled with infs.

Parameters

- **shape** (*tuple*) – Array shape.
- **dtype** (*data-type*) – Data type for the array.
- **order** (*{'C', 'F'}, optional*) – Create array using row-major or column-major format.
- **allocator** (*callable, optional*) – Returns an object that represents the memory allocated for the requested array.

Returns **out** – Array of infs with the given shape, dtype, and order.

Return type pycuda.gpuarray.GPUArray

skcuda.misc.init

skcuda.misc.**init**(*allocator=<Mock object>*)

Initialize libraries used by scikit-cuda.

Initialize the CUBLAS, CULA, CUSOLVER, and MAGMA libraries used by high-level functions provided by scikit-cuda.

Parameters **allocator** (*an allocator used internally by some of the high-level*) – functions.

Notes

This function does not initialize PyCUDA; it uses whatever device and context were initialized in the current host thread.

skcuda.misc.init_context

skcuda.misc.**init_context**(*dev*)

Create a context that will be cleaned up properly.

Create a context on the specified device and register its pop() method with atexit.

Parameters **dev** (*pycuda.driver.Device*) – GPU device.

Returns **ctx** – Created context.

Return type pycuda.driver.Context

skcuda.misc.init_device

skcuda.misc.**init_device**(*n=0*)

Initialize a GPU device.

Initialize a specified GPU device rather than the default device found by *pycuda.autoinit*.

Parameters **n** (*int*) – Device number.

Returns **dev** – Initialized device.

Return type pycuda.driver.Device

skcuda.misc.iscomplextype

skcuda.misc.**iscomplextype**(*x*)

Check whether a type is complex.

Parameters **t** (*numpy float type*) – Type to test.

Returns **result** – Result.

Return type bool

skcuda.misc.isdoubletype

skcuda.misc.**isdoubletype**(*x*)

Check whether a type has double precision.

Parameters **t** (*numpy float type*) – Type to test.

Returns **result** – Result.

Return type bool

skcuda.misc.max

skcuda.misc.**max**(*a_gpu*, *axis=None*, *keepdims=False*)

Return the maximum of an array or maximum along an axis.

Parameters

- **a_gpu** (*pycuda.gpuarray.GPUArray*) – Input array
- **axis** (*int (optional)*) – Axis along which the maxima are computed. The default is to compute the maximum of the flattened array.
- **keepdims** (*bool (optional, default False)*) – If True, the axes which are reduced are left in the result as dimensions with size one.

Returns **out** – maximum of elements, or maxima of elements along the desired axis.

Return type pycuda.gpuarray.GPUArray or float

skcuda.misc.maxabs

skcuda.misc.**maxabs**(*x_gpu*)

Get maximum absolute value.

Find maximum absolute value in the specified array.

Parameters **x_gpu** (*pycuda.gpuarray.GPUArray*) – Input array.

Returns **m_gpu** – Array containing maximum absolute value in *x_gpu*.

Return type pycuda.gpuarray.GPUArray

Examples

```
>>> import pycuda.autoinit
>>> import pycuda.gpuarray as gpuarray
>>> import misc
>>> x_gpu = gpuarray.to_gpu(np.array([-1, 2, -3], np.float32))
>>> m_gpu = misc.maxabs(x_gpu)
>>> np.allclose(m_gpu.get(), 3.0)
True
```

skcuda.misc.mean

skcuda.misc.**mean**(*x_gpu*, *axis=None*, *out=None*, *keepdims=False*)

Compute the arithmetic means along the specified axis.

Parameters

- **x_gpu** (*pycuda.gpuarray.GPUArray*) – Array containing numbers whose mean is desired.
- **axis** (*int (optional)*) – Axis along which the means are computed. The default is to compute the mean of the flattened array.
- **out** (*pycuda.gpuarray.GPUArray (optional)*) – Output array in which to place the result.
- **keepdims** (*bool (optional, default False)*) – If True, the axes which are reduced are left in the result as dimensions with size one.

Returns **out** – mean of elements, or means of elements along the desired axis.

Return type pycuda.gpuarray.GPUArray

skcuda.misc.min

skcuda.misc.**min**(*a_gpu*, *axis=None*, *keepdims=False*)

Return the minimum of an array or minimum along an axis.

Parameters

- **a_gpu** (*pycuda.gpuarray.GPUArray*) – Input array
- **axis** (*int (optional)*) – Axis along which the minima are computed. The default is to compute the minimum of the flattened array.
- **keepdims** (*bool (optional, default False)*) – If True, the axes which are reduced are left in the result as dimensions with size one.

Returns **out** – minimum of elements, or minima of elements along the desired axis.

Return type pycuda.gpuarray.GPUArray or float

skcuda.misc.mult_matvec

skcuda.misc.**mult_matvec**(*x_gpu*, *a_gpu*, *axis=None*, *out=None*, *stream=None*)

Multiplies a vector elementwise with each column/row of the matrix.

The numpy broadcasting rules apply so this would yield the same result as *x_gpu.get()* * *a_gpu.get()* in host-code.

Parameters

- **x_gpu** (*pycuda.gpuarray.GPUArray*) – Matrix to multiply by the vector *a_gpu*.
- **a_gpu** (*pycuda.gpuarray.GPUArray*) – The matrix *x_gpu* will be multiplied by this vector.
- **axis** (*int (optional)*) – The axis on which multiplication occurs. By default this is determined automatically by using the first axis with the correct dimensionality.
- **out** (*pycuda.gpuarray.GPUArray (optional)*) – Optional destination matrix.
- **stream** (*pycuda.driver.Stream (optional)*) – Optional Stream in which to perform this calculation.

Returns **out** – result of *x_gpu* * *a_gpu*

Return type pycuda.gpuarray.GPUArray

skcuda.misc.multiply

skcuda.misc.**multiply**(*x_gpu*, *y_gpu*)

Multiplies two scalars, vectors, or matrices with broadcasting.

The numpy broadcasting rules apply so this would yield the same result as *x_gpu.get()* * *y_gpu.get()* in host code.

Parameters **y_gpu** (*x_gpu,*) – The arrays to be multiplied.

Returns **out** – Equivalent to *x_gpu.get()* * *y_gpu.get()*.

Return type pycuda.gpuarray.GPUArray

Notes

The *out* and *stream* options are not supported because *GPUArray.__mul__* doesn't provide them.

skcuda.misc.ones

skcuda.misc.**ones**(*shape*, *dtype*, *order='C'*, *allocator=<Mock object>*)

Return an array of the given shape and dtype filled with ones.

Parameters

- **shape** (*tuple*) – Array shape.
- **dtype** (*data-type*) – Data type for the array.
- **order** (*{'C', 'F'}, optional*) – Create array using row-major or column-major format.
- **allocator** (*callable, optional*) – Returns an object that represents the memory allocated for the requested array.

Returns **out** – Array of ones with the given shape, dtype, and order.

Return type pycuda.gpuarray.GPUArray

skcuda.misc.ones_like

skcuda.misc.**ones_like**(*a*)

Return an array of ones with the same shape and type as a given array.

Parameters **a** (*array_like*) – The shape and data type of *a* determine the corresponding attributes of the returned array.

Returns **out** – Array of ones with the shape, dtype, and strides of *other*.

Return type pycuda.gpuarray.GPUArray

skcuda.misc.select_block_grid_sizes

misc.**select_block_grid_sizes**

skcuda.misc.set_by_index

skcuda.misc.**set_by_index**(*dest_gpu*, *ind*, *src_gpu*, *ind_which='dest'*)

Set values in a GPUArray by index.

Parameters

- **dest_gpu** (*pycuda.gpuarray.GPUArray*) – GPUArray instance to modify.
- **ind** (*pycuda.gpuarray.GPUArray or numpy.ndarray*) – 1D array of element indices to set. Must have an integer dtype.
- **src_gpu** (*pycuda.gpuarray.GPUArray*) – GPUArray instance from which to set values.
- **ind_which** (*str*) – If set to 'dest', set the elements in *dest_gpu* with indices *ind* to the successive values in *src_gpu*; the lengths of *ind* and *src_gpu* must be equal. If set to 'src', set the successive values in *dest_gpu* to the values in *src_gpu* with indices *ind*; the lengths of *ind* and *dest_gpu* must be equal.

Examples

```
>>> import pycuda.gpuarray as gpuarray
>>> import pycuda.autoinit
>>> import numpy as np
>>> import misc
>>> dest_gpu = gpuarray.to_gpu(np.arange(5, dtype=np.float32))
>>> ind = gpuarray.to_gpu(np.array([0, 2, 4]))
>>> src_gpu = gpuarray.to_gpu(np.array([1, 1, 1], dtype=np.float32))
>>> misc.set_by_index(dest_gpu, ind, src_gpu, 'dest')
>>> np.allclose(dest_gpu.get(), np.array([1, 1, 1, 3, 1], dtype=np.float32))
True
>>> dest_gpu = gpuarray.to_gpu(np.zeros(3, dtype=np.float32))
>>> ind = gpuarray.to_gpu(np.array([0, 2, 4]))
>>> src_gpu = gpuarray.to_gpu(np.arange(5, dtype=np.float32))
>>> misc.set_by_index(dest_gpu, ind, src_gpu)
```

(continues on next page)

(continued from previous page)

```
>>> np.allclose(dest_gpu.get(), np.array([0, 2, 4], dtype=np.float32))
True
```

Notes

Only supports 1D index arrays.

May not be efficient for certain index patterns because of lack of inability to coalesce memory operations.

skcuda.misc.set_realloc

skcuda.misc.**set_realloc**(*x_gpu*, *data*)

Transfer data into a GPUArray instance.

Copies the contents of a numpy array into a GPUArray instance. If the array has a different type or dimensions than the instance, the GPU memory used by the instance is reallocated and the instance updated appropriately.

Parameters

- **x_gpu** (*pycuda.gpuarray.GPUArray*) – GPUArray instance to modify.
- **data** (*numpy.ndarray*) – Array of data to transfer to the GPU.

Examples

```
>>> import pycuda.gpuarray as gpuarray
>>> import pycuda.autoinit
>>> import numpy as np
>>> import misc
>>> x = np.asarray(np.random.rand(5), np.float32)
>>> x_gpu = gpuarray.to_gpu(x)
>>> x = np.asarray(np.random.rand(10, 1), np.float64)
>>> set_realloc(x_gpu, x)
>>> np.allclose(x, x_gpu.get())
True
```

skcuda.misc.shutdown

skcuda.misc.**shutdown**()

Shutdown libraries used by scikit-cuda.

Shutdown the CUBLAS, CULA, CUSOLVER, and MAGMA libraries used by high-level functions provided by scikits-cuda.

Notes

This function does not shutdown PyCUDA.

skcuda.misc.std

skcuda.misc.**std**(*x_gpu*, *ddof=0*, *axis=None*, *stream=None*, *keepdims=False*)

Compute the standard deviation along the specified axis.

Returns the standard deviation of the array elements, a measure of the spread of a distribution. The standard deviation is computed for the flattened array by default, otherwise over the specified axis.

Parameters

- **x_gpu** (*pycuda.gpuarray.GPUArray*) – Array containing numbers whose std is desired.
- **ddof** (*int (optional)*) – "Delta Degrees of Freedom": the divisor used in computing the variance is `N - ddof`, where `N` is the number of elements. Setting `ddof = 1` is equivalent to applying Bessel's correction.
- **axis** (*int (optional)*) – Axis along which the std are computed. The default is to compute the std of the flattened array.
- **stream** (*pycuda.driver.Stream (optional)*) – Optional CUDA stream in which to perform this calculation
- **keepdims** (*bool (optional, default False)*) – If True, the axes which are reduced are left in the result as dimensions with size one.

Returns **out** – std of elements, or stds of elements along the desired axis.

Return type pycuda.gpuarray.GPUArray or float

skcuda.misc.subtract

skcuda.misc.**subtract**(*x_gpu*, *y_gpu*)

Subtracts two scalars, vectors, or matrices with broadcasting.

The numpy broadcasting rules apply so this would yield the same result as *x_gpu.get()* - *y_gpu.get()* in host code.

Parameters **y_gpu** (*x_gpu,*) – The arrays to be subtracted.

Returns **out** – Equivalent to *x_gpu.get()* - *y_gpu.get()*.

Return type pycuda.gpuarray.GPUArray

Notes

The *out* and *stream* options are not supported because *GPUArray.__sub__* doesn't provide them.

skcuda.misc.sum

skcuda.misc.**sum**(*x_gpu*, *axis=None*, *out=None*, *keepdims=False*)

Compute the sum along the specified axis.

Parameters

- **x_gpu** (*pycuda.gpuarray.GPUArray*) – Array containing numbers whose sum is desired.

- **axis** (*int (optional)*) – Axis along which the sums are computed. The default is to compute the sum of the flattened array.
- **out** (*pycuda.gpuarray.GPUArray (optional)*) – Output array in which to place the result.
- **keepdims** (*bool (optional, default False)*) – If True, the axes which are reduced are left in the result as dimensions with size one.

Returns **out** – sum of elements, or sums of elements along the desired axis.

Return type pycuda.gpuarray.GPUArray

skcuda.misc.var

skcuda.misc.**var**(*x_gpu*, *ddof=0*, *axis=None*, *stream=None*, *keepdims=False*)

Compute the variance along the specified axis.

Returns the variance of the array elements, a measure of the spread of a distribution. The variance is computed for the flattened array by default, otherwise over the specified axis.

Parameters

- **x_gpu** (*pycuda.gpuarray.GPUArray*) – Array containing numbers whose variance is desired.
- **ddof** (*int (optional)*) – "Delta Degrees of Freedom": the divisor used in computing the variance is `N - ddof`, where `N` is the number of elements. Setting `ddof = 1` is equivalent to applying Bessel's correction.
- **axis** (*int (optional)*) – Axis along which the variance are computed. The default is to compute the variance of the flattened array.
- **stream** (*pycuda.driver.Stream (optional)*) – Optional CUDA stream in which to perform this calculation
- **keepdims** (*bool (optional, default False)*) – If True, the axes which are reduced are left in the result as dimensions with size one.

Returns **out** – variance of elements, or variances of elements along the desired axis.

Return type pycuda.gpuarray.GPUArray

skcuda.misc.zeros

skcuda.misc.**zeros**(*shape*, *dtype*, *order='C'*, *allocator=<Mock object>*)

Return an array of the given shape and dtype filled with zeros.

Parameters

- **shape** (*tuple*) – Array shape.
- **dtype** (*data-type*) – Data type for the array.
- **order** (*{'C', 'F'}, optional*) – Create array using row-major or column-major format.
- **allocator** (*callable, optional*) – Returns an object that represents the memory allocated for the requested array.

Returns **out** – Array of zeros with the given shape, dtype, and order.

Return type pycuda.gpuarray.GPUArray

Notes

This function exists to work around the following numpy bug that prevents pycuda.gpuarray.zeros() from working properly with complex types in pycuda 2011.1.2: http://projects.scipy.org/numpy/ticket/1898

skcuda.misc.zeros_like

skcuda.misc.**zeros_like**(*a*)

Return an array of zeros with the same shape and type as a given array.

Parameters **a** (*array_like*) – The shape and data type of *a* determine the corresponding attributes of the returned array.

Returns **out** – Array of zeros with the shape, dtype, and strides of *a*.

Return type pycuda.gpuarray.GPUArray

1.3 Authors & Acknowledgments

This software was written and packaged by Lev Givon. Although it depends upon the excellent PyCUDA package by Andreas Klöckner, scikit-cuda is developed independently of PyCUDA.

Special thanks are due to the following parties for their contributions:

- Frédéric Bastien - CUBLAS version detection enhancements.
- Arnaud Bergeron - Fix to prevent LANG from affecting objdump output.
- David Wei Chiang - Improvements to vectorized functions, bug fixes.
- Sander Dieleman - CUBLAS 5 bindings.
- Chris Capdevila - MacOS X library search fix.
- Ben Erichson - QR decomposition, eigenvalue/eigenvector computation, Dynamic Mode Decomposition, randomized linear algebra routines.
- Ying Wei (Daniel) Fan - Kindly permitted reuse of CUBLAS wrapper code in his PARRET Python package.
- Michael M. Forbes - Improved MacOSX compatibility, bug fixes.
- Jacob Frelinger - Various enhancements.
- Tim Klein - Additional MAGMA wrappers.
- Joseph Martinot-Lagarde - Python 3 compatibility improvements.
- Eric Larson - Various enhancements.
- Gregory R. Lee - Enhanced FFT plan creation.
- Bryant Menn - CUSOLVER support for symmetric eigenvalue decomposition.
- Bruce Merry - Support for CUFFT extensible plan API.
- Teodor Mihai Moldovan - CUBLAS 5 bindings.

- Lars Pastewka - FFT tests and FFTW compatibility mode configuration.
- Li Yong Liu - CUBLAS batch wrappers.
- Luke Pfister - Bug fixes.
- Michael Rader - Bug fixes.
- Nate Merrill - PCA module.
- Alex Rubinsteyn - Support for CULA Dense Free R17.
- Xing Shi - Bug fixes.
- Steve Taylor - Cholesky factorization/solve functions.
- Rob Turetsky - Useful feedback.
- Thomas Unterthiner - Additional high-level and wrapper functions.
- Nikul H. Ukani - Additional MAGMA wrappers.
- S. Clarkson - Bug fixes.
- Stefan van der Walt - Bug fixes.
- Feng Wang - Bug reports.
- Alexander Weyman - Simpson's Rule.
- Evgeniy Zheltonozhskiy Complex Hermitian support eigenvalue decomposition.
- Yiyin Zhou - Patches, bug reports, and function wrappers

1.4 License

1.5 Change Log

1.5.1 Release 0.5.2 (under development)

- Prevent exceptions when CULA Dense free is present (#146).
- Fix Python 3 issues with CUSOLVER wrapper functions (#145)
- Add support for using either CUSOLVER or CULA for computing SVD.
- Add support for using either CUSOLVER or CULA for computing determinant.
- Compressed Dynamic Mode Decomposition (enh. by N. Benjamin Erichson).
- Support for CUFFT extensible plan API (enh. by Bruce Merry).
- Wrappers for CUFFT size estimation (enh. by Luke Pfister).
- Wrappers for CUBLAS-XT functions.
- More wrappers for MAGMA functions (enh. by Nikul H. Ukani).
- Python 3 compatibility improvements (enh. by Joseph Martinot-Lagarde).
- Allow specification of order in misc.zeros and misc.ones.
- Preserve strides in misc.zeros_like and misc.ones_like.
- Add support for Cholesky factorization/solving using CUSOLVER (#198).
- Add cholesky() function that zeros out non-factor entries in result (#199).
- Add support for CUDA 8.0 libraries (#171).
- Workaround for libgomp + CUDA 8.0 weirdness (fix by Kevin Flansburg).
- Fix broken matrix-vector dot product (#156).
- Initialize MAGMA before CUSOLVER to prevent internal errors in certain CUSOLVER functions.
- Skip CULA-dependent unit tests when CULA isn't present.
- CUSOLVER support for symmetric eigenvalue decomposition (enh. by Bryant Menn).
- CUSOLVER support for matrix inversion, QR decomposition (#198).
- Prevent objdump output from changing due to environment language (fix by Arnaud Bergeron).
- Fix diag() support for column-major 2D array inputs (#219).
- Use absolute path for skcuda header includes (enh. by S. Clarkson).
- Fix QR issues by reverting fix for #131 and raising PyCUDA version requirement (fix by S. Clarkson).
- More batch CUBLAS wrappers (enh. by Li Yong Liu)
- Numerical integration with Simpson's Rule (enh. by Alexander Weyman)
- Make CUSOLVER default backend for functions that can use either CULA or CUSOLVER.
- Fix CUDA errors that only occur when unit tests are run en masse with nose or setuptools (#257).

1.5.2 Release 0.5.1 - (October 30, 2015)

- More CUSOLVER wrappers.
- Eigenvalue/eigenvector computation (eng. by N. Benjamin Erichson).
- QR decomposition (enh. by N. Benjamin Erichson).
- Improved Windows 10 compatibility (enh. by N. Benjamin Erichson).
- Function for constructing Vandermonde matrix in GPU memory (enh. by N. Benjamin Erichson).
- Standard and randomized Dynamic Mode Decomposition (enh. by N. Benjamin Erichson).
- Randomized linear algebra routines (enh. by N. Benjamin Erichson).
- Add triu function (enh. by N. Benjamin Erichson).
- Support Bessel correction in computation of variance and standard deviation (#143).
- Fix pip installation issues.

1.5.3 Release 0.5.0 - (July 14, 2015)

- Rename package to scikit-cuda.
- Reductions sum, mean, var, std, max, min, argmax, argmin accept keepdims option.
- The same reductions now return a GPUArray instead of ndarray if axis=None.
- Switch to PEP 440 version numbering.
- Replace distribute_setup.py with ez_setup.py.
- Improve support for latest NVIDIA GPUs.
- Direct links to online NVIDIA documentation in CUBLAS, CUFFT wrapper docstrings.
- Add wrappers for CUSOLVER in CUDA 7.0.
- Add skcuda namespace package that contains all modules in scikits.cuda namespace.
- Add more wrappers for CUBLAS 5 functions (enh. by Teodor Moldovan, Sander Dieleman).
- Add support for CULA Dense Free R17 (enh. by Alex Rubinsteyn).
- Memoize elementwise kernel used by ifft scaling (#37).
- Speed up misc.maxabs using reduction and kernel memoization.
- Speed up misc.cumsum using scan and kernel memoization.
- Speed up linalg.conj and misc.diff using elementwise kernel and memoization.
- Speed up special.{sici,exp1,expi} using elementwise kernel and memoization.
- Add wrappers for experimental multi-GPU CULA routines in CULA Dense R14+.
- Use ldconfig to find library paths rather than libdl (#39).
- Fix win32 platform detection.
- Add Cholesky factorization/solve routines (enh. by Steve Taylor).
- Fix Cholesky factorization/solve routines (fix by Thomas Unterthiner).
- Enable dot() function to operate inplace (enh. by Thomas Unterthiner).

- Python 3 compatibility improvements (enh. by Thomas Unterthiner).
- Support for Fortran-order arrays in dot() and cho_solve() (enh. by Thomas Unterthiner)
- CULA-based matrix inversion (enh. by Thomas Unterthiner).
- Add add_diag() function (enh. by Thomas Unterthiner).
- Use cublas*copy in diag() function (enh. by Thomas Unterthiner).
- Improved MacOSX compatibility (enh. by Michael M. Forbes).
- Find CUBLAS version even when it is only accessible via LD_LIBRARY_PATH (enh. by Frédéric Bastien).
- Get both major and minor version numbers from CUBLAS library when determining version.
- Handle unset LD_LIBRARY_PATH variable (fix by Jan Schlüter).
- Fix library search on MacOS X (fix by capdevc).
- Fix library search on Windows.
- Add Windows support to CULA wrappers.
- Enable specification of memory pool allocator to linalg functions (enh. by Thomas Unterthiner).
- Improve misc.select_block_grid_sizes() logic to handle different GPU hardware.
- Compute transpose using CUDA 5.0 CUBLAS functions rather than with inefficient naive kernel.
- Use ReadTheDocs theme when building HTML docs locally.
- Support additional cufftPlanMany() parameters when creating FFT plans (enh. by Gregory R. Lee).
- Improved Python 3.4 compatibility (enh. by Eric Larson).
- Avoid unnecessary import of cublas when importing fft module (enh. by Eric Larson).
- Matrix trace function (enh. by Thomas Unterthiner).
- Functions for computing simple axis-wise stats over matrices (enh. by Thomas Unterthiner).
- Matrix add_dot, add_matvec, div_matvec, mult_matvec functions (enh. by Thomas Unterthiner).
- Faster dot_diag implementation using CUBLAS matrix-matrix multiplication (enh. by Thomas Unterthiner).
- Memoize SourceModule calls to speed up various high-level functions (enh. by Thomas Unterthiner).
- Function for computing matrix determinant (enh. by Thomas Unterthiner).
- Function for computing min/max and argmin/argmax along a matrix axis (enh. by Thomas Unterthiner).
- Set default value of the parameter 'overwrite' to False in all linalg functions.
- Elementwise arithmetic operations with broadcasting up to 2 dimensions (enh. David Wei Chiang)

1.5.4 Release 0.042 - (March 10, 2013)

- Add complex exponential integral.
- Fix typo in cublasCgbmv.
- Use CUBLAS v2 API, add preliminary support for CUBLAS 5 functions.
- Detect CUBLAS version without initializing the GPU.

- Work around numpy bug #1898.
- Fix issues with pycuda installations done via easy_install/pip.
- Add support for specifying streams when creating FFT plans.
- Successfully find CULA R13a libraries.
- Raise exceptions when functions in the full release of CULA Dense are invoked without the library installed.
- Perform post-fft scaling in-place.
- Fix broken Python 2.6 compatibility (#19).
- Download distribute for package installation if it isn't available.
- Prevent absence of CULA from causing import errors (enh. by Jacob Frelinger)
- FFT batch tests and FFTW mode configuration (enh. by Lars Pastewka)

1.5.5 Release 0.041 - (May 22, 2011)

- Fix bug preventing installation with pip.

1.5.6 Release 0.04 - (May 11, 2011)

- Fix bug in cutoff_invert kernel.
- Add get_compute_capability function and other goodies to misc module.
- Use pycuda-complex.hpp to improve kernel readability.
- Add integrate module.
- Add unit tests for high-level functions.
- Automatically determine device used by current context.
- Support batched and multidimensional FFT operations.
- Extended dot() function to support implicit transpose/Hermitian.
- Support for in-place computation of singular vectors in svd() function.
- Simplify kernel launch setup.
- More CULA routine wrappers.
- Wrappers for CULA R11 auxiliary routines.

1.5.7 Release 0.03 - (November 22, 2010)

- Add support for some functions in the premium version of CULA toolkit.
- Add wrappers for all lapack functions in basic CULA toolkit.
- Fix pinv() to properly invert complex matrices.
- Add Hermitian transpose.
- Add tril function.
- Fix missing library detection.

- Include missing CUDA headers in package.

1.5.8 Release 0.02 - (September 21, 2010)

- Add documentation.
- Update copyright information.

1.5.9 Release 0.01 - (September 17, 2010)

- First public release.

CHAPTER 2

Index

- genindex

Index

Symbols

A

C

D

E

G

H

I

M

N

O

P

Q

R

S

T

V

Z