Pyculib Documentation

Release 1.0.1

Anaconda, Inc.

Feb 18, 2019

Contents

High Performance Computing

Pyculib is a package that provides access to several numerical libraries that are optimized for performance on NVidia GPUs.

Pyculib was originally part of Accelerate, developed by Anaconda, Inc.

The current version, 1.0.1, was released on July 27, 2017.

Note: Pyculib is currently available for archival purposes and is not receiving updates. We think that CuPy provides a much more complete interface to standard GPU algorithms, and CuPy arrays work with Numba-compiled GPU kernels now.

CHAPTER 1

Features

- **Bindings to the following *CUDA libraries*:**
  - *cuBLAS*
  - *cuFFT*
  - *cuSPARSE*
  - *cuRAND*
  - *CUDA Sorting* algorithms from the CUB and Modern GPU libraries

CHAPTER 2

Installation

This section contains information related to:

2.1 Requirements

- 64-bit operating system–Windows, macOS or Linux
- **Supported Python and Numpy combinations:**
  - Python 2.7, 3.4-3.6 with Numpy 1.10-1.13
- Numba 0.33 or later
- Scipy 0.16 or later
- cffi

2.1.1 CUDA feature requirements

- CUDA toolkit 7.5 and supported NVIDIA driver
- At least one CUDA GPU with compute capability 2.0 or above

2.2 Installing Pyculib

If you already have the Anaconda free Python distribution, take the following steps to install Pyculib:

1. Run the command `conda update conda`.
2. Run the command `conda install pyculib`.

If you do not have Anaconda installed, see Downloads.

NOTE: Pyculib can also be installed into your own non-Anaconda Python environment via pip or setuptools.

2.3 Updating Pyculib

To update Pyculib, take the following steps:

1. Run the command `conda update conda`.
2. Run the command `conda update pyculib`.

CHAPTER 3

User guide

This section contains information related to:

3.1 CUDA libraries

The following CUDA libraries have bindings and algorithms that are available for use with Pyculib:

3.1.1 cuBLAS

Provides basic linear algebra building blocks. See NVIDIA cuBLAS.

The cuBLAS binding provides an interface that accepts NumPy arrays and Numba's CUDA device arrays. The binding automatically transfers NumPy array arguments to the device as required. This automatic transfer may generate some unnecessary transfers, so optimal performance is likely to be obtained by the manual transfer for NumPy arrays into device arrays and using the cuBLAS to manipulate device arrays where possible.

No special naming convention is used to identify the data type, unlike in the BLAS C and Fortran APIs. Arguments for array storage information which are part of the cuBLAS C API are also not necessary since NumPy arrays and device arrays contain this information.

All functions are accessed through the *pyculib.blas.Blas* class:

class pyculib.blas.**Blas**(*stream=0*)
All BLAS subprograms are available under the Blas object.

Parameters **stream** – Optional. A CUDA Stream.

BLAS Level 1

pyculib.blas.Blas.**nrm2**(*x*)
Computes the L2 norm for array *x*. Same as *numpy.linalg.norm(x)*.

Parameters **x** (*python.array*) – input vector

Returns resulting norm.

pyculib.blas.Blas.**dot**(*x*, *y*)
Compute the dot product of array *x* and array *y*. Same as *np.dot(x, y)*.

Parameters

- **x** (*python.array*) – vector
- **y** (*python.array*) – vector

Returns dot product of *x* and *y*

pyculib.blas.Blas.**dotc**(*x*, *y*)
Uses the conjugate of the element of the vectors to compute the dot product of array *x* and array *y* for complex dtype only. Same as *np.vdot(x, y)*.

Parameters

- **x** (*python.array*) – vector
- **y** (*python.array*) – vector

Returns dot product of *x* and *y*

pyculib.blas.Blas.**scal**(*alpha*, *x*)
Scale *x* inplace by alpha. Same as *x = alpha * x*

Parameters

- **alpha** – scalar
- **x** (*python.array*) – vector

pyculib.blas.Blas.**axpy**(*alpha*, *x*)
Compute *y = alpha * x + y* inplace.

Parameters

- **alpha** – scalar
- **x** (*python.array*) – vector

pyculib.blas.Blas.**amax**(*x*)
Find the index of the first largest element in array *x*. Same as *np.argmax(x)*

Parameters **x** (*python.array*) – vector

Returns index (start from 0).

pyculib.blas.Blas.**amin**(*x*)
Find the index of the first largest element in array *x*. Same as *np.argmin(x)*

Parameters **x** (*python.array*) – vector

Returns index (start from 0).

pyculib.blas.Blas.**asum**(*x*)
Compute the sum of all element in array *x*.

Parameters **x** (*python.array*) – vector

Returns *x.sum()*

pyculib.blas.Blas.**rot**(*x*, *y*, *c*, *s*)
Apply the Givens rotation matrix specified by the cosine element *c* and the sine element *s* inplace on vector element *x* and *y*.

Same as *x, y = c * x + s * y, -s * x + c * y*

Parameters

- **x** (*python.array*) – vector
- **y** (*python.array*) – vector

pyculib.blas.Blas.**rotg**(*a*, *b*)

Constructs the Givens rotation matrix with the column vector (a, b).

Parameters

- **a** – first element of the column vector
- **b** – second element of the column vector

Returns

a tuple (r, z, c, s)

r – *r = a**2 + b**2*

z – Use to reconstruct *c* and *s*. Refer to cuBLAS documentation for detail.

c – The consine element.

s – The sine element.

pyculib.blas.Blas.**rotm**(*x*, *y*, *param*)

Applies the modified Givens transformation inplace.

Same as:

```
param = flag, h11, h21, h12, h22
x[i] = h11 * x[i] + h12 * y[i]
y[i] = h21 * x[i] + h22 * y[i]
```

Refer to the cuBLAS documentation for the use of *flag*.

Parameters

- **x** (*python.array*) – vector
- **y** (*python.array*) – vector

pyculib.blas.Blas.**rotmg**(*d1*, *d2*, *x1*, *y1*)

Constructs the modified Givens transformation *H* that zeros out the second entry of a column vector *(d1 * x1, d2 * y1)*.

Parameters

- **d1** – scaling factor for the x-coordinate of the input vector
- **d2** – scaling factor for the y-coordinate of the input vector
- **x1** – x-coordinate of the input vector
- **y1** – y-coordinate of the input vector

Returns A 1D array that is usable in *rotm*. The first element is the flag for *rotm*. The rest of the elements corresponds to the *h11, h21, h12, h22* elements of *H*.

BLAS Level 2

All level 2 routines follow the following naming convention for all arguments:

- **A, B, C, AP – (2D array) Matrix argument.** *AP* implies packed storage for banded matrix.
- x, y, z – (1D arrays) Vector argument.
- alpha, beta – (scalar) Can be floats or complex numbers depending.
- m – (scalar) Number of rows of matrix *A*.
- **n – (scalar) Number of columns of matrix *A*. If *m* is not needed,** *n* also means the number of rows of the matrix *A*; thus, implying a square matrix.
- **trans, transa, transb – (string)** Select the operation *op* to apply to a matrix:
  - 'N': *op(X) = X*, the identity operation;
  - 'T': *op(X) = X**T*, the transpose;
  - 'C': *op(X) = X**H*, the conjugate transpose.

  trans only applies to the only matrix argument. *transa* and *transb* apply to matrix *A* and matrix *B*, respectively.
- **uplo – (string) Can be 'U' for filling the upper trianglar matrix; or 'L' for** filling the lower trianglar matrix.
- diag – (boolean) Whether the matrix diagonal has unit elements.
- **mode – (string) 'L' means the matrix is on the left side in the equation.** 'R' means the matrix is on the right side in the equation.

Note: The last array argument is always overwritten with the result.

`pyculib.blas.Blas.`**`gbmv`**(*trans*, *m*, *n*, *kl*, *ku*, *alpha*, *A*, *x*, *beta*, *y*)
banded matrix-vector multiplication *y = alpha * op(A) * x + beta * y* where *A* has *kl* sub-diagonals and *ku* super-diagonals.

`pyculib.blas.Blas.`**`gemv`**(*trans*, *m*, *n*, *alpha*, *A*, *x*, *beta*, *y*)
matrix-vector multiplication *y = alpha * op(A) * x + beta * y*

`pyculib.blas.Blas.`**`trmv`**(*uplo*, *trans*, *diag*, *n*, *A*, *x*)
triangular matrix-vector multiplication *x = op(A) * x*

`pyculib.blas.Blas.`**`tbmv`**(*uplo*, *trans*, *diag*, *n*, *k*, *A*, *x*)
triangular banded matrix-vector *x = op(A) * x*

`pyculib.blas.Blas.`**`tpmv`**(*uplo*, *trans*, *diag*, *n*, *AP*, *x*)
triangular packed matrix-vector multiplication *x = op(A) * x*

`pyculib.blas.Blas.`**`trsv`**(*uplo*, *trans*, *diag*, *n*, *A*, *x*)
Solves the triangular linear system with a single right-hand-side. *op(A) * x = b*

`pyculib.blas.Blas.`**`tpsv`**(*uplo*, *trans*, *diag*, *n*, *AP*, *x*)
Solves the packed triangular linear system with a single right-hand-side. *op(A) * x = b*

`pyculib.blas.Blas.`**`tbsv`**(*uplo*, *trans*, *diag*, *n*, *k*, *A*, *x*)
Solves the triangular banded linear system with a single right-hand-side. *op(A) * x = b*

`pyculib.blas.Blas.`**`symv`**(*uplo*, *n*, *alpha*, *A*, *x*, *beta*, *y*)
symmetric matrix-vector multiplication *y = alpha * A * x + beta * y*

pyculib.blas.Blas.**hemv**(*uplo*, *n*, *alpha*, *A*, *x*, *beta*, *y*)
Hermitian matrix-vector multiplication *y = alpha * A * x + beta * y*

pyculib.blas.Blas.**sbmv**(*uplo*, *n*, *k*, *alpha*, *A*, *x*, *beta*, *y*)
symmetric banded matrix-vector multiplication *y = alpha * A * x + beta * y*

pyculib.blas.Blas.**hbmv**(*uplo*, *n*, *k*, *alpha*, *A*, *x*, *beta*, *y*)
Hermitian banded matrix-vector multiplication *y = alpha * A * x + beta * y*

pyculib.blas.Blas.**spmv**(*uplo*, *n*, *alpha*, *AP*, *x*, *beta*, *y*)
symmetric packed matrix-vector multiplication *y = alpha * A * x + beta * y*

pyculib.blas.Blas.**hpmv**(*uplo*, *n*, *alpha*, *AP*, *x*, *beta*, *y*)
Hermitian packed matrix-vector multiplication *y = alpha * A * x + beta * y*

pyculib.blas.Blas.**ger**(*m*, *n*, *alpha*, *x*, *y*, *A*)
the rank-1 update *A := alpha * x * y ** T + A*

pyculib.blas.Blas.**geru**(*m*, *n*, *alpha*, *x*, *y*, *A*)
the rank-1 update *A := alpha * x * y ** T + A*

pyculib.blas.Blas.**gerc**(*m*, *n*, *alpha*, *x*, *y*, *A*)
the rank-1 update *A := alpha * x * y ** H + A*

pyculib.blas.Blas.**syr**(*uplo*, *n*, *alpha*, *x*, *A*)
symmetric rank 1 operation *A := alpha * x * x ** T + A*

pyculib.blas.Blas.**her**(*uplo*, *n*, *alpha*, *x*, *A*)
hermitian rank 1 operation *A := alpha * x * x ** H + A*

pyculib.blas.Blas.**spr**(*uplo*, *n*, *alpha*, *x*, *AP*)
the symmetric rank 1 operation *A := alpha * x * x ** T + A*

pyculib.blas.Blas.**hpr**(*uplo*, *n*, *alpha*, *x*, *AP*)
hermitian rank 1 operation *A := alpha * x * x ** H + A*

pyculib.blas.Blas.**syr2**(*uplo*, *n*, *alpha*, *x*, *y*, *A*)
symmetric rank-2 update *A = alpha * x * y ** T + y * x ** T + A*

pyculib.blas.Blas.**her2**(*uplo*, *n*, *alpha*, *x*, *y*, *A*)
Hermitian rank-2 update *A = alpha * x * y ** H + alpha * y * x ** H + A*

pyculib.blas.Blas.**spr2**(*uplo*, *n*, *alpha*, *x*, *y*, *A*)
packed symmetric rank-2 update *A = alpha * x * y ** T + y * x ** T + A*

pyculib.blas.Blas.**hpr2**(*uplo*, *n*, *alpha*, *x*, *y*, *A*)
packed Hermitian rank-2 update *A = alpha * x * y ** H + alpha * y * x ** H + A*

BLAS Level 3

All level 3 routines follow the same naming convention for arguments as in level 2 routines.

pyculib.blas.Blas.**gemm**(*transa*, *transb*, *m*, *n*, *k*, *alpha*, *A*, *B*, *beta*, *C*)
matrix-matrix multiplication *C = alpha * op(A) * op(B) + beta * C*

pyculib.blas.Blas.**syrk**(*uplo*, *trans*, *n*, *k*, *alpha*, *A*, *beta*, *C*)
symmetric rank- k update *C = alpha * op(A) * op(A) ** T + beta * C*

pyculib.blas.Blas.**herk**(*uplo*, *trans*, *n*, *k*, *alpha*, *A*, *beta*, *C*)
Hermitian rank- k update *C = alpha * op(A) * op(A) ** H + beta * C*

pyculib.blas.Blas.**symm**(*side*, *uplo*, *m*, *n*, *alpha*, *A*, *B*, *beta*, *C*)
symmetric matrix-matrix multiplication:

```
if  side == 'L':
    C = alpha * A * B + beta * C
else:  # side == 'R'
    C = alpha * B * A + beta * C
```

pyculib.blas.Blas.**hemm**(*side*, *uplo*, *m*, *n*, *alpha*, *A*, *B*, *beta*, *C*)
Hermitian matrix-matrix multiplication:

```
if  side == 'L':
    C = alpha * A * B + beta * C
else:   #  side == 'R':
    C = alpha * B * A + beta * C
```

pyculib.blas.Blas.**trsm**(*side*, *uplo*, *trans*, *diag*, *m*, *n*, *alpha*, *A*, *B*)
Solves the triangular linear system with multiple right-hand-sides:

```
if  side == 'L':
    op(A) * X = alpha * B
else:       # side == 'R'
    X * op(A) = alpha * B
```

pyculib.blas.Blas.**trmm**(*side*, *uplo*, *trans*, *diag*, *m*, *n*, *alpha*, *A*, *B*, *C*)
triangular matrix-matrix multiplication:

```
if  side == ':'
    C = alpha * op(A) * B
else:   # side == 'R'
    C = alpha * B * op(A)
```

pyculib.blas.Blas.**dgmm**(*side*, *m*, *n*, *A*, *x*, *C*)
matrix-matrix multiplication:

```
if  mode == 'R':
    C = A * x * diag(X)
else:       # mode == 'L'
    C = diag(X) * x * A
```

pyculib.blas.Blas.**geam**(*transa*, *transb*, *m*, *n*, *alpha*, *A*, *beta*, *B*, *C*)
matrix-matrix addition/transposition *C = alpha * op(A) + beta * op(B)*

3.1.2 cuSPARSE

Provides basic linear algebra operations for sparse matrices. See NVIDIA cuSPARSE for an in-depth description of the cuSPARSE library and its methods and data types. All functions are accessed through the *pyculib.sparse.Sparse* class:

class pyculib.sparse.**Sparse**(*idxbase=0*)
All cuSPARSE functions are available under the Sparse object.

Parameters **idxbase** – The base for indexing, either 0 or 1. Optional, defaults to 0.

Similarly to the cuBLAS interface, no special naming convention is used for functions to operate on different datatypes - all datatypes are handled by each function, and dispatch of the corresponding library function is handled by Pyculib. However, it is often necessary to provide a *matrix descriptor* to functions, which provides some information about the format and properties of a matrix. A matrix descriptor can be obtained from the *pyculib.sparse.Sparse.matdescr()* method:

pyculib.sparse.Sparse.**matdescr**(*indexbase*, *diagtype*, *fillmode*, *matrixtype*)
Creates a matrix descriptor that describes a matrix with the given *indexbase*, *diagtype*, *fillmode*, and *matrixtype*. Note that not all of these options are relevant to every matrix storage format.

Parameters

- **indexbase** – Optional. 0 for 0-based indexing, or 1 for 1-based indexing. If not specified, the default given to the *pyculib.sparse.Sparse* constructor is used instead.
- **diagtype** – Optional. Defaults to *'N'*. *'N'* signifies that the matrix diagonal has non-unit elements. *'U'* signifies that the matrix diagonal only contains unit elements.
- **fillmode** – Optional. Defaults to *'L'*. *'L'* indicates that the lower triangular part of the matrix is stored. *'U'* indicates that the upper triangular part of the matrix is stored.
- **matrixtype** – Optional. Defaults to *'G'*. *'S'* indicates that the matrix is symmetric. *'H'* indicates that the matrix is Hermitian. *'T'* indicates that the matrix is triangular. *'G'* is used for a *general* matrix, which is not symmetric, Hermitian, or triangular.

Returns A matrix descriptor.

Many of the methods of the *pyculib.sparse.Sparse* class accept the individual data structures that make up a sparse representation of a matrix (for example the values, the row pointers and the column indices for a CSR format matrix). However, some methods (such as *pyculib.sparse.Sparse.csrgemm_ez()*), accept an instance of the *pyculib.sparse.CudaSparseMatrix* class:

class pyculib.sparse.**CudaSparseMatrix**
Base class for a representation of a sparse matrix on a CUDA device. The constructor takes no arguments.

from_host_matrix(*matrix*, *stream*)
Initialise the matrix structure and values from an instance of a matrix on the host. The host matrix must be of the corresponding host type, which is documented for each subclass below.

copy_to_host(*stream*)
Create an instance of the corresponding host matrix type and copy the matrix structure and data into it from the device. See subclass documentation for an indication of the corresponding matrix type.

Subclasses of the sparse matrix type are:

class pyculib.sparse.**CudaBSRMatrix**
CUDA sparse matrix for which the corresponding type is a scipy.sparse.bsr_matrix.

class pyculib.sparse.**CudaCSRMatrix**
CUDA sparse matrix for which the corresponding type is a scipy.sparse.csr_matrix.

class pyculib.sparse.**CudaCSCMatrix**
CUDA sparse matrix for which the corresponding type is a scipy.sparse.csc_matrix.

There are also some convenience methods for constructing CUDA sparse matrices in a similar manner to Scipy sparse matrices:

sparse.**bsr_matrix**(***kws*)
Takes the same arguments as scipy.sparse.bsr_matrix.

Returns a BSR CUDA matrix.

sparse.**csr_matrix**(***kws*)
Takes the same arguments as scipy.sparse.csr_matrix.

Returns a CSR CUDA matrix.

sparse.**csc_matrix**(***kws*)
Takes the same arguments as scipy.sparse.csc_matrix.

Returns a CSC CUDA matrix.

BLAS Level 1

pyculib.sparse.Sparse.**axpyi**(*alpha*, *xVal*, *xInd*, *y*)
Multiplies the sparse vector *x* by *alpha* and adds the result to the dense vector *y*.

Parameters

- **alpha** – scalar
- **xVal** – vector of non-zero values of *x*
- **xInd** – vector of indices of non-zero values of *x*
- **y** – dense vector

Returns dense vector

pyculib.sparse.Sparse.**doti**(*xVal*, *xInd*, *y*)
Computes the dot product of the sparse vector *x* and dense vector *y*.

Parameters

- **xVal** – vector of non-zero values of *x*
- **xInd** – vector of indices of non-zero values of *x*
- **y** – dense vector

Returns scalar

pyculib.sparse.Sparse.**dotci**(*xVal*, *xInd*, *y*)
Computes the dot product of the complex conjugate of the sparse vector *x* and the dense vector *y*.

Parameters

- **xVal** – vector of non-zero values of *x*
- **xInd** – vector of indices of non-zero values of *x*
- **y** – dense vector

Returns scalar

pyculib.sparse.Sparse.**gthr**(*y*, *xVal*, *xInd*)
Gathers the elements of *y* at the indices *xInd* into the array *xVal*

Parameters

- **xVal** – vector of non-zero values of *x*
- **xInd** – vector of indices of non-zero values of *x*
- **y** – dense vector

Returns None

pyculib.sparse.Sparse.**gthrz**(*y*, *xVal*, *xInd*)
Gathers the elements of *y* at the indices *xInd* into the array *xVal* and zeroes out the gathered elements of *y*.

Parameters

- **xVal** – vector of non-zero values of *x*
- **xInd** – vector of indices of non-zero values of *x*
- **y** – dense vector

Returns None

pyculib.sparse.Sparse.**roti**(*xVal*, *xInd*, *y*, *c*, *s*)
Applies the Givens rotation matrix, *G*:

$$G = \begin{pmatrix} C & S \\ -S & C \end{pmatrix}$$

to the sparse vector *x* and dense vector *y*.

Parameters

- **xVal** – vector of non-zero values of *x*
- **xInd** – vector of indices of non-zero values of *x*
- **y** – dense vector
- **c** – cosine element of the rotation matrix
- **s** – sine element of the rotation matrix

Returns None

pyculib.sparse.Sparse.**sctr**(*xVal*, *xInd*, *y*)
Scatters the elements of the sparse vector *x* into the dense vector *y*. Elements of *y* whose indices are not listed in *xInd* are unmodified.

Parameters

- **xVal** – vector of non-zero values of *x*
- **xInd** – vector of indices of non-zero values of *x*
- **y** – dense vector

Returns None

BLAS Level 2

All level 2 routines follow the following naming convention for the following arguments:

- alpha, beta – (scalar) Can be real or complex numbers.
- descr, descrA, descrB – (descriptor) Matrix descriptor. An appropriate descriptor may be obtained by calling *pyculib.sparse.Sparse.matdescr()*. *descr* only applies to the only matrix argument. *descrA* and *descrB* apply to matrix *A* and matrix *B*, respectively.
- dir – (string) Can be *'C'* to indicate column-major block storage or *'R'* to indicate row-major block storage.
- **trans, transa, transb – (string)** Select the operation *op* to apply to a matrix:
  - *'N'*: *op(X) = X*, the identity operation;
  - *'T'*: *op(X) = X**T*, the transpose;
  - *'C'*: *op(X) = X**H*, the conjugate transpose.

  trans only applies to the only matrix argument. *transa* and *transb* apply to matrix *A* and matrix *B*, respectively.

pyculib.sparse.Sparse.**bsrmv_matrix**(*dir*, *trans*, *alpha*, *descr*, *bsrmat*, *x*, *beta*, *y*)
Matrix-vector multiplication *y = alpha * op(A) * x + beta * y* with a BSR-format matrix.

Parameters

- **dir** – block storage direction
- **trans** – operation to apply to the matrix
- **alpha** – scalar
- **descr** – matrix descriptor
- **bsrmat** – the matrix *A*
- **x** – dense vector
- **beta** – scalar
- **y** – dense vector

Returns None

pyculib.sparse.Sparse.**bsrmv**(*dir*, *trans*, *mb*, *nb*, *nnzb*, *alpha*, *descr*, *bsrVal*, *bsrRowPtr*, *bsrColInd*, *blockDim*, *x*, *beta*, *y*)

Matrix-vector multiplication *y = alpha * op(A) * x + beta * y* with a BSR-format matrix. This function accepts the individual arrays that make up the structure of a BSR matrix - if a *pyculib.sparse.CudaBSRMatrix* instance is to hand, it is recommended to use the *bsrmv_matrix()* method instead.

Parameters

- **dir** – block storage direction
- **trans** – operation to apply to the matrix
- **mb** – Number of block rows of the matrix
- **nb** – Number of block columns of the matrix
- **nnzb** – Number of nonzero blocks of the matrix
- **alpha** – scalar
- **descr** – matrix descriptor
- **bsrVal** – vector of nonzero values of the matrix
- **bsrRowPtr** – vector of block row pointers of the matrix
- **bsrColInd** – vector of block column indices of the matrix
- **blockDim** – block dimension of the matrix
- **x** – dense vector
- **beta** – scalar
- **y** – dense vector

Returns None

pyculib.sparse.Sparse.**bsrxmv**(*dir*, *trans*, *sizeOfMask*, *mb*, *nb*, *nnzb*, *alpha*, *descr*, *bsrVal*, *bsrMaskPtr*, *bsrRowPtr*, *bsrEndPtr*, *bsrColInd*, *blockDim*, *x*, *beta*, *y*)

Matrix-vector multiplication similar to *bsrmv()*, but including a mask operation: *y(mask) = (alpha * op(A) * x + beta * y)(mask)*. The blocks of y to be updated are specified in *bsrMaskPtr*. Blocks whose indices are not specified in *bsrMaskPtr* are left unmodified.

Parameters

- **dir** – block storage direction
- **trans** – operation to apply to the matrix
- **sizeOfMask** – number of updated blocks of rows of *y*

- **mb** – Number of block rows of the matrix
- **nb** – Number of block columns of the matrix
- **nnzb** – Number of nonzero blocks of the matrix
- **alpha** – scalar
- **descr** – matrix descriptor
- **bsrVal** – vector of nonzero values of the matrix
- **bsrMaskPtr** – vector of indices of the block elements to be updated
- **bsrRowPtr** – vector of block row pointers of the matrix
- **bsrEndPtr** – vector of pointers to the end of every block row plus one
- **bsrColInd** – vector of block column indices of the matrix
- **blockDim** – block dimension of the matrix
- **x** – dense vector
- **beta** – scalar
- **y** – dense vector

Returns None

pyculib.sparse.Sparse.**csrmv**(*trans*, *m*, *n*, *nnz*, *alpha*, *descr*, *csrVal*, *csrRowPtr*, *csrColInd*, *x*, *beta*, *y*)

Matrix-vector multiplication *y = alpha * op(A) * x + beta * y* with a CSR-format matrix.

Parameters

- **trans** – operation to apply to the matrix
- **m** – Number of rows of the matrix
- **n** – Number of columns of the matrix
- **nnz** – Number of nonzeroes of the matrix
- **alpha** – scalar
- **descr** – matrix descriptor
- **csrVal** – vector of nonzero values of the matrix
- **csrRowPtr** – vector of row pointers of the matrix
- **csrColInd** – vector of column indices of the matrix
- **x** – dense vector
- **beta** – scalar
- **y** – dense vector

Returns None

pyculib.sparse.Sparse.**csrsv_analysis**(*trans*, *m*, *nnz*, *descr*, *csrVal*, *csrRowPtr*, *csrColInd*)

Performs the analysis phase of the solution of the sparse triangular linear system *op(A) * y = alpha * x*. This needs to be executed only once for a given matrix and operation type.

Parameters

- **trans** – operation to apply to the matrix
- **m** – number of rows of the matrix

- **nnz** – number of nonzeroes of the matrix
- **descr** – matrix descriptor
- **csrVal** – vector of nonzero values of the matrix
- **csrRowPtr** – vector of row pointers of the matrix
- **csrColInd** – vector of column indices of the matrix

Returns the analysis result, which can be used as input to the solve phase

pyculib.sparse.Sparse.**csrsv_solve**(*trans*, *m*, *alpha*, *descr*, *csrVal*, *csrRowPtr*, *csrColInd*, *info*, *x*, *y*)

Performs the analysis phase of the solution of the sparse triangular linear system *op(A) * y = alpha * x*.

Parameters

- **trans** – operation to apply to the matrix
- **m** – number of rows of the matrix
- **alpha** – scalar
- **descr** – matrix descriptor
- **csrVal** – vector of nonzero values of the matrix
- **csrRowPtr** – vector of row pointers of the matrix
- **csrColInd** – vector of column indices of the matrix
- **info** – the analysis result from *csrsv_analysis()*
- **x** – dense vector
- **y** – dense vector into which the solve result is stored

Returns None

BLAS Level 3

pyculib.sparse.Sparse.**csrmm**(*transA*, *m*, *n*, *k*, *nnz*, *alpha*, *descrA*, *csrValA*, *csrRowPtrA*, *csrColIndA*, *B*, *ldb*, *beta*, *C*, *ldc*)

Matrix-matrix multiplication *C = alpha * op(A) * B + beta * C* where *A* is a sparse matrix in CSR format and *B* and *C* are dense matrices.

Parameters

- **transA** – operation to apply to *A*
- **m** – number of rows of *A*
- **n** – number of columns of *B* and *C*
- **k** – number of columns of *A*
- **nnz** – number of nonzeroes in *A*
- **alpha** – scalar
- **descrA** – matrix descriptor
- **csrValA** – vector of nonzero values of *A*
- **csrRowPtrA** – vector of row pointers of *A*
- **csrColIndA** – vector of column indices of *A*

- **B** – dense matrix
- **ldb** – leading dimension of *B*
- **beta** – scalar
- **C** – dense matrix
- **ldc** – leading dimension of *C*

Returns None

pyculib.sparse.Sparse.**csrmm2**(*transA*, *transB*, *m*, *n*, *k*, *nnz*, *alpha*, *descrA*, *csrValA*, *csrRowPtrA*, *csrColIndA*, *B*, *ldb*, *beta*, *C*, *ldc*)

Matrix-matrix multiplication *C = alpha * op(A) * op(B) + beta * C* where *A* is a sparse matrix in CSR format and *B* and *C* are dense matrices.

Parameters

- **transA** – operation to apply to *A*
- **transB** – operation to apply to *B*
- **m** – number of rows of *A*
- **n** – number of columns of *B* and *C*
- **k** – number of columns of *A*
- **nnz** – number of nonzeroes in *A*
- **alpha** – scalar
- **descrA** – matrix descriptor
- **csrValA** – vector of nonzero values of *A*
- **csrRowPtrA** – vector of row pointers of *A*
- **csrColIndA** – vector of column indices of *A*
- **B** – dense matrix
- **ldb** – leading dimension of *B*
- **beta** – scalar
- **C** – dense matrix
- **ldc** – leading dimension of *C*

Returns None

pyculib.sparse.Sparse.**csrsm_analysis**(*transA*, *m*, *nnz*, *descrA*, *csrValA*, *csrRowPtrA*, *csrColIndA*)

Performs the analysis phase of the solution of a sparse triangular linear system *op(A) * Y = alpha * X* with multiple right-hand sides where *A* is a sparse matrix in CSR format, and *X* and *Y* are dense matrices.

Parameters

- **transA** – operation to apply to *A*
- **m** – number of rows of *A*
- **nnz** – number of nonzeroes in *A*
- **descrA** – matrix descriptor
- **csrValA** – vector of nonzero values of *A*

- **csrRowPtrA** – vector of row pointers of *A*
- **csrColIndA** – vector of column indices of *A*

Returns the analysis result

pyculib.sparse.Sparse.**csrsm_solve**(*transA*, *m*, *n*, *alpha*, *descrA*, *csrValA*, *csrRowPtrA*, *csrColIndA*, *info*, *X*, *ldx*, *Y*, *ldy*)

Performs the analysis phase of the solution of a sparse triangular linear system *op(A) * Y = alpha * X* with multiple right-hand sides where *A* is a sparse matrix in CSR format, and *X* and *Y* are dense matrices.

Parameters

- **transA** – operation to apply to *A*
- **m** – number of rows of *A*
- **n** – number of columns of *B* and *C*
- **alpha** – scalar
- **descrA** – matrix descriptor
- **csrValA** – vector of nonzero values of *A*
- **csrRowPtrA** – vector of row pointers of *A*
- **csrColIndA** – vector of column indices of *A*
- **info** – the analysis result from *csrsm_analysis()*
- **X** – dense matrix
- **ldx** – leading dimension of *X*
- **Y** – dense matrix
- **ldy** – leading dimension of *Y*

Returns None

Extra Functions

pyculib.sparse.Sparse.**XcsrgeamNnz**(*m*, *n*, *descrA*, *nnzA*, *csrRowPtrA*, *csrColIndA*, *descrB*, *nnzB*, *csrRowPtrB*, *csrColIndB*, *descrC*, *csrRowPtrC*)

Set up the sparsity pattern for the matrix operation *C = alpha * A + beta * B* where *A*, *B*, and *C* are all sparse matrices in CSR format.

Parameters

- **m** – number of rows of all matrices
- **n** – number of columns of all matrices
- **descrA** – matrix descriptor for *A*
- **nnzA** – number of nonzeroes in *A*
- **csrRowPtrA** – vector of row pointers of *A*
- **csrColIndA** – vector of column indices of *A*
- **descrB** – matrix descriptor for *B*
- **nnzB** – number of nonzeroes in *B*
- **csrRowPtrB** – vector of row pointers of *B*

- **csrColIndB** – vector of column indices of *B*
- **descrC** – matrix descriptor for *B*
- **csrRowPtrC** – vector of row pointers of *C*, written to by this method

Returns number of nonzeroes in *C*

pyculib.sparse.Sparse.**csrgeam**(*m*, *n*, *alpha*, *descrA*, *nnzA*, *csrValA*, *csrRowPtrA*, *csrColIndA*, *beta*, *descrB*, *nnzB*, *csrValB*, *csrRowPtrB*, *csrColIndB*, *descrC*, *csrValC*, *csrRowPtrC*, *csrColIndC*)

Performs the the matrix operation *C = alpha * A + beta * B* where *A*, *B*, and *C* are all sparse matrices in CSR format.

Parameters

- **m** – number of rows of all matrices
- **n** – number of columns of all matrices
- **alpha** – scalar
- **descrA** – matrix descriptor for *A*
- **nnzA** – number of nonzeroes in *A*
- **csrValA** – vector of nonzero values of *A*
- **csrRowPtrA** – vector of row pointers of *A*
- **csrColIndA** – vector of column indices of *A*
- **beta** – scalar
- **descrB** – matrix descriptor for *B*
- **nnzB** – number of nonzeroes in *B*
- **csrValB** – vector of nonzero values of *B*
- **csrRowPtrB** – vector of row pointers of *B*
- **csrColIndB** – vector of column indices of *B*
- **descrC** – matrix descriptor for *B*
- **csrValC** – vector of nonzero values of *C*
- **csrRowPtrC** – vector of row pointers of *C*
- **csrColIndC** – vector of column indices of *C*

Returns None

pyculib.sparse.Sparse.**XcsrgemmNnz**(*transA*, *transB*, *m*, *n*, *k*, *descrA*, *nnzA*, *csrRowPtrA*, *csrColIndA*, *descrB*, *nnzB*, *csrRowPtrB*, *csrColIndB*, *descrC*, *csrRowPtrC*)

Set up the sparsity pattern for the matrix operation *C = op(A) * op(B)* where *A*, *B*, and *C* are all sparse matrices in CSR format.

Parameters

- **transA** – operation to apply to *A*
- **transB** – operation to apply to *B*
- **m** – number of rows of *A* and *C*
- **n** – number of columns of *B* and *C*

- **k** – number of columns/rows of *A*/*B*
- **descrA** – matrix descriptor for *A*
- **nnzA** – number of nonzeroes in *A*
- **csrRowPtrA** – vector of row pointers of *A*
- **csrColIndA** – vector of column indices of *A*
- **descrB** – matrix descriptor for *B*
- **nnzB** – number of nonzeroes in *B*
- **csrRowPtrB** – vector of row pointers of *B*
- **csrColIndB** – vector of column indices of *B*
- **descrC** – matrix descriptor for *C*
- **csrRowPtrC** – vector of row pointers of *C*, written by this function

Returns number of nonzeroes in *C*

pyculib.sparse.Sparse.**csrgemm**(*transA*, *transB*, *m*, *n*, *k*, *descrA*, *nnzA*, *csrValA*, *csrRowPtrA*, *csrColIndA*, *descrB*, *nnzB*, *csrValB*, *csrRowPtrB*, *csrColIndB*, *descrC*, *csrValC*, *csrRowPtrC*, *csrColIndC*)

Perform the matrix operation *C = op(A) * op(B)* where *A*, *B*, and *C* are all sparse matrices in CSR format.

Parameters

- **transA** – operation to apply to *A*
- **transB** – operation to apply to *B*
- **m** – number of rows of *A* and *C*
- **n** – number of columns of *B* and *C*
- **k** – number of columns/rows of *A*/*B*
- **descrA** – matrix descriptor for *A*
- **nnzA** – number of nonzeroes in *A*
- **csrValA** – vector of nonzero values in *A*
- **csrRowPtrA** – vector of row pointers of *A*
- **csrColIndA** – vector of column indices of *A*
- **descrB** – matrix descriptor for *B*
- **nnzB** – number of nonzeroes in *B*
- **csrValB** – vector of nonzero values in *B*
- **csrRowPtrB** – vector of row pointers of *B*
- **csrColIndB** – vector of column indices of *B*
- **descrC** – matrix descriptor for *C*
- **csrValC** – vector of nonzero values in *C*
- **csrRowPtrC** – vector of row pointers of *C*
- **csrColIndC** – vector of column indices of *C*

Returns None

pyculib.sparse.Sparse.**csrgemm_ez**(*A*, *B*, *transA='N'*, *transB='N'*, *descrA=None*, *descrB=None*, *descrC=None*)

Performs the matrix operation *C = op(A) * op(B)* where *A*, *B* and *C* are all sparse matrices in CSR format. This function accepts and returns *pyculib.sparse.CudaCSRMatrix* matrices, and makes calls to *XcsrgemmNnz()* and *csrgemm()*.

Parameters

- **A** – *pyculib.sparse.CudaCSRMatrix*
- **B** – *pyculib.sparse.CudaCSRMatrix*
- **transA** – optional, operation to apply to *A*
- **transB** – optional, operation to apply to *B*
- **descrA** – optional, matrix descriptor for *A*
- **descrB** – optional, matrix descriptor for *B*
- **descrC** – optional, matrix descriptor for *C*

Returns *pyculib.sparse.CudaCSRMatrix*

Preconditioners

pyculib.sparse.Sparse.**csric0**(*trans*, *m*, *descr*, *csrValA*, *csrRowPtrA*, *csrColIndA*, *info*)

Computes incomplete Cholesky factorization of a sparse matrix in CSR format with 0 fill-in and no pivoting: *op(A) = R**T * R*. This method must follow a call to *csrsv_analysis()*. The matrix *A* is overwritten with the upper or lower triangular factors *R* or *R**T*.

Parameters

- **trans** – operation to apply to the matrix
- **m** – number of rows and columns of the matrix
- **descr** – matrix descriptor
- **csrValA** – vector of nonzero values in *A*
- **csrRowPtrA** – vector of row pointers of *A*
- **csrColIndA** – vector of column indices of *A*
- **info** – analysis result

Returns None

pyculib.sparse.Sparse.**csrilu0**(*trans*, *m*, *descr*, *csrValA*, *csrRowPtrA*, *csrColIndA*, *info*)

Computes incomplete-LU factorization of a sparse matrix in CSR format with 0 fill-in and no pivoting: *op(A) = L * U*. This method must follow a call to *csrsv_analysis()*. The matrix *A* is overwritten with the lower and upper triangular factors *L* and *U*.

Parameters

- **trans** – operation to apply to the matrix
- **m** – number of rows and columns of the matrix
- **descr** – matrix descriptor
- **csrValA** – vector of nonzero values in *A*
- **csrRowPtrA** – vector of row pointers of *A*

- **csrColIndA** – vector of column indices of *A*
- **info** – analysis result

Returns None

pyculib.sparse.Sparse.**gtsv**(*m*, *n*, *dl*, *d*, *du*, *B*, *ldb*)
Computes the solution of a tridiagonal linear system with multiple right-hand sides: *A * Y = alpha * X*.

Parameters

- **m** – the size of the linear system
- **n** – the number of right-hand sides in the system
- **dl** – dense vector storing the lower-diagonal elements
- **d** – dense vector storing the diagonal elements
- **du** – dense vector storing the upper-diagonal elements
- **B** – dense matrix holding the right-hand sides of the system
- **ldb** – the leading dimension of *B*

Returns None

pyculib.sparse.Sparse.**gtsv_nopivot**(*m*, *n*, *dl*, *d*, *du*, *B*, *ldb*)
Similar to *gtsv()*, but computes the solution without performing any pivoting.

Parameters

- **m** – the size of the linear system
- **n** – the number of right-hand sides in the system
- **dl** – dense vector storing the lower-diagonal elements
- **d** – dense vector storing the diagonal elements
- **du** – dense vector storing the upper-diagonal elements
- **B** – dense matrix holding the right-hand sides of the system
- **ldb** – the leading dimension of *B*

Returns None

pyculib.sparse.Sparse.**gtsvStridedBatch**(*m*, *dl*, *d*, *du*, *x*, *batchCount*, *batchStride*)
Computes the solution of *i* tridiagonal linear systems: *A(i) * y(i) = alpha * x(i)*.

Parameters

- **m** – the size of the linear systems
- **dl** – stacked dense vector storing the lower-diagonal elements of each system
- **d** – stacked dense vector storing the diagonal elements of each system
- **du** – stacked dense vector storing the upper-diagonal elements of each system
- **x** – dense matrix holding the right-hand sides of the systems
- **batchCount** – number of systems to solve
- **batchStride** – number of elements separating the vectors of each system

Returns None

Format Conversion

pyculib.sparse.Sparse.**bsr2csr**(*dirA*, *mb*, *nb*, *descrA*, *bsrValA*, *bsrRowPtrA*, *bsrColIndA*, *blockDim*, *descrC*, *csrValC*, *csrRowPtrC*, *csrColIndC*)

Convert the sparse matrix *A* in BSR format to CSR format, stored in *C*.

Parameters

- **dirA** – row ('R') or column ('C') orientation of block storage
- **mb** – number of block rows of *A*
- **nb** – number of block columns of *A*
- **descrA** – matrix descriptor for *A*
- **bsrValA** – vector of nonzero values of *A*
- **bsrRowPtrA** – vector of block row pointers of *A*
- **bsrColIndA** – vector of block column indices of *A*
- **blockDim** – block dimension of *A*
- **descrC** – matrix descriptor for *C*
- **csrValA** – vector of nonzero values in *C*
- **csrRowPtrA** – vector of row pointers of *C*
- **csrColIndA** – vector of column indices of *C*

Returns None

pyculib.sparse.Sparse.**Xcoo2csr**(*cooRowInd*, *nnz*, *m*, *csrRowPtr*)

Converts an array containing uncompressed row indices corresponding to the COO format into into an array of compressed row pointers corresponding to the CSR format.

Parameters

- **cooRowInd** – integer array of uncompressed row indices
- **nnz** – number of nonzeroes
- **m** – number of matrix rows
- **csrRowPtr** – vector of row pointers to be written to

Returns None

pyculib.sparse.Sparse.**csc2dense**(*m*, *n*, *descrA*, *cscValA*, *cscRowIndA*, *cscColPtrA*, *A*, *lda*)

Convert the sparse matrix *A* in CSC format into a dense matrix.

Parameters

- **m** – number of rows of *A*
- **n** – number of columns of *A*
- **descrA** – matrix descriptor for *A*
- **cscValA** – values in the CSC representation of *A*
- **cscRowIndA** – row indices in the CSC representation of *A*
- **cscColPtrA** – column pointers in the CSC representation of *A*
- **A** – dense matrix representation of *A* to be written by this function.
- **lda** – leading dimension of *A*

Returns None

pyculib.sparse.Sparse.**Xcsr2bsrNnz**(*dirA*, *m*, *n*, *descrA*, *csrRowPtrA*, *csrColIndA*, *blockDim*, *descrC*, *bsrRowPtrC*)

Performs the analysis necessary for converting a matrix in CSR format into BSR format.

Parameters

- **dirA** – row ('R') or column ('C') orientation of block storage
- **m** – number of rows of matrix
- **n** – number of columns of matrix
- **descrA** – matrix descriptor for input matrix *A*
- **csrRowPtrA** – row pointers of matrix
- **csrColIndA** – column indices of matrix
- **blockDim** – block dimension of output matrix *C*
- **descrC** – matrix descriptor for output matrix *C*

Returns number of nonzeroes of matrix

pyculib.sparse.Sparse.**csr2bsr**(*dirA*, *m*, *n*, *descrA*, *csrValA*, *csrRowPtrA*, *csrColIndA*, *blockDim*, *descrC*, *bsrValC*, *bsrRowPtrC*, *bsrColIndC*)

Performs conversion of a matrix from CSR format into BSR format.

Parameters

- **dirA** – row ('R') or column ('C') orientation of block storage
- **m** – number of rows of matrix
- **n** – number of columns of matrix
- **descrA** – matrix descriptor for input matrix *A*
- **csrValA** – nonzero values of matrix
- **csrRowPtrA** – row pointers of matrix
- **csrColIndA** – column indices of matrix
- **blockDim** – block dimension of output matrix *C*
- **descrC** – matrix descriptor for output matrix *C*
- **bsrValC** – nonzero values of output matrix *C*
- **bsrRowPtrC** – block row pointers of output matrix *C*
- **bsrColIndC** – block column indices of output matrix *C*

Returns number of nonzeroes of matrix

pyculib.sparse.Sparse.**Xcsr2coo**(*csrRowPtr*, *nnz*, *m*, *cooRowInd*)

Converts an array of compressed row pointers corresponding to the CSR format into an array of uncompressed row indices corresponding to the COO format.

Parameters

- **csrRowPtr** – vector of row pointers
- **nnz** – number of nonzeroes
- **m** – number of rows of matrix

- **cooRowInd** – vector of uncompressed row indices written by this function

Returns None

pyculib.sparse.Sparse.**csr2csc**(*m*, *n*, *nnz*, *csrVal*, *csrRowPtr*, *csrColInd*, *cscVal*, *cscRowInd*, *cscColPtr*, *copyValues*)

Converts a sparse matrix in CSR format into a sparse matrix in CSC format.

Parameters

- **m** – number of rows of matrix
- **n** – number of columns of matrix
- **nnz** – number of nonzeroes of the matrix
- **csrVal** – values in the CSR representation
- **csrRowPtr** – row indices in the CSR representation
- **csrColInd** – column pointers in the CSR representation
- **cscVal** – values in the CSC representation
- **cscRowInd** – row indices in the CSC representation
- **cscColPtr** – column pointers in the CSC representation
- **copyValues** – *'N'* or *'S'* for symbolic or numeric copy of values

Returns None

pyculib.sparse.Sparse.**csr2dense**(*m*, *n*, *descr*, *csrVal*, *csrRowPtr*, *csrColInd*, *A*, *lda*)

Convert a sparse matrix in CSR format into dense format.

Parameters

- **m** – number of rows of matrix
- **n** – number of columns of matrix
- **descr** – matrix descriptor
- **csrVal** – values in the CSR representation
- **csrRowPtr** – row indices in the CSR representation
- **csrColInd** – column pointers in the CSR representation
- **A** – the dense representation, written to by this function
- **lda** – leading dimension of the matrix

Returns None

pyculib.sparse.Sparse.**dense2csc**(*m*, *n*, *descrA*, *A*, *lda*, *nnzPerCol*, *cscVal*, *cscRowInd*, *cscColPtr*)

Convert a dense matrix into a sparse matrix in CSC format. The *nnzPerCol* parameter may be computed with a call to *nnz()*.

Parameters

- **m** – number of rows of matrix
- **n** – number of columns of matrix
- **descrA** – matrix descriptor
- **A** – the matrix in dense format
- **lda** – leading dimension of the matrix

- **nnzPerCol** – array containing the number of nonzero elements per column
- **cscVal** – values in the CSC representation
- **cscRowInd** – row indices in the CSC representation
- **cscColPtr** – column pointers in the CSC representation

Returns None

pyculib.sparse.Sparse.**dense2csr**(*m*, *n*, *descrA*, *A*, *lda*, *nnzPerRow*, *csrVal*, *csrRowPtr*, *csrColInd*)

Convert a dense matrix into a sparse matrix in CSR format. The *nnzPerRow* parameter may be computed with a call to *nnz()*.

Parameters

- **m** – number of rows of matrix
- **n** – number of columns of matrix
- **descrA** – matrix descriptor
- **A** – the matrix in dense format
- **lda** – leading dimension of the matrix
- **nnzPerRow** – array containing the number of nonzero elements per row
- **csrVal** – values in the CSR representation
- **csrRowPtr** – row indices in the CSR representation
- **csrColInd** – column pointers in the CSR representation

Returns None

pyculib.sparse.Sparse.**nnz**(*dirA*, *m*, *n*, *descrA*, *A*, *lda*, *nnzPerRowCol*)

Computes the number of nonzero elements per row or column of a dense matrix, and the total number of nonzero elements in the matrix.

Parameters

- **dirA** – *'R'* for the number of nonzeroes per row, or *'C'* for per column.
- **m** – number of rows of matrix
- **n** – number of columns of matrix
- **descrA** – matrix descriptor
- **A** – the matrix
- **lda** – leading dimension of the matrix
- **nnzPerRowCol** – array to contain the number of nonzeroes per row or column

Returns total number of nonzeroes in the matrix

3.1.3 cuFFT

Provides FFT and inverse FFT for 1D, 2D and 3D arrays. See NVIDIA cuFFT.

Note: cuFFT only supports FFT operations on numpy.float32, numpy float64, numpy.complex64, numpy.complex128 with C-contiguous datalayout.

Forward FFT

pyculib.fft.**fft**(*ary*, *out*[, *stream*])

pyculib.fft.**fft_inplace**(*ary*[, *stream*])

Parameters

- **ary** – The input array. The inplace version stores the result in here.
- **out** – The output array for non-inplace versions.
- **stream** – The CUDA stream in which all operations will take place.

Inverse FFT

pyculib.fft.**ifft**(*ary*, *out*[, *stream*])

pyculib.fft.**ifft_inplace**(*ary*[, *stream*])

Parameters

- **ary** – The input array. The inplace version stores the result in here.
- **out** – The output array for non-inplace versions.
- **stream** – The CUDA stream in which all operations will take place.

FFTPlan

class pyculib.fft.**FFTPlan**(*shape*, *itype*, *otype*, *batch=1*, *stream=0*, *mode=1*)

Parameters

- **shape** – Input array shape.
- **itype** – Input data type.
- **otype** – Output data type.
- **batch** – Maximum number of operation to perform.
- **stream** – A CUDA stream for all the operations to put on.
- **mode** – Operation mode; e.g. MODE_NATIVE, MODE_FFTW_PADDING, MODE_FFTW_ASYMMETRIC, MODE_FFTW_ALL, MODE_DEFAULT.

forward(*ary*, *out=None*)
Perform forward FFT

Parameters

- **ary** – Input array
- **out** – Optional output array

Returns The output array or a new numpy array is *out* is None.

Note: If *ary* is *out*, an inplace operation is performed.

---

inverse(*ary*, *out=None*)
Perform inverse FFT

Parameters

- **ary** – Input array
- **out** – Optional output array

Returns The output array or a new numpy array is *out* is None.

3.1.4 cuRAND

Provides *pseudo-random number generator* (PRNG) and *quasi-random generator* (QRNG). See NVIDIA cuRAND.

class PRNG

class pyculib.rand.**PRNG**(*rndtype=100*, *seed=None*, *offset=None*, *stream=None*)

cuRAND pseudo random number generator

Parameters

- **rndtype** – Algorithm type. All possible values are listed as class attributes of this class, e.g. TEST, DEFAULT, XORWOW, MRG32K3A, MTGP32.
- **seed** – Seed for the RNG.
- **offset** – Offset to the random number stream.
- **stream** – CUDA stream.

Example:

```
>>> from pyculib import rand
>>> from numpy import empty
>>> prng = rand.PRNG(rndtype=rand.PRNG.XORWOW)
>>> r = empty(10)
>>> prng.uniform(r)
>>> r
array([ ... ])
```

lognormal(*ary*, *mean*, *sigma*, *size=None*)

Generate floating point random number sampled from a log-normal distribution and fill into ary.

Parameters

- **ary** – Numpy array or cuda device array.
- **mean** – Center of the distribution.
- **sigma** – Standard deviation of the distribution.
- **size** – Number of samples. Default to array size.

normal(*ary*, *mean*, *sigma*, *size=None*)

Generate floating point random number sampled from a normal distribution and fill into ary.

Parameters

- **ary** – Numpy array or cuda device array.
- **mean** – Center of the distribution.
- **sigma** – Standard deviation of the distribution.

- **size** – Number of samples. Default to array size.

poisson(*ary*, *lmbd*, *size=None*)
Generate floating point random number sampled from a poisson distribution and fill into ary.

Parameters

- **ary** – Numpy array or cuda device array.
- **lmbda** – Lambda for the distribution.
- **size** – Number of samples. Default to array size.

seed
Mutatable attribute for the seed for the RNG

uniform(*ary*, *size=None*)

Generate floating point random number sampled from a uniform distribution and fill into ary.

Parameters

- **ary** – Numpy array or cuda device array.
- **size** – Number of samples. Default to array size.

class QRNG

class pyculib.rand.**QRNG**(*rndtype=200*, *ndim=None*, *offset=None*, *stream=None*)
cuRAND quasi random number generator

Parameters

- **rndtype** – Algorithm type. Also control output data type. All possible values are listed as class attributes of this class, e.g. TEST, DEFAULT, SOBOL32, SCRAMBLED_SOBOL32, SOBOL64, SCRAMABLED_SOBOL64.
- **ndim** – Number of dimension for the QRNG.
- **offset** – Offset to the random number stream.
- **stream** – CUDA stream.

generate(*ary*, *size=None*)
Generate quasi random number in ary.

Parameters

- **ary** – Numpy array or cuda device array.
- **size** – Number of samples; Default to array size. Must be multiple of ndim.

ndim
Mutatable attribute for number of dimension for the QRNG.

Top Level PRNG Functions

Simple interface to the PRNG methods.

Note: This methods automatically create a PRNG object.

pyculib.rand.**uniform**(*size*, *dtype=<class 'float'>*, *device=False*)
Generate floating point random number sampled from a uniform distribution

Parameters

- **size** – Number of samples.
- **dtype** – np.float32 or np.float64.
- **device** – Set to True to return a device array instead or numpy array.

Returns A numpy array or a device array.

```
>>> from pyculib import rand
>>> rand.uniform(size=10)
array([...])
```

See also:

pyculib.rand.PRNG.uniform()

pyculib.rand.**normal**(*mean*, *sigma*, *size*, *dtype=<class 'float'>*, *device=False*)
Generate floating point random number sampled from a normal distribution

Parameters

- **mean** – Center point of the distribution.
- **sigma** – Standard deviation of the distribution.
- **size** – — Number of samples.
- **dtype** – np.float32 or np.float64.
- **device** – Set to True to return a device array instead or ndarray.

Returns A numpy array or a device array.

```
>>> from pyculib import rand
>>> rand.normal(mean=0, sigma=1, size=10)
array([...])
```

See also:

pyculib.rand.PRNG.normal()

pyculib.rand.**lognormal**(*mean*, *sigma*, *size*, *dtype=<class 'float'>*, *device=False*)
Generate floating point random number sampled from a log-normal distribution.

Parameters

- **mean** – Center point of the distribution.
- **sigma** – Standard deviation of the distribution.
- **size** – Number of samples.
- **dtype** – np.float32 or np.float64.
- **device** – set to True to return a device array instead or ndarray.

Returns A numpy array or a device array.

```
>>> from pyculib import rand
>>> rand.lognormal(mean=0, sigma=1, size=10)
array([...])
```

See also:

pyculib.rand.PRNG.lognormal()

pyculib.rand.**poisson**(*lmbd*, *size*, *device=False*)

Generate int32 random number sampled from a poisson distribution.

Parameters

- **lmbda** – Lambda of the distribution.
- **size** – Number of samples
- **device** – Set to True to return a device array instead or ndarray.

Returns A numpy array or a device array.

```
>>> from pyculib import rand
>>> rand.poisson(lmbd=1, size=10)
array([...], dtype=uint32)
```

See also:

pyculib.rand.PRNG.poisson()

Top Level QRNG Functions

Simple interface to the QRNG methods.

Note: This methods automatically create a QRNG object.

pyculib.rand.**quasi**(*size*, *bits=32*, *nd=1*, *device=False*)

Generate quasi random number using SOBOL{bits} RNG type.

Parameters

- **size** – Number of samples.
- **bits** – Bit length of output element; e.g. 32 or 64.
- **nd** – Number of dimension .
- **device** – Set to True to return a device array instead or ndarray.

Returns A numpy array or a device array.

```
>>> from pyculib import rand
>>> rand.quasi(10)
array([...], dtype=uint32)
```

See also:

pyculib.rand.QRNG.generate()

3.1.5 CUDA Sorting

Pyculib provides routines for sorting arrays on CUDA GPUs.

Sorting Large Arrays

The *pyculib.sorting.RadixSort* class is recommended for sorting large (approx. more than 1 million items) arrays of numeric types.

class pyculib.sorting.**RadixSort**(*maxcount*, *dtype*, *descending=False*, *stream=0*)
Provides radix sort and radix select.

The algorithm implemented here is best for large arrays (`N > 1e6`) due to the latency introduced by its use of multiple kernel launches. It is recommended to use `segmented_sort` instead for batches of smaller arrays.

Parameters

- **maxcount** (*int*) – Maximum number of items to sort
- **dtype** (*numpy.dtype*) – The element type to sort
- **descending** (*bool*) – Sort in descending order?
- **stream** – The CUDA stream to run the kernels in

argselect(*k*, *keys*, *begin_bit=0*, *end_bit=None*)
Similar to `RadixSort.select` but returns the new sorted indices.

Parameters

- **keys** (*numpy.ndarray*) – Keys to sort inplace
- **begin_bit** (*int*) – The first bit to sort
- **end_bit** (*int*) – Optional. The last bit to sort

Returns The indices indicating the new ordering as an array on the CUDA device or on the host.

argsort(*keys*, *begin_bit=0*, *end_bit=None*)
Similar to `RadixSort.sort` but returns the new sorted indices.

Parameters

- **keys** (*numpy.ndarray*) – Keys to sort inplace
- **begin_bit** (*int*) – The first bit to sort
- **end_bit** (*int*) – Optional. The last bit to sort

Returns The indices indicating the new ordering as an array on the CUDA device or on the host.

close()
Explicitly release internal resources

Called automatically when the object is deleted.

init_arg(*size*)
Initialize an empty CUDA ndarray of uint32 with ascending integers starting from zero

Parameters **size** (*int*) – Number of elements for the output array

Returns An array with values `[0, 1, 2, ...m size - 1 ]`

select(*k*, *keys*, *vals=None*, *begin_bit=0*, *end_bit=None*)
Perform a inplace k-select on `keys`.

Memory transfer is performed automatically.

Parameters

- **keys** (*numpy.ndarray*) – Keys to sort inplace

- **vals** (*numpy.ndarray*) – Optional. Additional values to be reordered along the sort. It is modified in place. Only the `uint32` dtype is supported in this version.
- **begin_bit** (*int*) – The first bit to sort
- **end_bit** (*int*) – Optional. The last bit to sort

sort (*keys*, *vals=None*, *begin_bit=0*, *end_bit=None*)
Perform a inplace sort on `keys`. Memory transfer is performed automatically.

Parameters

- **keys** (*numpy.ndarray*) – Keys to sort inplace
- **vals** (*numpy.ndarray*) – Optional. Additional values to be reordered along the sort. It is modified in place. Only the `uint32` dtype is supported in this version.
- **begin_bit** (*int*) – The first bit to sort
- **end_bit** (*int*) – Optional. The last bit to sort

Sorting Many Small Arrays

Using *pyculib.sorting.RadixSort* on small (approx. less than 1 million items) arrays has significant overhead due to multiple kernel launches.

A better alternative is to use *pyculib.sorting.segmented_sort()*-which launches a single kernel for sorting a batch of many small arrays.

pyculib.sorting.**segmented_sort** (*keys*, *vals*, *segments*, *stream=0*)
Performs an inplace sort on small segments (N < 1e6).

Parameters

- **keys** (*numpy.ndarray*) – Keys to sort inplace.
- **vals** (*numpy.ndarray*) – Values to be reordered inplace along the sort. Only the `uint32` dtype is supported in this implementation.
- **segments** (*numpy.ndarray*) – Segment separation location. e.g. `array([3, 6, 8])` for segments of `keys[:3]`, `keys[3:6]`, `keys[6:8]`, `keys[8:]`.
- **stream** – Optional. A cuda stream in which the kernels are executed.

3.2 Environment variables

`PYCULIB_WARNINGS`

If set to anything but 0 (zero), Pyculib may issue performance warnings, such as when input arguments need to be copied to adjust their data layout, or types, to match particular backend requirements.

CHAPTER 4

Release notes

4.1 Release notes

4.1.1 Version 1.0.2

Packaging fixes, remove fixed version of CUDA toolkit. No functional change.

4.1.2 Version 1.0.1

Minor documentation and packaging fixes.

4.1.3 Version 1.0.0

NumbaPro and Accelerate have been deprecated, and code generation features have been moved into open-source Numba. The CUDA library functions have been moved into Pyculib. There will be no further updates to NumbaPro or Accelerate.

CUDA libraries

Pyculib CUDA library functionality is equivalent to that in Accelerate 2.+, with the following packages renamed:

Accelerate package	Pyculib package
`accelerate.cuda.blas`	`pyculib.blas`
`accelerate.cuda.fft`	`pyculib.fft`
`accelerate.cuda.rand`	`pyculib.rand`
`accelerate.cuda.sparse`	`pyculib.sparse`
`accelerate.cuda.sorting`	`pyculib.sorting`

Index