CUDA
CUBLAS Library

PG-00000-002_V05
January, 2007
# Table of Contents

## The CUBLAS Library

- CUBLAS Types .................................................. 2  
  - Type cublasStatus .............................................. 2
- CUBLAS Helper Functions.  ..................................... 3
  - Function cublasInit() ........................................... 3
  - Function cublasShutdown() ...................................... 3
  - Function cublasGetError() ..................................... 4
  - Function cublasAlloc() .......................................... 4
  - Function cublasFree() ........................................... 4
  - Function cublasSetVector() .................................... 5
  - Function cublasGetVector() .................................... 5
  - Function cublasSetMatrix() .................................... 6
  - Function cublasGetMatrix() .................................... 6

## BLAS1 Functions .................................................. 8

- Single Precision BLAS1 Functions ............................ 9
  - Function cublasIsamax() ....................................... 9
  - Function cublasSasum() ........................................ 10
  - Function cublasSaxpy() ....................................... 10
  - Function cublasScopy() ....................................... 11
  - Function cublasSdot() ......................................... 12
  - Function cublasSnrm2() ....................................... 13
  - Function cublasSrot() ......................................... 14
  - Function cublasSrotg() ....................................... 15
  - Function cublasSrotm() ....................................... 16
  - Function cublasSrotmg() ...................................... 17
  - Function cublasSscal() ....................................... 18
  - Function cublasSswap() ....................................... 19

- Single Precision Complex BLAS1 Functions .................. 20
  - Function cublasCaxpy() ....................................... 20
  - Function cublasCcopy() ...................................... 21
  - Function cublasCscal() ....................................... 22
  - Function cublasCsscal() ...................................... 23
  - Function cublasCswap() ....................................... 23
  - Function cublasCcasum() ...................................... 24
BLAS2 and BLAS3 Functions ................................................................. 26
Single Precision BLAS2 Functions .................................................... 27
  Function cublasSgbmv() ................................................................. 27
  Function cublasSgemv() ................................................................. 29
  Function cublasSger() ................................................................. 30
  Function cublasSsbmv() ................................................................. 31
  Function cublasSspmv() ................................................................. 33
  Function cublasSspr() ................................................................. 34
  Function cublasSspr2() ................................................................. 35
  Function cublasSsymv() ................................................................. 36
  Function cublasSsyr() ................................................................. 37
  Function cublasSsyr2() ................................................................. 38
  Function cublasStbmv() ................................................................. 40
  Function cublasStbsv() ................................................................. 41
  Function cublasStpmv() ................................................................. 43
  Function cublasStpsv() ................................................................. 44
  Function cublasStrmv() ................................................................. 45
  Function cublasStrsv() ................................................................. 47
Single Precision Complex BLAS2 Functions .................................... 48
Single Precision BLAS3 Functions .................................................... 49
  Function cublasSgemm() ................................................................. 49
  Function cublasSsymm() ................................................................. 51
  Function cublasSsyrk() ................................................................. 53
  Function cublasSsyr2k() ................................................................. 54
  Function cublasStrmm() ................................................................. 56
  Function cublasStrsm() ................................................................. 58
Single Precision Complex BLAS3 Functions .................................... 59
  Function cublasCgemm() ................................................................. 60
CUBLAS Fortran Bindings ................................................................. 63
CHAPTER 1

The CUBLAS Library

CUBLAS is an implementation of BLAS (Basic Linear Algebra Subprograms) on top of the NVIDIA® CUDA™ (compute unified device architecture) driver. It allows access to the computational resources of NVIDIA GPUs. The library is self-contained at the API level, that is, no direct interaction with the CUDA driver is necessary.

The basic model by which applications use the CUBLAS library is to create matrix and vector objects in GPU memory space, fill them with data, call a sequence of CUBLAS functions, and, finally, upload the results from GPU memory space back to the host. To accomplish this, CUBLAS provides helper functions for creating and destroying objects in GPU space, and for writing data to and retrieving data from these objects.

For maximum compatibility with existing Fortran environments, CUBLAS uses column-major storage, and 1-based indexing. Calls to CUBLAS functions look very similar to calls to the original Fortran BLAS functions. For example, the Fortran function call

\[ \text{SDOT(KRANK+1-J,W(I,J),MDW,W(J,J),MDW)} \]

would map to this CUBLAS C/C++ function call:

\[ /* \text{Column-major addressing} */ \]
Because the CUBLAS core functions (as opposed to the helper functions) do not return error status directly (for reasons of compatibility with existing BLAS libraries), CUBLAS provides a separate function to retrieve the last error that was recorded, to aid in debugging.

Currently, only a subset of the CUBLAS core functions is implemented.

The interface to the CUBLAS library is the header file cublas.h. Applications using CUBLAS need to link against the DSO cublas.so (Linux) or the DLL cublas.dll (Win32).

The remainder of this chapter discusses “CUBLAS Types” on page 2 and “CUBLAS Helper Functions” on page 3.

---

### CUBLAS Types

The only CUBLAS type is **cublasStatus**.

#### Type cublasStatus

The type **cublasStatus** is used for function status returns. CUBLAS helper functions return status directly, while the status of CUBLAS core functions can be retrieved via **cublasGetError()**. Currently, the following values are defined:

<table>
<thead>
<tr>
<th>cublasStatus Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUBLAS_STATUS_SUCCESS</td>
<td>operation completed successfully</td>
</tr>
<tr>
<td>CUBLAS_STATUS_NOT_INITIALIZED</td>
<td>CUBLAS library not initialized</td>
</tr>
<tr>
<td>CUBLAS_STATUS_ALLOC_FAILED</td>
<td>resource allocation failed</td>
</tr>
<tr>
<td>CUBLAS_STATUS_INVALID_VALUE</td>
<td>unsupported numerical value was passed to function</td>
</tr>
<tr>
<td>CUBLAS_STATUS_MAPPING_ERROR</td>
<td>access to GPU memory space failed</td>
</tr>
<tr>
<td>CUBLAS_STATUS_EXECUTION_FAILED</td>
<td>GPU program failed to execute</td>
</tr>
<tr>
<td>CUBLAS_STATUS_INTERNAL_ERROR</td>
<td>an internal CUBLAS operation failed</td>
</tr>
</tbody>
</table>
CUBLAS Helper Functions

The following are the CUBLAS helper functions:

- "Function cublasInit()" on page 3
- "Function cublasShutdown()" on page 3
- "Function cublasGetError()" on page 4
- "Function cublasAlloc()" on page 4
- "Function cublasFree()" on page 4
- "Function cublasSetVector()" on page 5
- "Function cublasGetVector()" on page 5
- "Function cublasSetMatrix()" on page 6
- "Function cublasGetMatrix()" on page 6

Function cublasInit()

cublasStatus

cublasInit (void)

initializes the CUBLAS library and must be called before any other
CUBLAS API function is invoked. It allocates hardware resources
necessary for accessing the GPU.

Return Values

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUBLAS_STATUS_ALLOC_FAILED</td>
<td>if resources could not be allocated</td>
</tr>
<tr>
<td>CUBLAS_STATUS_SUCCESS</td>
<td>if CUBLAS library initialized successfully</td>
</tr>
</tbody>
</table>

Function cublasShutdown()

cublasStatus

cublasShutdown (void)

releases CPU-side resources used by the CUBLAS library. The release
of GPU-side resources may be deferred until the application shuts
down.

Return Values

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUBLAS_STATUS_NOT_INITIALIZED</td>
<td>if CUBLAS library was not initialized</td>
</tr>
<tr>
<td>CUBLAS_STATUS_SUCCESS</td>
<td>CUBLAS library shut down successfully</td>
</tr>
</tbody>
</table>
Function `cublasGetError()`

```c
void *cublasGetError()
```

returns the last error that occurred on invocation of any of the CUBLAS core functions. While the CUBLAS helper functions return status directly, the CUBLAS core functions do not, improving compatibility with those existing environments that do not expect BLAS functions to return status. Reading the error status via `cublasGetError()` resets the internal error state to `CUBLAS_STATUS_SUCCESS`.

Function `cublasAlloc()`

```c
int *cublasAlloc (int n, int elemSize, void **devicePtr)
```

creates an object in GPU memory space capable of holding an array of \( n \) elements, where each element requires \( \text{elemSize} \) bytes of storage. If the function call is successful, a pointer to the object in GPU memory space is placed in `devicePtr`. Note that this is a device pointer that cannot be dereferenced in host code.

Return Values

- `CUBLAS_STATUS_NOT_INITIALIZED` if CUBLAS library was not initialized
- `CUBLAS_STATUS_INVALID_VALUE` if \( n \leq 0 \) or \( \text{elemSize} \leq 0 \)
- `CUBLAS_STATUS_ALLOC_FAILED` if the object could not be allocated due to lack of resources.
- `CUBLAS_STATUS_SUCCESS` if storage was successfully allocated

Function `cublasFree()`

```c
void *cublasFree (const void *devicePtr)
```

destroys the object in GPU memory space referenced by `devicePtr`.

Return Values

- `CUBLAS_STATUS_NOT_INITIALIZED` if CUBLAS library was not initialized
- `CUBLAS_STATUS_INTERNAL_ERROR` if the object could not be deallocated
- `CUBLAS_STATUS_SUCCESS` if object was deallocated successfully
CHAPTER 1 The CUBLAS Library

Function cublasSetVector()

```
cublasStatus
cublasSetVector (int n, int elemSize, const void *x,
                int incx, void *y, int incy)
```

copies n elements from a vector x in CPU memory space to a vector y in GPU memory space. Elements in both vectors are assumed to have a size of `elemSize` bytes. Storage spacing between consecutive elements is `incx` for the source vector x and `incy` for the destination vector y. In general, y points to an object, or part of an object, allocated via `cublasAlloc()`. Column-major format for two-dimensional matrices is assumed throughout CUBLAS. If the vector is part of a matrix, a vector increment equal to 1 accesses a (partial) column of the matrix. Similarly, using an increment equal to the leading dimension of the matrix accesses a (partial) row.

Return Values

<table>
<thead>
<tr>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUBLAS_STATUS_NOT_INITIALIZED</td>
<td>if CUBLAS library was not initialized</td>
</tr>
<tr>
<td>CUBLAS_STATUS_INVALID_VALUE</td>
<td>if <code>incx</code>, <code>incy</code>, or <code>elemSize</code> &lt;= 0</td>
</tr>
<tr>
<td>CUBLAS_STATUS_MAPPING_ERROR</td>
<td>if error accessing GPU memory</td>
</tr>
<tr>
<td>CUBLAS_STATUS_SUCCESS</td>
<td>if operation completed successfully</td>
</tr>
</tbody>
</table>

Function cublasGetVector()

```
cublasStatus
cublasGetVector (int n, int elemSize, const void *x,
                int incx, void *y, int incy)
```

copies n elements from a vector x in GPU memory space to a vector y in CPU memory space. Elements in both vectors are assumed to have a size of `elemSize` bytes. Storage spacing between consecutive elements is `incx` for the source vector x and `incy` for the destination vector y. In general, x points to an object, or part of an object, allocated via `cublasAlloc()`. Column-major format for two-dimensional matrices is assumed throughout CUBLAS. If the vector is part of a matrix, a vector increment equal to 1 accesses a (partial) column of the matrix.
Similarly, using an increment equal to the leading dimension of the matrix accesses a (partial) row.

Return Values

<table>
<thead>
<tr>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUBLAS_STATUS_NOT_INITIALIZED</td>
<td>if CUBLAS library was not initialized</td>
</tr>
<tr>
<td>CUBLAS_STATUS_INVALID_VALUE</td>
<td>if incx, incy, or elemSize &lt;= 0</td>
</tr>
<tr>
<td>CUBLAS_STATUS_MAPPING_ERROR</td>
<td>if error accessing GPU memory</td>
</tr>
<tr>
<td>CUBLAS_STATUS_SUCCESS</td>
<td>if operation completed successfully</td>
</tr>
</tbody>
</table>

Function `cublasSetMatrix()`

```c

cublasStatus

cublasSetMatrix (int rows, int cols, int elemSize,
    const void *A, int lda, void *B, int ldb)
```

copies a tile of $rows \times cols$ elements from a matrix A in CPU memory space to a matrix B in GPU memory space. Each element requires storage of elemSize bytes. Both matrices are assumed to be stored in column-major format, with the leading dimension (that is, the number of rows) of source matrix A provided in lda, and the leading dimension of destination matrix B provided in ldb. B is a device pointer that points to an object, or part of an object, that was allocated in GPU memory space via `cublasAlloc()`.

Return Values

<table>
<thead>
<tr>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUBLAS_STATUS_NOT_INITIALIZED</td>
<td>if CUBLAS library was not initialized</td>
</tr>
<tr>
<td>CUBLAS_STATUS_INVALID_VALUE</td>
<td>if rows or cols &lt; 0; or elemSize, lda, or ldb &lt;= 0</td>
</tr>
<tr>
<td>CUBLAS_STATUS_MAPPING_ERROR</td>
<td>if error accessing GPU memory</td>
</tr>
<tr>
<td>CUBLAS_STATUS_SUCCESS</td>
<td>if operation completed successfully</td>
</tr>
</tbody>
</table>

Function `cublasGetMatrix()`

```c

cublasStatus

cublasGetMatrix (int rows, int cols, int elemSize,
    const void *A, int lda, void *B, int ldb)
```

copies a tile of $rows \times cols$ elements from a matrix A in GPU memory space to a matrix B in CPU memory space. Each element requires
storage of `elemSize` bytes. Both matrices are assumed to be stored in column-major format, with the leading dimension (that is, the number of rows) of source matrix `A` provided in `lda`, and the leading dimension of destination matrix `B` provided in `ldb`. `A` is a device pointer that points to an object, or part of an object, that was allocated in GPU memory space via `cublasAlloc()`.

Return Values

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUBLAS_STATUS_NOT_INITIALIZED</td>
<td>if CUBLAS library was not initialized</td>
</tr>
<tr>
<td>CUBLAS_STATUS_INVALID_VALUE</td>
<td>if <code>rows</code> or <code>cols</code> &lt; 0; or <code>elemSize</code>, <code>lda</code>, or <code>ldb</code> &lt;= 0</td>
</tr>
<tr>
<td>CUBLAS_STATUS_MAPPING_ERROR</td>
<td>if error accessing GPU memory</td>
</tr>
<tr>
<td>CUBLAS_STATUS_SUCCESS</td>
<td>if operation completed successfully</td>
</tr>
</tbody>
</table>
BLAS1 Functions

Level 1 Basic Linear Algebra Subprograms (BLAS1) are functions that perform scalar, vector, and vector-vector operations. The CUBLAS BLAS1 implementation is described in these sections:

- “Single Precision BLAS1 Functions” on page 9
- “Single Precision Complex BLAS1 Functions” on page 20
Single Precision BLAS1 Functions

The single precision BLAS1 functions are as follows:

- “Function cublasIsamax()” on page 9
- “Function cublasSasum()” on page 10
- “Function cublasSaxpy()” on page 10
- “Function cublasScopy()” on page 11
- “Function cublasSdot()” on page 12
- “Function cublasSnrm2()” on page 13
- “Function cublasSscal()” on page 14
- “Function cublasSrotg()” on page 15
- “Function cublasSrotm()” on page 16
- “Function cublasSswap()” on page 19

Function cublasIsamax()

```c
int cublasIsamax (int n, const float *x, int incx)
```

finds the smallest index of the maximum magnitude element of single precision vector \( x \); that is, the result is the first \( i, i = 0 \) to \( n-1 \), that maximizes \( \text{abs}(x[1+i*incx]) \). The result reflects 1-based indexing for compatibility with Fortran.

**Input**

- \( n \) number of elements in input vector
- \( x \) single precision vector with \( n \) elements
- \( incx \) storage spacing between elements of \( x \)

**Output**

returns the smallest index (returns zero if \( n <= 0 \) or \( incx <= 0 \))

Reference: [http://www.netlib.org/blas/isamax.f](http://www.netlib.org/blas/isamax.f)
Error status for this function can be retrieved via `cublasGetError()`.

**Error Status**

<table>
<thead>
<tr>
<th>Error Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUBLAS_STATUS_NOT_INITIALIZED</td>
<td>If CUBLAS library was not initialized</td>
</tr>
<tr>
<td>CUBLAS_STATUS_EXECUTION_FAILED</td>
<td>If function failed to launch on GPU</td>
</tr>
</tbody>
</table>

**Function cublasSasum()**

```c
float
cublasSasum (int n, const float *x, int incx)
```

Computes the sum of the absolute values of the elements of single precision vector `x`; that is, the result is the sum from `i = 0` to `n-1` of `abs(x[i + i*incx])`.

**Input**

- `n`: number of elements in input vector
- `x`: single precision vector with `n` elements
- `incx`: storage spacing between elements of `x`

**Output**

- Returns the single precision sum of absolute values
  (returns zero if `n <= 0` or `incx <= 0`, or if an error occurs)

**Reference:** [http://www.netlib.org/blas/sasum.f](http://www.netlib.org/blas/sasum.f)

Error status for this function can be retrieved via `cublasGetError()`.

**Error Status**

<table>
<thead>
<tr>
<th>Error Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUBLAS_STATUS_NOT_INITIALIZED</td>
<td>If CUBLAS library was not initialized</td>
</tr>
<tr>
<td>CUBLAS_STATUS_EXECUTION_FAILED</td>
<td>If function failed to launch on GPU</td>
</tr>
</tbody>
</table>

**Function cublasSaxpy()**

```c
void
cublasSaxpy (int n, float alpha, const float *x,
             int incx, float *y, int incy)
```

Multiplies single precision vector `x` by single precision scalar `alpha` and adds the result to single precision vector `y`; that is, it overwrites single precision `y` with single precision `alpha * x + y`. 
For $i = 0$ to $n-1$, it replaces

$$y[\text{ly} + i \times \text{incy}] \text{ with } \alpha \times [\text{lx} + i \times \text{incx}] + y[\text{ly} + i \times \text{incy}],$$

where

$$\text{lx} = 1 \text{ if incx} \geq 0, \text{ else }$$

$$\text{lx} = 1 + (1 - n) \times \text{incx};$$

$\text{ly}$ is defined in a similar way using incy.

**Input**

- $n$: number of elements in input vectors
- $\alpha$: single precision scalar multiplier
- $x$: single precision vector with $n$ elements
- incx: storage spacing between elements of $x$
- $y$: single precision vector with $n$ elements
- incy: storage spacing between elements of $y$

**Output**

- $y$: single precision result (unchanged if $n \leq 0$)

Reference: [http://www.netlib.org/blas/saxpy.f](http://www.netlib.org/blas/saxpy.f)

Error status for this function can be retrieved via `cublasGetError()`.

**Error Status**

- `CUBLAS_STATUS_NOT_INITIALIZED` if CUBLAS library was not initialized
- `CUBLAS_STATUS_EXECUTION_FAILED` if function failed to launch on GPU

**Function cublasScopy()**

```c
void
cublasScopy (int n, const float *x, int incx, float *y, int incy)
```

copies the single precision vector $x$ to the single precision vector $y$. For $i = 0$ to $n-1$, it copies

$$x[\text{lx} + i \times \text{incx}] \text{ to } y[\text{ly} + i \times \text{incy}],$$
where

\[ l_x = 1 \text{ if } \text{incx} \geq 0, \text{ else } \]
\[ l_x = 1 + (1 - n) \times \text{incx}; \]

\( l_y \) is defined in a similar way using \( \text{incy} \).

**Input**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( n )</td>
<td>number of elements in input vectors</td>
</tr>
<tr>
<td>( x )</td>
<td>single precision vector with ( n ) elements</td>
</tr>
<tr>
<td>( \text{incx} )</td>
<td>storage spacing between elements of ( x )</td>
</tr>
<tr>
<td>( y )</td>
<td>single precision vector with ( n ) elements</td>
</tr>
<tr>
<td>( \text{incy} )</td>
<td>storage spacing between elements of ( y )</td>
</tr>
</tbody>
</table>

**Output**

\( y \) contains single precision vector \( x \)

**Reference:** [http://www.netlib.org/blas/scopy.f](http://www.netlib.org/blas/scopy.f)

Error status for this function can be retrieved via `cublasGetError()`.

**Error Status**

- `CUBLAS_STATUS_NOT_INITIALIZED` if CUBLAS library was not initialized
- `CUBLAS_STATUS_EXECUTION_FAILED` if function failed to launch on GPU

**Function cublasSdot()**

```c
float

cublasSdot (int n, const float *x, int incx, const float *y, int incy)
```

computes the dot product of two single precision vectors. It returns the dot product of the single precision vectors \( x \) and \( y \) if successful, and \( 0.0f \) otherwise. It computes the sum for \( i = 0 \) to \( n-1 \) of

\[ x[l_x + i \times \text{incx}] \times y[l_y + i \times \text{incy}], \]

where

\[ l_x = 1 \text{ if } \text{incx} \geq 0, \text{ else } \]
\[ l_x = 1 + (1 - n) \times \text{incx}; \]
ly is defined in a similar way using incy.

Input

- **n**: number of elements in input vectors
- **x**: single precision vector with *n* elements
- **incx**: storage spacing between elements of *x*
- **y**: single precision vector with *n* elements
- **incy**: storage spacing between elements of *y*

Output

returns single precision dot product (returns zero if *n* <= 0)

Reference: [http://www.netlib.org/blas/sdot.f](http://www.netlib.org/blas/sdot.f)

Error status for this function can be retrieved via `cublasGetError()`. Error Status

- `CUBLAS_STATUS_NOT_INITIALIZED`: if CUBLAS library was not initialized
- `CUBLAS_STATUS_EXECUTION_FAILED`: if function failed to execute on GPU

Function `cublasSnrm2()`

```c
float
float cublasSnrm2 (int n, const float *x, int incx)
```

computes the Euclidean norm of the single precision *n*-vector *x* (with storage increment `incx`). This code uses a multiphase model of accumulation to avoid intermediate underflow and overflow.

Input

- **n**: number of elements in input vector
- **x**: single precision vector with *n* elements
- **incx**: storage spacing between elements of *x*

Output

returns the Euclidian norm
(returns zero if *n* <= 0, `incx` <= 0, or if an error occurred)

Reference: [http://www.netlib.org/blas/snrm2.f](http://www.netlib.org/blas/snrm2.f)

Error status for this function can be retrieved via `cublasGetError()`.  

### Error Status

<table>
<thead>
<tr>
<th>Error Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUBLAS_STATUS_NOT_INITIALIZED</td>
<td>If CUBLAS library was not initialized</td>
</tr>
<tr>
<td>CUBLAS_STATUS_EXECUTION_FAILED</td>
<td>If function failed to launch on GPU</td>
</tr>
</tbody>
</table>

### Function `cublasSrot()`

```c
void
cublasSrot (int n, float *x, int incx, float *y, int incy, float sc, float ss)
```

Multiplies a $2 \times 2$ matrix

$$
\begin{bmatrix}
sc & ss \\
-ss & sc
\end{bmatrix}
$$

with the $2 \times n$ matrix

$$
\begin{bmatrix}
x^T \\
y^T
\end{bmatrix}
$$

The elements of $x$ are in $x[lx + i \times incx], i = 0 \text{ to } n-1$, where

$$
lx = 1 \text{ if } incx >= 0, \text{ else } lx = 1 + (1-n) \times incx;
$$

$y$ is treated similarly using $ly$ and $incy$.

### Input

- $n$: number of elements in input vectors
- $x$: single precision vector with $n$ elements
- $incx$: storage spacing between elements of $x$
- $y$: single precision vector with $n$ elements
- $incy$: storage spacing between elements of $y$
- $sc$: element of rotation matrix
- $ss$: element of rotation matrix

### Output

- $x$: rotated vector $x$ (unchanged if $n \leq 0$
- $y$: rotated vector $y$ (unchanged if $n \leq 0$

Reference: [http://www.netlib.org/blas/srot.f](http://www.netlib.org/blas/srot.f)
Error status for this function can be retrieved via `cublasGetError()`. 

**Error Status**

<table>
<thead>
<tr>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CUBLAS_STATUS_NOT_INITIALIZED</code></td>
<td>if CUBLAS library was not initialized</td>
</tr>
<tr>
<td><code>CUBLAS_STATUS_EXECUTION_FAILED</code></td>
<td>if function failed to launch on GPU</td>
</tr>
</tbody>
</table>

**Function cublasSrotg()**

```c
void

        cublasSrotg (float *sa, float *sb, float *sc, float *ss)
```

constructs the Givens transformation

\[
G = \begin{bmatrix}
sc & ss \\
-sc & ss
\end{bmatrix}, \quad sc^2 + ss^2 = 1
\]

which zeros the second entry of the 2-vector \([sa \ sb]^T\).

The quantity \(r = \pm \sqrt{sa^2 + sb^2}\) overwrites \(sa\) in storage. The value of \(sb\) is overwritten by a value \(z\) which allows \(sc\) and \(ss\) to be recovered by the following algorithm:

- if \(z = 1\) set \(sc = 0.0\) and \(ss = 1.0\).
- if \(\text{abs}(z) < 1\) set \(sc = \sqrt{1 - z^2}\) and \(ss = z\).
- if \(\text{abs}(z) > 1\) set \(sc = 1/z\) and \(ss = \sqrt{1 - sc^2}\).

The function `cublasSrot(n, x, incx, y, incy, sc, ss)` normally is called next to apply the transformation to a \(2\times n\) matrix.

**Input**

- \(sa\) single precision scalar
- \(sb\) single precision scalar

**Output**

- \(sa\) single precision \(r\)
- \(sb\) single precision \(z\)
- \(sc\) single precision result
- \(ss\) single precision result
Reference: http://www.netlib.org/blas/srotg.f
This function does not set any error status.

Function cublasSrotm()

```c
void cublasSrotm (int n, float *x, int incx, float *y, int incy, const float* sparam)
```

applies the modified Givens transformation, \( h \), to the \( 2 \times n \) matrix
\[
\begin{bmatrix}
  x^T \\
  y^T
\end{bmatrix}
\]

The elements of \( x \) are in \( x[lx+i*incy] \), \( i = 0 \) to \( n-1 \), where
- \( lx = 1 \) if \( incx >= 0 \), else
- \( lx = 1+(1-n)*incy \);

\( y \) is treated similarly using \( ly \) and \( incy \).

With \( sparam[0] = sflag \), \( h \) has one of the following forms:

- \( sflag = -1.0f \):
  \[
  h = \begin{bmatrix}
    sh00 & sh01 \\
    sh10 & sh11
  \end{bmatrix}
  \]

- \( sflag = 0.0f \):
  \[
  h = \begin{bmatrix}
    1.0f & 0.0f \\
    0.0f & 1.0f
  \end{bmatrix}
  \]

- \( sflag = 1.0f \):
  \[
  h = \begin{bmatrix}
    sh00 & 1.0f \\
    -1.0f & sh11
  \end{bmatrix}
  \]

- \( sflag = -2.0f \):
  \[
  h = \begin{bmatrix}
    1.0f & 0.0f \\
    0.0f & 1.0f
  \end{bmatrix}
  \]

**Input**

- \( n \) number of elements in input vectors.
- \( x \) single precision vector with \( n \) elements.
- \( incx \) storage spacing between elements of \( x \).
- \( y \) single precision vector with \( n \) elements.
- \( incy \) storage spacing between elements of \( y \).
- \( sparam \) 5-element vector. \( sparam[0] \) is \( sflag \) described above. \( sparam[1] \) through \( sparam[4] \) contain the \( 2 \times 2 \) rotation matrix \( h \): \( sparam[1] \) contains \( sh00 \), \( sparam[2] \) contains \( sh10 \), \( sparam[3] \) contains \( sh01 \), and \( sparam[4] \) contains \( sh11 \).
CHAPTER 2

BLAS1 Functions

Reference: http://www.netlib.org/blas/srotm.f

Error status for this function can be retrieved via cublasGetError().

Error Status

<table>
<thead>
<tr>
<th>Error Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUBLAS_STATUS_NOT_INITIALIZED</td>
<td>if CUBLAS library was not initialized</td>
</tr>
<tr>
<td>CUBLAS_STATUS_EXECUTION_FAILED</td>
<td>if function failed to launch on GPU</td>
</tr>
</tbody>
</table>

Function cublasSrotmg()

void

cublasSrotmg (float *sd1, float *sd2, float *sx1,
              const float *sy1, float *sparam)

constructs the modified Givens transformation matrix \( h \) which zeros the second component of the 2-vector \( (sd1*sx1, sd2*sy1)^T \).

With \( sparam[0] = sflag \), \( h \) has one of the following forms:

- \( sflag = -1.0f \)
  \[
  h = \begin{bmatrix}
  sh00 & sh01 \\
  sh10 & sh11
  \end{bmatrix}
  \]

- \( sflag = 0.0f \)
  \[
  h = \begin{bmatrix}
  1.0f & sh01 \\
  sh10 & 1.0f
  \end{bmatrix}
  \]

- \( sflag = 1.0f \)
  \[
  h = \begin{bmatrix}
  sh00 & 1.0f \\
  -1.0f & sh11
  \end{bmatrix}
  \]

- \( sflag = -2.0f \)
  \[
  h = \begin{bmatrix}
  1.0f & 0.0f \\
  0.0f & 1.0f
  \end{bmatrix}
  \]

\( sparam[1] \) through \( sparam[4] \) contain \( sh00, sh10, sh01, \) and \( sh11 \), respectively. Values of \( 1.0f, -1.0f, \) or \( 0.0f \) implied by the value of \( sflag \) are not stored in \( sparam \).

Input

- \( sd1 \) single precision scalar.
- \( sd2 \) single precision scalar.
- \( sx1 \) single precision scalar.
- \( sy1 \) single precision scalar.
This function does not set any error status.

Function cublasSscal()

```c
void
cublasSscal (int n, float alpha, float *x, int incx)
```
replaces single precision vector \( x \) with single precision \( \alpha \times x \). For \( i = 0 \) to \( n-1 \), it replaces

\[
x[lx+i*incx] \text{ with } \alpha \times x[lx+i*incx],
\]

where

\[
lx = 1 \text{ if incx} \geq 0, \text{ else } \lx = 1 + (1-n) \times \text{incx}.
\]

Input

- \( n \): number of elements in input vector
- \( \alpha \): single precision scalar multiplier
- \( x \): single precision vector with \( n \) elements
- \( \text{incx} \): storage spacing between elements of \( x \)

Output

- \( x \): single precision result (unchanged if \( n \leq 0 \) or \( \text{incx} \leq 0 \))

Reference: [http://www.netlib.org/blas/sscal.f](http://www.netlib.org/blas/sscal.f)
Error status for this function can be retrieved via \texttt{cublasGetError()}.  

\textbf{Error Status}  
\begin{center}
\begin{tabular}{ll}
\texttt{CUBLAS_STATUS_NOT_INITIALIZED} & if CUBLAS library was not initialized \\
\texttt{CUBLAS_STATUS_EXECUTION_FAILED} & if function failed to launch on GPU \\
\end{tabular}
\end{center}

\textbf{Function \texttt{cublasSswap()}}  
\begin{verbatim}
void
\texttt{cublasSswap (int n, float *x, int incx, float *y, int incy)}
\end{verbatim}

interchanges single precision vector \(x\) with single precision vector \(y\). For \(i = 0\) to \(n-1\), it interchanges  
\[x[lx + i * \text{incx}] \text{ with } y[ly + i * \text{incy}],\]
where  
\[lx = 1 \text{ if incx} \geq 0, \text{ else} \]
\[lx = 1 + (1-n) * \text{incx};\]

\(ly\) is defined in a similar manner using \texttt{incy}.  

\textbf{Input}  
\begin{itemize}
\item \texttt{n} \quad \text{number of elements in input vectors}
\item \texttt{x} \quad \text{single precision vector with } n \text{ elements}
\item \texttt{incx} \quad \text{storage spacing between elements of } \texttt{x}
\item \texttt{y} \quad \text{single precision vector with } n \text{ elements}
\item \texttt{incy} \quad \text{storage spacing between elements of } \texttt{y}
\end{itemize}

\textbf{Output}  
\begin{itemize}
\item \texttt{x} \quad \text{input vector } y \text{ (unchanged if } n \leq 0)\]
\item \texttt{y} \quad \text{input vector } x \text{ (unchanged if } n \leq 0)\]
\end{itemize}

\textbf{Reference:} \url{http://www.netlib.org/blas/sswap.f}

Error status for this function can be retrieved via \texttt{cublasGetError()}.  

\textbf{Error Status}  
\begin{center}
\begin{tabular}{ll}
\texttt{CUBLAS_STATUS_NOT_INITIALIZED} & if CUBLAS library was not initialized \\
\texttt{CUBLAS_STATUS_EXECUTION_FAILED} & if function failed to launch on GPU \\
\end{tabular}
\end{center}
Single Precision Complex BLAS1 Functions

The single precision complex BLAS1 functions are as follows:

- “Function cublasCaxpy()” on page 20
- “Function cublasCcopy()” on page 21
- “Function cublasCscal()” on page 22
- “Function cublasCsscal()” on page 23
- “Function cublasCswap()” on page 23
- “Function cublasScasum()” on page 24

Function cublasCaxpy()

void
cublasCaxpy (int n, cuComplex alpha, const cuComplex *x, int incx, cuComplex *y, int incy)

multiplies single precision complex vector x by single precision complex scalar alpha and adds the result to single precision complex vector y; that is, it overwrites single precision complex y with single precision complex alpha * x + y.

For i = 0 to n-1, it replaces

\[ y[ly+i*incy] \text{ with } alpha * x[lx+i*incx] + y[ly+i*incy], \]

where

\[ lx = 0 \text{ if incx} \geq 0, \text{ else } \]
\[ lx = 1 + (1-n)*incx; \]

\[ ly \] is defined in a similar way using incy.

Input

<table>
<thead>
<tr>
<th>n</th>
<th>number of elements in input vectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>alpha</td>
<td>single precision complex scalar multiplier</td>
</tr>
<tr>
<td>x</td>
<td>single precision complex vector with n elements</td>
</tr>
<tr>
<td>incx</td>
<td>storage spacing between elements of x</td>
</tr>
<tr>
<td>y</td>
<td>single precision complex vector with n elements</td>
</tr>
<tr>
<td>incy</td>
<td>storage spacing between elements of y</td>
</tr>
</tbody>
</table>
CHAPTER 2

BLAS1 Functions

Reference: http://www.netlib.org/blas/caxpy.f

Error status for this function can be retrieved via `cublasGetError()`.

<table>
<thead>
<tr>
<th>Error Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUBLAS_STATUS_NOT_INITIALIZED</td>
<td>if CUBLAS library was not initialized</td>
</tr>
<tr>
<td>CUBLAS_STATUS_EXECUTION_FAILED</td>
<td>if function failed to launch on GPU</td>
</tr>
</tbody>
</table>

Function `cublasCcopy()`

```c
void
cublasCcopy (int n, const cuComplex *x, int incx,
cuComplex *y, int incy)
```

copies the single precision complex vector `x` to the single precision complex vector `y`.

For `i = 0` to `n-1`, it copies

\[ x[lx + i \times incx] \rightarrow y[ly + i \times incy], \]

where

\[ lx = 1 \text{ if } incx \geq 0, \text{ else } \]
\[ lx = 1 + (1 - n) \times incx; \]

`ly` is defined in a similar way using `incy`.

Input

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>n</code></td>
<td>number of elements in input vectors</td>
</tr>
<tr>
<td><code>x</code></td>
<td>single precision complex vector with <code>n</code> elements</td>
</tr>
<tr>
<td><code>incx</code></td>
<td>storage spacing between elements of <code>x</code></td>
</tr>
<tr>
<td><code>y</code></td>
<td>single precision complex vector with <code>n</code> elements</td>
</tr>
<tr>
<td><code>incy</code></td>
<td>storage spacing between elements of <code>y</code></td>
</tr>
</tbody>
</table>

Output

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>y</code></td>
<td>contains single precision complex vector <code>x</code></td>
</tr>
</tbody>
</table>

Reference: http://www.netlib.org/blas/ccopy.f
Error status for this function can be retrieved via `cublasGetError()`.

**Error Status**

<table>
<thead>
<tr>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CUBLAS_STATUS_NOT_INITIALIZED</code></td>
<td>if CUBLAS library was not initialized</td>
</tr>
<tr>
<td><code>CUBLAS_STATUS_EXECUTION_FAILED</code></td>
<td>if function failed to launch on GPU</td>
</tr>
</tbody>
</table>

### Function `cublasCscal()`

```c
void
cublasCscal (int n, cuComplex alpha, cuComplex *x, int incx)
```

replaces single precision complex vector `x` with single precision complex `alpha * x`.

For `i = 0` to `n-1`, it replaces

\[
x[lx+i*incx] \text{ with } alpha \times x[lx+i*incx],
\]

where

\[
 lx = 1 \text{ if incx } \geq 0, \text{ else } lx = 1 + (1 - n) \times \text{incx}.
\]

**Input**

- `n` number of elements in input vector
- `alpha` single precision complex scalar multiplier
- `x` single precision complex vector with `n` elements
- `incx` storage spacing between elements of `x`

**Output**

- `x` single precision complex result (unchanged if `n <= 0` or `incx <= 0`)

**Reference:** [http://www.netlib.org/blas/cscal.f](http://www.netlib.org/blas/cscal.f)

Error status for this function can be retrieved via `cublasGetError()`.

**Error Status**

<table>
<thead>
<tr>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CUBLAS_STATUS_NOT_INITIALIZED</code></td>
<td>if CUBLAS library was not initialized</td>
</tr>
<tr>
<td><code>CUBLAS_STATUS_EXECUTION_FAILED</code></td>
<td>if function failed to launch on GPU</td>
</tr>
</tbody>
</table>
Function cublasCsscal()

void

cublasCsscal (int n, float alpha, cuComplex *x, int incx)

replaces single precision complex vector \( x \) with single precision complex \( \alpha \times x \). For \( i = 0 \) to \( n-1 \), it replaces

\[
x[lx+i \times incx] \text{ with } \alpha \times x[lx+i \times incx],
\]

where

- \( lx = 1 \) if \( incx \geq 0 \), else
- \( lx = 1 + (1-n) \times incx \).

Input

- \( n \): number of elements in input vector
- \( \alpha \): single precision scalar multiplier
- \( x \): single precision complex vector with \( n \) elements
- \( incx \): storage spacing between elements of \( x \)

Output

- \( x \): single precision complex result (unchanged if \( n \leq 0 \) or \( incx \leq 0 \))

Reference: [http://www.netlib.org/blas/csscal.f](http://www.netlib.org/blas/csscal.f)

Error status for this function can be retrieved via cublasGetError().

Error Status

- `CUBLAS_STATUS_NOT_INITIALIZED` if CUBLAS library was not initialized
- `CUBLAS_STATUS_EXECUTION_FAILED` if function failed to launch on GPU

Function cublasCswap()

void

cublasCswap (int n, const cuComplex *x, int incx,
            cuComplex *y, int incy)

interchanges the single precision complex vector \( x \) with the single precision complex vector \( y \). For \( i = 0 \) to \( n-1 \), it interchanges

\[
x[lx+i \times incx] \text{ with } y[ly+i \times incy],
\]
where

\[
1x = 1 \text{ if } \text{incx} \geq 0, \text{ else } \\
1x = 1 + (1 - n) \times \text{incx}; \\
\]

is defined in a similar way using \( \text{incy} \).

**Input**

- \( n \) number of elements in input vectors
- \( x \) single precision complex vector with \( n \) elements
- \( \text{incx} \) storage spacing between elements of \( x \)
- \( y \) single precision complex vector with \( n \) elements
- \( \text{incy} \) storage spacing between elements of \( y \)

**Output**

- \( x \) contains single precision complex vector \( y \)
- \( y \) contains single precision complex vector \( x \)

**Reference:** [http://www.netlib.org/blas/cswap.f](http://www.netlib.org/blas/cswap.f)

Error status for this function can be retrieved via `cublasGetError()`.

**Error Status**

- `CUBLAS_STATUS_NOT_INITIALIZED` if CUBLAS library was not initialized
- `CUBLAS_STATUS_EXECUTION_FAILED` if function failed to launch on GPU

**Function cublasScasum()**

```c
float cublasScasum (int n, const cuDouble *x, int incx)
```

takes the sum of the absolute values of a complex vector and returns a single precision result. Note that this is not the L1 norm of the vector. The result is the sum from 0 to \( n-1 \) of

\[
\text{abs}(\text{real}(x[1x+i \times \text{incx}]))) + \text{abs}(\text{imag}(x[1x+i \times \text{incx}]))
\]

where

\[
1x = 1 \text{ if } \text{incx} \leq 0, \text{ else } \\
1x = 1 + (1 - n) \times \text{incx}.
\]
Input

\( n \)  
number of elements in input vector

\( x \)  
single precision complex vector with \( n \) elements

\( \text{incx} \)  
storage spacing between elements of \( x \)

Output

returns the single precision sum of absolute values of real and imaginary parts  
(returns zero if \( n \leq 0, \text{incx} \leq 0 \), or if an error occurred)

Reference: [http://www.netlib.org/blas/scasum.f](http://www.netlib.org/blas/scasum.f)

Error status for this function can be retrieved via `cublasGetError()`.  

Error Status

- `CUBLAS_STATUS_NOT_INITIALIZED` if CUBLAS library was not initialized
- `CUBLAS_STATUS_EXECUTION_FAILED` if function failed to launch on GPU
CHAPTER 3

BLAS2 and BLAS3 Functions

The Level 2 Basic Linear Algebra Subprograms (BLAS2) are functions that perform matrix-vector operations, while Level 3 Basic Linear Algebra Subprograms (BLAS3) perform matrix-matrix operations. The CUBLAS implementations are described in these sections:

- “Single Precision BLAS2 Functions” on page 27
- “Single Precision Complex BLAS2 Functions” on page 48 *(Not yet implemented)*
- “Single Precision BLAS3 Functions” on page 49
- “Single Precision Complex BLAS3 Functions” on page 59
CHAPTER 3  BLAS2 and BLAS3 Functions

Single Precision BLAS2 Functions

The single precision BLAS2 functions are as follows:

- “Function cublasSgbmv()” on page 27
- “Function cublasSgemv()” on page 29
- “Function cublasSger()” on page 30
- “Function cublasSsbmv()” on page 31
- “Function cublasSspmv()” on page 33
- “Function cublasSspr()” on page 34
- “Function cublasSspr2()” on page 35
- “Function cublasSsymv()” on page 36
- “Function cublasSsyr()” on page 37
- “Function cublasSsyr2()” on page 38
- “Function cublasStbmv()” on page 40
- “Function cublasStbsv()” on page 41
- “Function cublasStpmv()” on page 43
- “Function cublasStpsv()” on page 44
- “Function cublasStrmv()” on page 45
- “Function cublasStrsv()” on page 47

Function cublasSgbmv()

```c
void cublasSgbmv (char trans, int m, int n, int kl, int ku,
                 float alpha, const float *A, int lda,
                 const float *x, int incx, float beta,
                 float *y, int incy);
```

performs one of the matrix-vector operations

\[
y = \alpha \cdot \text{op}(A) \cdot x + \beta \cdot y,
\]

where \( \text{op}(A) = A \) or \( \text{op}(A) = A^T \),
alpha and beta are single precision scalars, and x and y are single precision vectors. A is an \( m \times n \) band matrix consisting of single precision elements with \( kl \) subdiagonals and \( ku \) superdiagonals.

Input

- \texttt{trans} specifies \( \text{op}(A) \). If \texttt{trans} == 'N' or 'n', \( \text{op}(A) = A \). If \texttt{trans} == 'T', 't', 'C', or 'c', \( \text{op}(A) = A^T \).
- \( m \) the number of rows of matrix \( A \); \( m \) must be at least zero.
- \( n \) the number of columns of matrix \( A \); \( n \) must be at least zero.
- \( kl \) the number of subdiagonals of matrix \( A \); \( kl \) must be at least zero.
- \( ku \) the number of superdiagonals of matrix \( A \); \( ku \) must be at least zero.
- \( \alpha \) single precision scalar multiplier applied to \( \text{op}(A) \).
- \( A \) single precision array of dimensions \((l \text{da}, n)\). The leading \((kl + ku + 1) \times n \) part of array \( A \) must contain the band matrix \( A \), supplied column by column, with the leading diagonal of the matrix in row \( ku+1 \) of the array, the first superdiagonal starting at position 2 in row \( ku \), the first subdiagonal starting at position 1 in row \( ku+2 \), and so on. Elements in the array \( A \) that do not correspond to elements in the band matrix (such as the top left \( ku \times ku \) triangle) are not referenced.
- \( l \text{da} \) leading dimension of \( A \); \( l \text{da} \) must be at least \( kl + ku + 1 \).
- \( x \) single precision array of length at least \((1 + (n-1) \times \text{abs}(\text{incx}))\) when \texttt{trans} == 'N' or 'n', and at least \((1 + (m-1) \times \text{abs}(\text{incx}))\) otherwise.
- \( \text{incx} \) storage spacing between elements of \( x \); \( \text{incx} \) must not be zero.
- \( \beta \) single precision scalar multiplier applied to vector \( y \). If \( \beta \) is zero, \( y \) is not read.
- \( y \) single precision array of length at least \((1 + (m-1) \times \text{abs}(\text{incy}))\) when \texttt{trans} == 'N' or 'n' and at least \((1 + (n-1) \times \text{abs}(\text{incy}))\) otherwise. If \( \beta \) is zero, \( y \) is not read.
- \( \text{incy} \) storage spacing between elements of \( y \); \( \text{incy} \) must not be zero.

Output

\( y \) updated according to \( y = \alpha \text{op}(A) \times x + \beta y \).

Reference: http://www.netlib.org/blas/sgbmv.f
CHAPTER 3 \( \text{BLAS2 and BLAS3 Functions} \)

Error status for this function can be retrieved via \texttt{cublasGetError()}.

\begin{center}
\begin{tabular}{ll}
\textbf{CUBLAS_STATUS_NOT_INITIALIZED} & if CUBLAS library was not initialized \\
\textbf{CUBLAS_STATUS_INVALID_VALUE} & if \( m < 0, n < 0, kl < 0, ku < 0, \)
\quad incx == 0, or incy == 0 \\
\textbf{CUBLAS_STATUS_EXECUTION_FAILED} & if function failed to launch on GPU
\end{tabular}
\end{center}

Function \texttt{cublasSgemv()}

\begin{verbatim}
void 
cublasSgemv (char trans, int m, int n, float alpha, 
        const float *A, int lda, const float *x, 
        int incx, float beta, float *y, int incy)
\end{verbatim}

performs one of the matrix-vector operations

\[ y = \alpha \text{op}(A) * x + \beta \text{ } y, \]

where \( \text{op}(A) = A \text{ or } A^T, \)

\( \alpha \) and \( \beta \) are single precision scalars, and \( x \) and \( y \) are single precision vectors. \( A \) is an \( m \times n \) matrix consisting of single precision elements. Matrix \( A \) is stored in column-major format, and \( \text{lda} \) is the leading dimension of the two-dimensional array in which \( A \) is stored.

**Input**

- \( \text{trans} \): specifies \( \text{op}(A) \). If \( \text{trans} = 'N' \) or \( 'n' \), \( \text{op}(A) = A \).
  - If \( \text{trans} = 'T', 't', 'C', \) or \( 'c' \), \( \text{op}(A) = A^T \).
- \( m \): specifies the number of rows of matrix \( A; \) \( m \) must be at least zero.
- \( n \): specifies the number of columns of matrix \( A; \) \( n \) must be at least zero.
- \( \alpha \): single precision scalar multiplier applied to \( \text{op}(A) \).
- \( A \): single precision array of dimensions \((\text{lda}, n)\) if \( \text{trans} = 'N' \) or \( 'n' \), of dimensions \((\text{lda}, m)\) otherwise; \( \text{lda} \) must be at least \( \max(1, m) \) if \( \text{trans} = 'N' \) or \( 'n' \) and at least \( \max(1, n) \) otherwise.
- \( \text{lda} \): leading dimension of two-dimensional array used to store matrix \( A \).
- \( x \): single precision array of length at least \( (1 + (n-1) \times \text{abs(incx)}) \) if \( \text{trans} = 'N' \) or \( 'n' \), else at least \( (1 + (m-1) \times \text{abs(incx)}) \).
- \( \text{incx} \): specifies the storage spacing for elements of \( x; \) \( \text{incx} \) must not be zero.
- \( \beta \): single precision scalar multiplier applied to vector \( y \). If \( \beta \) is zero, \( y \) is not read.
Function `cublasSger()`

```c
void

cublasSger (int m, int n, float alpha, const float *x,
       int incx, const float *y, int incy, float *A,
       int lda)
```

performs the symmetric rank 1 operation

\[
A = \alpha x y^T + A,
\]

where \( \alpha \) is a single precision scalar, \( x \) is an \( m \)-element single precision vector, \( y \) is an \( n \)-element single precision vector, and \( A \) is an \( m \times n \) matrix consisting of single precision elements. Matrix \( A \) is stored in column-major format, and \( lda \) is the leading dimension of the two-dimensional array used to store \( A \).

Input

- \( m \): specifies the number of rows of the matrix \( A \); \( m \) must be at least zero.
- \( n \): specifies the number of columns of matrix \( A \); \( n \) must be at least zero.
- \( \alpha \): single precision scalar multiplier applied to \( x \) \( y^T \).
- \( x \): single precision array of length at least \( 1 + (m-1) \times \text{abs}(\text{incx}) \).
- \( \text{incx} \): the storage spacing between elements of \( x \); \( \text{incx} \) must not be zero.
- \( y \): single precision array of length at least \( 1 + (n-1) \times \text{abs}(\text{incy}) \).
CHAPTER 3 \hspace{1cm} BLAS2 and BLAS3 Functions

Input (continued)

\begin{verbatim}
incy \hspace{0.5cm} \text{the storage spacing between elements of y; incy must not be zero.}
A \hspace{0.5cm} \text{single precision array of dimensions (lda, n).}
lda \hspace{0.5cm} \text{leading dimension of two-dimensional array used to store matrix A.}
\end{verbatim}

Output

\begin{verbatim}
A \hspace{1cm} \text{updated according to } A = \alpha \times x \times y + A.
\end{verbatim}

Reference: \url{http://www.netlib.org/blas/sger.f}

Error status for this function can be retrieved via \texttt{cublasGetError()).

Error Status

\begin{verbatim}
CUBLAS_STATUS_NOT_INITIALIZED \hspace{1cm} \text{if CUBLAS library was not initialized}
CUBLAS_STATUS_INVALID_VALUE \hspace{1cm} \text{if } m < 0, n < 0, \text{incx} == 0, \text{or incy} == 0
CUBLAS_STATUS_EXECUTION_FAILED \hspace{1cm} \text{if function failed to launch on GPU}
\end{verbatim}

Function \texttt{cublasSsbmv()}

\begin{verbatim}
void 
cublasSsbmv (char uplo, int n, int k, float alpha, 
            const float *A, int lda, const float *x, 
            int incx, float beta, float *y, int incy)
\end{verbatim}

performs the matrix-vector operation

\begin{verbatim}
y = \alpha \times A \times x + \beta \times y,
\end{verbatim}

where \( \alpha \) and \( \beta \) are single precision scalars, and \( x \) and \( y \) are \( n \)-element single precision vectors. \( A \) is an \( n \times n \) symmetric band matrix consisting of single precision elements, with \( k \) superdiagonals and the same number of subdiagonals.

Input

\begin{verbatim}
uplo \hspace{1cm} \text{specifies whether the upper or lower triangular part of the symmetric}
\hspace{2cm} \text{band matrix } A \text{ is being supplied. If } \text{uplo} == 'U' \text{ or 'u'}, \text{the upper}
\hspace{2cm} \text{triangular part is being supplied. If } \text{uplo} == 'L' \text{ or 'l'}, \text{the lower}
\hspace{2cm} \text{triangular part is being supplied.}
\end{verbatim}

\begin{verbatim}
n \hspace{1cm} \text{specifies the number of rows and the number of columns of the}
\hspace{2cm} \text{symmetric matrix } A; \text{ } n \text{ must be at least zero.}
\end{verbatim}
CUDA CUBLAS Library

Input (continued)

- **k** specifies the number of superdiagonals of matrix A. Since the matrix is symmetric, this is also the number of subdiagonals; k must be at least zero.
- **alpha** single precision scalar multiplier applied to A * x.
- **A** single precision array of dimensions (lda, n). When uplo == 'U' or 'u', the leading (k+1)×n part of array A must contain the upper triangular band of the symmetric matrix, supplied column by column, with the leading diagonal of the matrix in row k+1 of the array, the first superdiagonal starting at position 2 in row k, and so on. The top left k×k triangle of the array A is not referenced. When uplo == 'L' or 'l', the leading (k+1)×n part of the array A must contain the lower triangular band part of the symmetric matrix, supplied column by column, with the leading diagonal of the matrix in row 1 of the array, the first subdiagonal starting at position 1 in row 2, and so on. The bottom right k×k triangle of the array A is not referenced.
- **lda** leading dimension of A; lda must be at least k+1.
- **x** single precision array of length at least \((1+(n-1)\times\text{abs}(\text{incx})))
- **incx** storage spacing between elements of x; incx must not be zero.
- **beta** single precision scalar multiplier applied to vector y. If beta is zero, y is not read.
- **y** single precision array of length at least \((1+(n-1)\times\text{abs}(\text{incy})))
  - If beta is zero, y is not read.
- **incy** storage spacing between elements of y; incy must not be zero.

Output

- **y** updated according to \(y = \alpha \times A \times x + \beta \times y\).

Reference: [http://www.netlib.org/blas/ssbmv.f](http://www.netlib.org/blas/ssbmv.f)

Error status for this function can be retrieved via `cublasGetError()`. Error Status

- **CUBLAS_STATUS_NOT_INITIALIZED** if CUBLAS library was not initialized
- **CUBLAS_STATUS_INVALID_VALUE** if k < 0, n < 0, incx == 0, or incy == 0
- **CUBLAS_STATUS_EXECUTION_FAILED** if function failed to launch on GPU
Function cublasSspmv()

```c
void
cublasSspmv (char uplo, int n, float alpha,
            const float *AP, const float *x, int incx,
            float beta, float *y, int incy)
```

performs the matrix-vector operation

\[ y = \alpha A x + \beta y, \]

where \( \alpha \) and \( \beta \) are single precision scalars, and \( x \) and \( y \) are \( n \)-element single precision vectors. \( A \) is a symmetric \( n \times n \) matrix that consists of single precision elements and is supplied in packed form.

**Input**

- \( \text{uplo} \) specifies whether the matrix data is stored in the upper or the lower triangular part of array \( \text{AP} \). If \( \text{uplo} == 'U' \) or \( 'u' \), the upper triangular part of \( A \) is supplied in \( \text{AP} \). If \( \text{uplo} == 'L' \) or \( 'l' \), the lower triangular part of \( A \) is supplied in \( \text{AP} \).
- \( n \) the number of rows and columns of matrix \( A \); \( n \) must be at least zero.
- \( \alpha \) single precision array with at least \( n \) elements. If \( \text{uplo} == 'U' \) or \( 'u' \), array \( \text{AP} \) contains the upper triangular part of the symmetric matrix \( A \), packed sequentially, column by column; that is, if \( i <= j, A[i,j] \) is stored in \( \text{AP}[i+(j*(j+1)/2)] \). If \( \text{uplo} == 'L' \) or \( 'l' \), the array \( \text{AP} \) contains the lower triangular part of the symmetric matrix \( A \), packed sequentially, column by column; that is, if \( i >= j, A[i,j] \) is stored in \( \text{AP}[i+(2*n-j+1)*j/2] \).
- \( x \) single precision array of length at least \( 1+(n-1)*\text{abs}(\text{incx}) \).
- \( \text{incx} \) storage spacing between elements of \( x \); \( \text{incx} \) must not be zero.
- \( \beta \) single precision scalar multiplier applied to vector \( y \). If \( \beta \) is zero, \( y \) is not read.
- \( y \) single precision array of length at least \( 1+(n-1)*\text{abs}(\text{incy}) \).
- \( \text{incy} \) storage spacing between elements of \( y \); \( \text{incy} \) must not be zero.

**Output**

- \( y \) updated according to \( y = \alpha A x + \beta y \).

**Reference:** [http://www.netlib.org/blas/sspmv.f](http://www.netlib.org/blas/sspmv.f)
Error status for this function can be retrieved via `cublasGetError()`.  

<table>
<thead>
<tr>
<th>Error Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUBLAS_STATUS_NOT_INITIALIZED</td>
<td>if CUBLAS library was not initialized</td>
</tr>
<tr>
<td>CUBLAS_STATUS_INVALID_VALUE</td>
<td>if ( n &lt; 0, \text{incx} = 0, \text{incy} = 0 )</td>
</tr>
<tr>
<td>CUBLAS_STATUS_EXECUTION_FAILED</td>
<td>if function failed to launch on GPU</td>
</tr>
</tbody>
</table>

**Function `cublasSspr()`**

```c
void

`cublasSspr` (char `uplo`, int `n`, float `alpha`,
               const float *`x`, int `incx`, float *`AP`)
```
performs the symmetric rank 1 operation

\[
A = \alpha x x^T + A,
\]

where `alpha` is a single precision scalar, and `x` is an \( n \)-element single precision vector. `A` is a symmetric \( n \times n \) matrix that consists of single precision elements and is supplied in packed form.

**Input**

- `uplo` specifies whether the matrix data is stored in the upper or the lower triangular part of `AP`. If `uplo` == 'U' or 'u', the upper triangular part of `A` is supplied in `AP`. If `uplo` == 'L' or 'l', the lower triangular part of `A` is supplied in `AP`.
- `n` the number of rows and columns of matrix `A`; `n` must be at least zero.
- `alpha` single precision scalar multiplier applied to \( x x^T \).
- `x` single precision array of length at least \( (1 + (n - 1) \times \text{abs(incx)}) \).
- `incx` storage spacing between elements of `x`; `incx` must not be zero.
- `AP` single precision array with at least \( n \times (n + 1)/2 \) elements. If `uplo` == 'U' or 'u', array `AP` contains the upper triangular part of the symmetric matrix `A`, packed sequentially, column by column; that is, if \( i <= j \), \( A[i,j] \) is stored in `AP[i + (j * (j + 1)/2)]`. If `uplo` == 'L' or 'l', the array `AP` contains the lower triangular part of the symmetric matrix `A`, packed sequentially, column by column; that is, if \( i >= j \), \( A[i,j] \) is stored in `AP[i + ((2 * n - j + 1) * j)/2]`.

**Output**

- `A` updated according to \( A = \alpha x x^T + A \).

**Reference:** [http://www.netlib.org/blas/sspr.f](http://www.netlib.org/blas/sspr.f)
Error status for this function can be retrieved via `cublasGetError()`.

**Error Status**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUBLAS_STATUS_NOT_INITIALIZED</td>
<td>if CUBLAS library was not initialized</td>
</tr>
<tr>
<td>CUBLAS_STATUS_INVALID_VALUE</td>
<td>if ( n &lt; 0 ) or ( \text{incx} == 0 )</td>
</tr>
<tr>
<td>CUBLAS_STATUS_EXECUTION_FAILED</td>
<td>if function failed to launch on GPU</td>
</tr>
</tbody>
</table>

**Function cublasSspr2()**

```c
void

cublasSspr2 (char uplo, int n, float alpha,
              const float *x, int incx, const float *y,
              int incy, float *AP)
```

performs the symmetric rank 2 operation

\[
A = \alpha x^T y + \alpha y^T x + A,
\]

where \( \alpha \) is a single precision scalar, and \( x \) and \( y \) are \( n \)-element single precision vectors. \( A \) is a symmetric \( n \times n \) matrix that consists of single precision elements and is supplied in packed form.

**Input**

<table>
<thead>
<tr>
<th>uplo</th>
<th>specifies whether the matrix data is stored in the upper or the lower triangular part of array ( A ). If ( \text{uplo} == 'U' ) or ( 'u' ), only the upper triangular part of ( A ) may be referenced and the lower triangular part of ( A ) is inferred. If ( \text{uplo} == 'L' ) or ( 'l' ), only the lower triangular part of ( A ) may be referenced and the upper triangular part of ( A ) is inferred.</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n )</td>
<td>the number of rows and columns of matrix ( A ); ( n ) must be at least zero.</td>
</tr>
<tr>
<td>( \alpha )</td>
<td>single precision scalar multiplier applied to ( x^T y + \alpha y x^T ).</td>
</tr>
<tr>
<td>( x )</td>
<td>single precision array of length at least ( (1 + (n - 1) \cdot \text{abs(incx)}) ).</td>
</tr>
<tr>
<td>( \text{incx} )</td>
<td>storage spacing between elements of ( x ); ( \text{incx} ) must not be zero.</td>
</tr>
<tr>
<td>( y )</td>
<td>single precision array of length at least ( (1 + (n - 1) \cdot \text{abs(incy)}) ).</td>
</tr>
<tr>
<td>( \text{incy} )</td>
<td>storage spacing between elements of ( y ); ( \text{incy} ) must not be zero.</td>
</tr>
<tr>
<td>( \text{AP} )</td>
<td>single precision array with at least ( \frac{n \cdot (n + 1)}{2} ) elements. If ( \text{uplo} == 'U' ) or ( 'u' ), array ( \text{AP} ) contains the upper triangular part of the symmetric matrix ( A ), packed sequentially, column by column; that is, if ( i &lt;= j ), ( A[i,j] ) is stored in ( \text{AP}[i + (j + 1)/2] ). If ( \text{uplo} == 'L' ) or ( 'l' ), the array ( \text{AP} ) contains the lower triangular part of the symmetric matrix ( A ), packed sequentially, column by column; that is, if ( i &gt;= j ), ( A[i,j] ) is stored in ( \text{AP}[i + ((2 \cdot n - j + 1) \cdot j)/2] ).</td>
</tr>
</tbody>
</table>
CUDA

NVIDIA

Output

\[ A \] updated according to \[ A = \alpha x \times y^2 + \alpha y \times x^2 + A. \]

Reference: [http://www.netlib.org/blas/sspr2.f](http://www.netlib.org/blas/sspr2.f)

Error status for this function can be retrieved via `cublasGetError()`.

Error Status

- `CUBLAS_STATUS_NOT_INITIALIZED` if CUBLAS library was not initialized
- `CUBLAS_STATUS_INVALID_VALUE` if \( n < 0, \text{incx} = 0, \) or \( \text{incy} = 0 \)
- `CUBLAS_STATUS_EXECUTION_FAILED` if function failed to launch on GPU

Function `cublasSsymv()`

```c
void
cublasSsymv (char uplo, int n, float alpha,
             const float *A, int lda, const float *x,
             int incx, float beta, float *y, int incy)
```

performs the matrix-vector operation

\[
y = \alpha A \times x + \beta y,
\]

where \( \alpha \) and \( \beta \) are single precision scalars, and \( x \) and \( y \) are \( n \)-element single precision vectors. \( A \) is a symmetric \( n \times n \) matrix that consists of single precision elements and is stored in either upper or lower storage mode.

Input

- `uplo` specifies whether the upper or lower triangular part of the array \( A \) is referenced. If \( \text{uplo} = 'U' \) or \( 'u' \), the symmetric matrix \( A \) is stored in upper storage mode; that is, only the upper triangular part of \( A \) is referenced while the lower triangular part of \( A \) is inferred. If \( \text{uplo} = 'L' \) or \( 'l' \), the symmetric matrix \( A \) is stored in lower storage mode; that is, only the lower triangular part of \( A \) is referenced while the upper triangular part of \( A \) is inferred.
- `n` specifies the number of rows and the number of columns of the symmetric matrix \( A; n \) must be at least zero.
- `alpha` single precision scalar multiplier applied to \( A \times x \).
Input (continued)

A         single precision array of dimensions (lda, n). If uplo == 'U' or 'u', the leading n×n upper triangular part of the array A must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of A is not referenced. If uplo == 'L' or 'l', the leading n×n lower triangular part of the array A must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of A is not referenced.

lda       leading dimension of A; lda must be at least max(1, n).

x         single precision array of length at least (1+(n-1) * abs(incx)).

incx      storage spacing between elements of x; incx must not be zero.

beta      single precision scalar multiplier applied to vector y. If beta is zero, y is not read.

y         single precision array of length at least (1+(n-1) * abs(incy)). If beta is zero, y is not read.

incy      storage spacing between elements of y; incy must not be zero.

Output

y         updated according to $y = \alpha A x + \beta y$.

Reference: [http://www.netlib.org/blas/ssymv.f](http://www.netlib.org/blas/ssymv.f)

Error status for this function can be retrieved via `cublasGetError()`.

Error Status

<table>
<thead>
<tr>
<th>Error Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUBLAS_STATUS_NOT_INITIALIZED</td>
<td>if CUBLAS library was not initialized</td>
</tr>
<tr>
<td>CUBLAS_STATUS_INVALID_VALUE</td>
<td>if n &lt; 0, incx == 0, or incy == 0</td>
</tr>
<tr>
<td>CUBLAS_STATUS_EXECUTION_FAILED</td>
<td>if function failed to launch on GPU</td>
</tr>
</tbody>
</table>

Function cublasSsyr()

```c
void
    cublasSsyr (char uplo, int n, float alpha,
                const float *x, int incx, float *A, int lda)
```

performs the symmetric rank 1 operation

$$A = \alpha x^T A + \alpha^T x A,$$

where `alpha` is a single precision scalar, `x` is an n-element single precision vector, and A is an n×n symmetric matrix consisting of single
precision elements. \( A \) is stored in column-major format, and \( \text{lda} \) is the leading dimension of the two-dimensional array containing \( A \).

**Input**

- **uplo** specifies whether the matrix data is stored in the upper or the lower triangular part of array \( A \). If \( \text{uplo} == \text{'U'} \) or \( \text{'u'} \), only the upper triangular part of \( A \) is referenced. If \( \text{uplo} == \text{'L'} \) or \( \text{'l'} \), only the lower triangular part of \( A \) is referenced.
- **n** is the number of rows and columns of matrix \( A \); \( n \) must be at least zero.
- **alpha** is a single precision scalar multiplier applied to \( x \times x^T \).
- **x** is a single precision array of length at least \( 1 + (n - 1) \times \text{abs}(\text{incx}) \).
- **incx** is the storage spacing between elements of \( x \); \( \text{incx} \) must not be zero.
- **A** is a single precision array of dimensions \( \text{(lda, n)} \). If \( \text{uplo} == \text{'U'} \) or \( \text{'u'} \), \( A \) contains the upper triangular part of the symmetric matrix, and the strictly lower triangular part is not referenced. If \( \text{uplo} == \text{'L'} \) or \( \text{'l'} \), \( A \) contains the lower triangular part of the symmetric matrix, and the strictly upper triangular part is not referenced.
- **lda** is the leading dimension of the two-dimensional array containing \( A \); \( \text{lda} \) must be at least \( \max(1, n) \).

**Output**

- \( A \) is updated according to \( A = \alpha \times x \times x^T + A \).

**Reference:** [http://www.netlib.org/blas/ssyr.f](http://www.netlib.org/blas/ssyr.f)

Error status for this function can be retrieved via `cublasGetError()`.

**Error Status**

- **CUBLAS_STATUS_NOT_INITIALIZED** if CUBLAS library was not initialized
- **CUBLAS_STATUS_INVALID_VALUE** if \( n < 0 \) or \( \text{incx} == 0 \)
- **CUBLAS_STATUS_EXECUTION_FAILED** if function failed to launch on GPU

**Function cublasSsyr2()**

```c
void
cublasSsyr2 (char uplo, int n, float alpha,
            const float *x, int incx, const float *y,
            int incy, float *A, int lda)
```

performs the symmetric rank 2 operation

\[
A = \alpha \times x \times y^T + \alpha \times y \times x^T + A,
\]
where $\alpha$ is a single precision scalar, $x$ and $y$ are $n$-element single precision vectors, and $A$ is an $n \times n$ symmetric matrix consisting of single precision elements.

Input

- $\text{uplo}$ specifies whether the matrix data is stored in the upper or the lower triangular part of array $A$. If $\text{uplo} == 'U'$ or 'u', only the upper triangular part of $A$ is referenced and the lower triangular part of $A$ is inferred. If $\text{uplo} == 'L'$ or 'l', only the lower triangular part of $A$ is referenced and the upper triangular part of $A$ is inferred.
- $n$ is the number of rows and columns of matrix $A$; $n$ must be at least zero.
- $\alpha$ is a single precision scalar multiplier applied to $x^TX + y^TY$.
- $x$ is a single precision array of length at least $(1+(n-1)\cdot\text{abs}(\text{incx}))$.
- $\text{incx}$ is the storage spacing between elements of $x$; $\text{incx}$ must not be zero.
- $y$ is a single precision array of length at least $(1+(n-1)\cdot\text{abs}(\text{incy}))$.
- $\text{incy}$ is the storage spacing between elements of $y$; $\text{incy}$ must not be zero.
- $A$ is a single precision array of dimensions $(\text{lda}, n)$. If $\text{uplo} == 'U'$ or 'u', $A$ contains the upper triangular part of the symmetric matrix, and the strictly lower triangular part is not referenced. If $\text{uplo} == 'L'$ or 'l', $A$ contains the lower triangular part of the symmetric matrix, and the strictly upper triangular part is not referenced.
- $\text{lda}$ is the leading dimension of $A$; $\text{lda}$ must be at least $\text{max}(1, n)$.

Output

- $A$ is updated according to $A = \alpha \cdot x^TX + \alpha \cdot y^TY$.

Reference: [http://www.netlib.org/blas/ssyr2.f](http://www.netlib.org/blas/ssyr2.f)

Error status for this function can be retrieved via `cublasGetError()`. Error Status

- `CUBLAS_STATUS_NOT_INITIALIZED` if CUBLAS library was not initialized
- `CUBLAS_STATUS_INVALID_VALUE` if $n < 0$, $\text{incx} == 0$, or $\text{incy} == 0$
- `CUBLAS_STATUS_EXECUTION_FAILED` if function failed to launch on GPU
Function cublasStbmv()

```c
void cublasStbmv (char uplo, char trans, char diag, int n,
                  int k, const float *A, int lda, float *x,
                  int incx)
```

performs one of the matrix-vector operations

\[ x = op(A) \times x, \]

where \( op(A) = A \text{ or } A^T \).

\( x \) is an \( n \)-element single precision vector, and \( A \) is an \( n \times n \), unit or non-unit, upper or lower, triangular band matrix consisting of single precision elements.

**Input**

- **uplo** specifies whether the matrix \( A \) is an upper or lower triangular band matrix. If \( uplo == 'U' \) or \('u' \), \( A \) is an upper triangular band matrix. If \( uplo == 'L' \) or \('l' \), \( A \) is a lower triangular band matrix.
- **trans** specifies \( op(A) \). If \( trans == 'N' \) or \('n' \), \( op(A) = A \). If \( trans == 'T' \), \('t' \), \('C' \), or \('c' \), \( op(A) = A^T \).
- **diag** specifies whether or not matrix \( A \) is unit triangular. If \( diag == 'U' \) or \('u' \), \( A \) is assumed to be unit triangular. If \( diag == 'N' \) or \('n' \), \( A \) is not assumed to be unit triangular.
- **n** specifies the number of rows and columns of the matrix \( A \); \( n \) must be at least zero. In the current implementation \( n \) must not exceed 4070.
- **k** specifies the number of superdiagonals or subdiagonals. If \( uplo == 'U' \) or \('u' \), \( k \) specifies the number of superdiagonals. If \( uplo == 'L' \) or \('l' \), \( k \) specifies the number of subdiagonals; \( k \) must at least be zero.
- **A** single precision array of dimension \((lda,n)\). If \( uplo == 'U' \) or \('u' \), the leading \((k+1)\times n\) part of the array \( A \) must contain the upper triangular band matrix, supplied column by column, with the leading diagonal of the matrix in row \( k+1 \) of the array, the first superdiagonal starting at position 2 in row \( k \), and so on. The top left \( k \times k \) triangle of the array \( A \) is not referenced. If \( uplo == 'L' \) or \('l' \), the leading \((k+1)\times n\) part of the array \( A \) must contain the lower triangular band matrix, supplied column by column, with the leading diagonal of the matrix in row 1 of the array, the first subdiagonal starting at position 1 in row 2, and so on. The bottom right \( k \times k \) triangle of the array is not referenced.
- **lda** is the leading dimension of \( A \); \( lda \) must be at least \( k+1 \).
CHAPTER 3  BLAS2 and BLAS3 Functions

Input (continued)

\[ x \] single precision array of length at least \((1+(n-1) \times \text{abs}(\text{incx}))\). On entry, \(x\) contains the source vector. On exit, \(x\) is overwritten with the result vector.

\[ \text{incx} \] specifies the storage spacing for elements of \(x\); \(\text{incx}\) must not be zero.

Output

\[ x \] updated according to \(x = \text{op}(A) \times x\).


Error status for this function can be retrieved via `cublasGetError()`.

Error Status

- **CUBLAS_STATUS_NOT_INITIALIZED** if CUBLAS library was not initialized
- **CUBLAS_STATUS_INVALID_VALUE** if \(n < 0\), \(n > 4070\), \(k < 0\), or \(\text{incx} = 0\)
- **CUBLAS_STATUS_EXECUTION_FAILED** if function failed to launch on GPU

Function `cublasStbsv()`

```c
void cublasStbsv (char uplo, char trans, char diag, int n, int k, const float *A, int lda, float X, int incx)
```

solves one of the systems of equations

\[
\text{op}(A) \times x = b,
\]

where \(\text{op}(A) = A\) or \(\text{op}(A) = A^T\),

\(b\) and \(x\) are \(n\)-element vectors, and \(A\) is an \(n \times n\), unit or non-unit, upper or lower, triangular band matrix with \(k+1\) diagonals.

No test for singularity or near-singularity is included in this function. Such tests must be performed before calling this function.

Input

- **uplo** specifies whether the matrix is an upper or lower triangular band matrix: If \(\text{uplo} = 'U'\) or \('u'\), \(A\) is an upper triangular band matrix. If \(\text{uplo} = 'L'\) or \('l'\), \(A\) is a lower triangular band matrix.

- **trans** specifies \(\text{op}(A)\). If \(\text{trans} = 'N'\) or \('n'\), \(\text{op}(A) = A\). If \(\text{trans} = 'T', 't', 'C', \text{or} 'c'\), \(\text{op}(A) = A^T\).
Input (continued)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>diag</td>
<td>specifies whether $A$ is unit triangular. If $\text{diag} = \text{'U'}$ or $\text{'u'}$, $A$ is assumed to be unit triangular; that is, diagonal elements are not read and are assumed to be unity. If $\text{diag} = \text{'N'}$ or $\text{'n'}$, $A$ is not assumed to be unit triangular.</td>
</tr>
<tr>
<td>n</td>
<td>the number of rows and columns of matrix $A$; $n$ must be at least zero.</td>
</tr>
<tr>
<td>k</td>
<td>specifies the number of superdiagonals or subdiagonals. If $\text{uplo} = \text{'U'}$ or $\text{'u'}$, $k$ specifies the number of superdiagonals. If $\text{uplo} = \text{'L'}$ or $\text{'l'}$, $k$ specifies the number of subdiagonals; $k$ must be at least zero.</td>
</tr>
<tr>
<td>$A$</td>
<td>single precision array of dimension $(\text{lda}, n)$. If $\text{uplo} = \text{'U'}$ or $\text{'u'}$, the leading $(k+1) \times n$ part of the array $A$ must contain the upper triangular band matrix, supplied column by column, with the leading diagonal of the matrix in row $k+1$ of the array, the first superdiagonal starting at position 2 in row $k$, and so on. The top left $k \times k$ triangle of the array $A$ is not referenced. If $\text{uplo} = \text{'L'}$ or $\text{'l'}$, the leading $(k+1) \times n$ part of the array $A$ must contain the lower triangular band matrix, supplied column by column, with the leading diagonal of the matrix in row 1 of the array, the first sub-diagonal starting at position 1 in row 2, and so on. The bottom right $k \times k$ triangle of the array is not referenced.</td>
</tr>
<tr>
<td>$x$</td>
<td>single precision array of length at least $1 + (n - 1) \times \text{abs}(\text{incx})$. On entry, $x$ contains the $n$-element right-hand side vector $b$. On exit, it is overwritten with the solution vector $x$.</td>
</tr>
<tr>
<td>incx</td>
<td>storage spacing between elements of $x$; incx must not be zero.</td>
</tr>
</tbody>
</table>

Output

$x$ updated to contain the solution vector $x$ that solves $\text{op}(A) \times x = b$.

Reference: 
http://www.netlib.org/blas/stbsv.f

Error status for this function can be retrieved via $\text{cublasGetError()}$.

Error Status

<table>
<thead>
<tr>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUBLAS_STATUS_NOT_INITIALIZED</td>
<td>if CUBLAS library was not initialized</td>
</tr>
<tr>
<td>CUBLAS_STATUS_INVALID_VALUE</td>
<td>if $\text{incx} = 0$, $n &lt; 0$, or $n &gt; 4070$</td>
</tr>
<tr>
<td>CUBLAS_STATUS_EXECUTION_FAILED</td>
<td>if function failed to launch on GPU</td>
</tr>
</tbody>
</table>
**Function cublasStpmv()**

```c
void
cublasStpmv (char uplo, char trans, char diag, int n,
const float *AP, float *x, int incx)
```

performs one of the matrix-vector operations

\[
x = \text{op}(A) \times x,
\]

where \( \text{op}(A) = A \) or \( \text{op}(A) = A^T \).

\( x \) is an \( n \)-element single precision vector, and \( A \) is an \( n \times n \), unit or non-unit, upper or lower, triangular matrix consisting of single precision elements.

**Input**

- `uplo` specifies whether the matrix \( A \) is an upper or lower triangular matrix.
  - If `uplo` == `'U'` or `'u'`, \( A \) is an upper triangular matrix.
  - If `uplo` == `'L'` or `'l'`, \( A \) is a lower triangular matrix.
- `trans` specifies \( \text{op}(A) \).
  - If `trans` == `'N'` or `'n'`, \( \text{op}(A) = A \).
  - If `trans` == `'T'`, `'t'`, `'C'`, or `'c'`, \( \text{op}(A) = A^T \).
- `diag` specifies whether or not matrix \( A \) is unit triangular.
  - If `diag` == `'U'` or `'u'`, \( A \) is assumed to be unit triangular.
  - If `diag` == `'N'` or `'n'`, \( A \) is not assumed to be unit triangular.
- `n` specifies the number of rows and columns of the matrix \( A \); \( n \) must be at least zero. In the current implementation \( n \) must not exceed 4070.
- `AP` single precision array with at least \( (n * (n + 1)) / 2 \) elements. If `uplo` == `'U'` or `'u'`, the array \( AP \) contains the upper triangular part of the symmetric matrix \( A \), packed sequentially, column by column; that is, if \( i \leq j \), \( A[i, j] \) is stored in \( AP[i + j * (j + 1) / 2] \). If `uplo` == `'L'` or `'l'`, array \( AP \) contains the lower triangular part of the symmetric matrix \( A \), packed sequentially, column by column; that is, if \( i \geq j \), \( A[i, j] \) is stored in \( AP[i + ((2 * n - j + 1) * j) / 2] \).
- `x` single precision array of length at least \( (1 + (n - 1) * \text{abs}(incx)) \).
  - On entry, \( x \) contains the source vector. On exit, \( x \) is overwritten with the result vector.
- `incx` specifies the storage spacing for elements of \( x \); `incx` must not be zero.

**Output**

- `x` updated according to \( x = \text{op}(A) \times x \).

**Reference:** [http://www.netlib.org/blas/stpmv.f](http://www.netlib.org/blas/stpmv.f)
Error status for this function can be retrieved via `cublasGetError()`.

### Error Status

<table>
<thead>
<tr>
<th>Error Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CUBLAS_STATUS_NOT_INITIALIZED</code></td>
<td>if CUBLAS library was not initialized</td>
</tr>
<tr>
<td><code>CUBLAS_STATUS_INVALID_VALUE</code></td>
<td>if incx == 0, n &lt; 0, or n &gt; 4070</td>
</tr>
<tr>
<td><code>CUBLAS_STATUS_EXECUTION_FAILED</code></td>
<td>if function failed to launch on GPU</td>
</tr>
</tbody>
</table>

#### Function cublasStpsv()

```c
void

void cublasStpsv (char uplo, char trans, char diag, int n,
                  const float *AP, float *X, int incx)
```

Solves one of the systems of equations

```
op(A) * x = b,
where \( \text{op}(A) = A \text{ or } A^T \),
```

\( b \) and \( x \) are \( n \)-element single precision vectors, and \( A \) is an \( n \times n \), unit or non-unit, upper or lower, triangular matrix.

No test for singularity or near-singularity is included in this function.

Such tests must be performed before calling this function.

### Input

- **uplo** specifies whether the matrix is an upper or lower triangular matrix. If `uplo == 'U'` or `'u'`, \( A \) is an upper triangular matrix. If `uplo == 'L'` or `'l'`, \( A \) is a lower triangular matrix.
- **trans** specifies \( \text{op}(A) \). If `trans == 'N'` or `'n'`, \( \text{op}(A) = A \). If `trans == 'T'`, `'t'`, `'C'`, or `'c'`, \( \text{op}(A) = A^T \).
- **diag** specifies whether \( A \) is unit triangular. If `diag == 'U'` or `'u'`, \( A \) is assumed to be unit triangular; that is, diagonal elements are not read and are assumed to be unity. If `diag == 'N'` or `'n'`, \( A \) is not assumed to be unit triangular.
- **n** specifies the number of rows and columns of the matrix \( A \); \( n \) must be at least zero. In the current implementation \( n \) must not exceed 4070.
CHAPTER 3 BLAS2 and BLAS3 Functions

Input (continued)

\[ \text{AP} \] single precision array with at least \((n \times (n + 1))/2\) elements. If \(\text{uplo} == 'U'\) or \('u'\), array \(\text{AP}\) contains the upper triangular matrix \(A\), packed sequentially, column by column; that is, if \(i <= j\), \(A[i, j]\) is stored in \(\text{AP}[i + (j \times (j + 1))/2]\). If \(\text{uplo} == 'L'\) or \('l'\), array \(\text{AP}\) contains the lower triangular matrix \(A\), packed sequentially, column by column; that is, if \(i >= j\), \(A[i, j]\) is stored in \(\text{AP}[i + ((2 \times n - j + 1) \times j)/2]\). When \(\text{diag} == 'U'\) or \('u'\), the diagonal elements of \(A\) are not referenced and are assumed to be unity.

\[ x \] single precision array of length at least \((1 + (n - 1) \times \text{abs}(\text{incx}))\). On entry, \(x\) contains the \(n\)-element right-hand side vector \(b\). On exit, it is overwritten with the solution vector \(x\).

\[ \text{incx} \] storage spacing between elements of \(x\); \(\text{incx}\) must not be zero.

Output

\[ x \] updated to contain the solution vector \(x\) that solves \(\text{op}(A) \times x = b\).

Reference: http://www.netlib.org/blas/stpsv.f

Error status for this function can be retrieved via \text{cublasGetError}().

Error Status

\begin{tabular}{ll}
\text{CUBLAS_STATUS_NOT_INITIALIZED} & if CUBLAS library was not initialized \\
\text{CUBLAS_STATUS_INVALID_VALUE} & if \(\text{incx} == 0, n < 0\), or \(n > 4070\) \\
\text{CUBLAS_STATUS_EXECUTION_FAILED} & if function failed to launch on GPU
\end{tabular}

Function cublasStrmv()

\begin{verbatim}
void 
cublasStrmv (char uplo, char trans, char diag, int n,  
    const float *A, int lda, float *x, int incx) 
\end{verbatim}

performs one of the matrix-vector operations

\[ x = \text{op}(A) \times x, \]

where \(\text{op}(A) = A \text{ or } \text{op}(A) = A^T,\)
x is an n-element single precision vector, and A is an n×n, unit or non-unit, upper or lower, triangular matrix consisting of single precision elements.

Input

<table>
<thead>
<tr>
<th>uplo</th>
<th>specifies whether the matrix A is an upper or lower triangular matrix.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>If uplo == 'U' or 'u', A is an upper triangular matrix.</td>
</tr>
<tr>
<td></td>
<td>If uplo == 'L' or 'l', A is an lower triangular matrix.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>trans</th>
<th>specifies op(A). If trans == 'N' or 'n', op(A) = A.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>If trans == 'T', 't', 'C', or 'c', op(A) = Aᵀ.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>diag</th>
<th>specifies whether or not A is a unit triangular matrix. If diag == 'U' or 'u', A is assumed to be unit triangular. If diag == 'N' or 'n', A is not assumed to be unit triangular.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>n</th>
<th>specifies the number of rows and columns of the matrix A; n must be at least zero. In the current implementation, n must not exceed 4070.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>single precision array of dimensions (lda, n). If uplo == 'U' or 'u', the leading n×n upper triangular part of the array A must contain the upper triangular matrix, and the strictly lower triangular part of A is not referenced. If uplo == 'L' or 'l', the leading n×n lower triangular part of the array A must contain the lower triangular matrix, and the strictly upper triangular part of A is not referenced. When diag == 'U' or 'u', the diagonal elements of A are not referenced either, but are assumed to be unity.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>lda</th>
<th>leading dimension of A; lda must be at least max(1, n).</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>x</th>
<th>single precision array of length at least ((1 + (n - 1) \times \text{abs(incx)})). On entry, x contains the source vector. On exit, x is overwritten with the result vector.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>incx</th>
<th>the storage spacing between elements of x; incx must not be zero.</th>
</tr>
</thead>
</table>

Output

\[
\text{x} \quad \text{updated according to } \text{x} = \text{op}(A) \times \text{x}. 
\]

Reference: http://www.netlib.org/blas/strmv.f

Error status for this function can be retrieved via cublasGetError().

Error Status

<table>
<thead>
<tr>
<th>Error Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUBLAS_STATUS_NOT_INITIALIZED</td>
<td>if CUBLAS library was not initialized</td>
</tr>
<tr>
<td>CUBLAS_STATUS_INVALID_VALUE</td>
<td>if incx == 0, n &lt; 0, or n &gt; 4070</td>
</tr>
<tr>
<td>CUBLAS_STATUS_EXECUTION_FAILED</td>
<td>if function failed to launch on GPU</td>
</tr>
</tbody>
</table>
Function **cublasStrsv()**

```c
void
cublasStrsv (char uplo, char trans, char diag, int n,
             const float *A, int lda, float *x, int incx)
```

Solves a system of equations

\[ \text{op}(A) \times x = b, \]

where \( \text{op}(A) = A \) or \( \text{op}(A) = A^T \).

\( b \) and \( x \) are \( n \)-element single precision vectors, and \( A \) is an \( n \times n \), unit or non-unit, upper or lower, triangular matrix consisting of single precision elements. Matrix \( A \) is stored in column-major format, and \( lda \) is the leading dimension of the two-dimensional array containing \( A \).

No test for singularity or near-singularity is included in this function. Such tests must be performed before calling this function.

**Input**

- **uplo** specifies whether the matrix data is stored in the upper or the lower triangular part of array \( A \). If \( \text{uplo} == 'U' \) or \( 'u' \), only the upper triangular part of \( A \) may be referenced. If \( \text{uplo} == 'L' \) or \( 'l' \), only the lower triangular part of \( A \) may be referenced.
- **trans** specifies \( \text{op}(A) \). If \( \text{trans} == 'N' \) or \( 'n' \), \( \text{op}(A) = A \). If \( \text{trans} == 'T', 't', 'C', \) or \( 'c' \), \( \text{op}(A) = A^T \).
- **diag** specifies whether or not \( A \) is a unit triangular matrix. If \( \text{diag} == 'U' \) or \( 'u' \), \( A \) is assumed to be unit triangular. If \( \text{diag} == 'N' \) or \( 'n' \), \( A \) is not assumed to be unit triangular.
- **n** specifies the number of rows and columns of the matrix \( A \); \( n \) must be at least zero. In the current implementation, \( n \) must not exceed 4070.
- **A** single precision array of dimensions \((lda,n)\). If \( \text{uplo} == 'U' \) or \( 'u' \), \( A \) contains the upper triangular part of the symmetric matrix, and the strictly lower triangular part is not referenced. If \( \text{uplo} == 'L' \) or \( 'l' \), \( A \) contains the lower triangular part of the symmetric matrix, and the strictly upper triangular part is not referenced.
- **lda** leading dimension of the two-dimensional array containing \( A \); \( lda \) must be at least \( \max(1,n) \).
- **x** single precision array of length at least \((1+(n-1)\times \text{abs}(\text{incx}))\).
  - On entry, \( x \) contains the \( n \)-element, right-hand-side vector \( b \). On exit, it is overwritten with the solution vector \( x \).
- **incx** the storage spacing between elements of \( x \); \( incx \) must not be zero.
Output

\[ \mathbf{x} \] updated to contain the solution vector \( \mathbf{x} \) that solves \( \text{op}(\mathbf{A}) \cdot \mathbf{x} = \mathbf{b} \).

Reference: [http://www.netlib.org/blas/strsv.f](http://www.netlib.org/blas/strsv.f)

Error status for this function can be retrieved via `cublasGetError()`.

Error Status

<table>
<thead>
<tr>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUBLAS_STATUS_NOT_INITIALIZED</td>
<td>If CUBLAS library was not initialized</td>
</tr>
<tr>
<td>CUBLAS_STATUS_INVALID_VALUE</td>
<td>If ( \text{incx} = 0, n &lt; 0, ) or ( n &gt; 4070 )</td>
</tr>
<tr>
<td>CUBLAS_STATUS_EXECUTION_FAILED</td>
<td>If function failed to launch on GPU</td>
</tr>
</tbody>
</table>

Single Precision Complex BLAS2 Functions

These functions have not been implemented yet.
Single Precision BLAS3 Functions

The single precision BLAS3 functions are listed below:

- “Function cublasSgemm()” on page 49
- “Function cublasSsymm()” on page 51
- “Function cublasSyrk()” on page 53
- “Function cublasSyr2k()” on page 54
- “Function cublasStrmm()” on page 56
- “Function cublasStrsm()” on page 58

Function cublasSgemm()

```c
void
cublasSgemm (char transa, char transb, int m, int n, int k, float alpha, const float *A, int lda, const float *B, int ldb, float beta, float *C, int ldc)
```

computes the product of matrix A and matrix B, multiplies the result by scalar alpha, and adds the sum to the product of matrix C and scalar beta. It performs one of the matrix-matrix operations:

\[
C = \alpha \cdot \text{op}(A) \cdot \text{op}(B) + \beta \cdot C,
\]

where \( \text{op}(X) = X \) or \( \text{op}(X) = X^T \),

and \( \alpha \) and \( \beta \) are single precision scalars. \( A, B, \) and \( C \) are matrices consisting of single precision elements, with \( \text{op}(A) \) an \( m \times k \) matrix, \( \text{op}(B) \) a \( k \times n \) matrix, and \( C \) an \( m \times n \) matrix. Matrices \( A, B, \) and \( C \) are stored in column-major format, and \( \text{lda}, \text{ldb}, \) and \( \text{ldc} \) are the leading dimensions of the two-dimensional arrays containing \( A, B, \) and \( C. \)

Input

- `transa` specifies \( \text{op}(A) \). If `transa` == 'N' or 'n', \( \text{op}(A) = A \).
  
  If `transa` == 'T', 't', 'C', or 'c', \( \text{op}(A) = A^T \).

- `transb` specifies \( \text{op}(B) \). If `transb` == 'N' or 'n', \( \text{op}(B) = B \).
  
  If `transb` == 'T', 't', 'C', or 'c', \( \text{op}(B) = B^T \).

- `m` number of rows of matrix \( \text{op}(A) \) and rows of matrix \( C; \) \( m \) must be at least zero.
Input (continued)

\[ C = \alpha A B + \beta C \]

\( n \) number of columns of matrix \( \text{op}(B) \) and number of columns of \( C \); 
\( n \) must be at least zero.

\( k \) number of columns of matrix \( \text{op}(A) \) and number of rows of \( \text{op}(B) \); 
\( k \) must be at least zero.

\( \alpha \) single precision scalar multiplier applied to \( \text{op}(A) \) * \( \text{op}(B) \).

\( A \) single precision array of dimensions \((\text{lda}, k)\) if \( \text{transa} = 'N' \) or \( 'n' \), and of dimensions \((\text{lda}, m)\) otherwise. If \( \text{transa} = 'N' \) or \( 'n' \), \( \text{lda} \) must be at least \( \max(1, m) \); otherwise, \( \text{lda} \) must be at least \( \max(1, k) \).

\( \text{lda} \) leading dimension of two-dimensional array used to store matrix \( A \).

\( B \) single precision array of dimensions \((\text{ldb}, n)\) if \( \text{transb} = 'N' \) or \( 'n' \), and of dimensions \((\text{ldb}, k)\) otherwise. If \( \text{transb} = 'N' \) or \( 'n' \), \( \text{ldb} \) must be at least \( \max(1, k) \); otherwise, \( \text{ldb} \) must be at least \( \max(1, n) \).

\( \text{ldb} \) leading dimension of two-dimensional array used to store matrix \( B \).

\( \beta \) single precision scalar multiplier applied to \( C \). If zero, \( C \) does not have to be a valid input.

\( C \) single precision array of dimensions \((\text{ldc}, n)\); \( \text{ldc} \) must be at least \( \max(1, m) \).

\( \text{ldc} \) leading dimension of two-dimensional array used to store matrix \( C \).

Output

\( C \) updated based on \( C = \alpha A B + \beta C \).

Reference: [http://www.netlib.org/blas/sgemm.f](http://www.netlib.org/blas/sgemm.f)

Error status for this function can be retrieved via `cublasGetError()`.

Error Status

- `CUBLAS_STATUS_NOT_INITIALIZED` if CUBLAS library was not initialized
- `CUBLAS_STATUS_INVALID_VALUE` if \( m < 0 \), \( n < 0 \), or \( k < 0 \)
- `CUBLAS_STATUS_EXECUTION_FAILED` if function failed to launch on GPU
CHAPTER 3 
BLAS2 and BLAS3 Functions

Function cublasSsymm()

```c
void
cublasSsymm (char side, char uplo, int m, int n,
    float alpha, const float *A, int lda,
    const float *B, int ldb, float beta,
    float *C, int ldc)
```

performs one of the matrix-matrix operations

\[
C = \alpha A \times B + \beta C \text{ or } C = \alpha B \times A + \beta C,
\]

where \( \alpha \) and \( \beta \) are single precision scalars, \( A \) is a symmetric matrix consisting of single precision elements and is stored in either lower or upper storage mode. \( B \) and \( C \) are \( m \times n \) matrices consisting of single precision elements.

Input

- **side** specifies whether the symmetric matrix \( A \) appears on the left-hand side or right-hand side of matrix \( B \).
  - If `side` == 'L' or 'l', \( C = \alpha A \times B + \beta C \).
  - If `side` == 'R' or 'r', \( C = \alpha B \times A + \beta C \).

- **uplo** specifies whether the symmetric matrix \( A \) is stored in upper or lower storage mode. If `uplo` == 'U' or 'u', only the upper triangular part of the symmetric matrix is referenced, and the elements of the strictly lower triangular part are inferred from those in the upper triangular part. If `uplo` == 'L' or 'l', only the lower triangular part of the symmetric matrix is referenced, and the elements of the strictly upper triangular part are inferred from those in the lower triangular part.

- **m** specifies the number of rows of matrix \( C \), and the number of rows of matrix \( B \). It also specifies the dimensions of symmetric matrix \( A \) when `side` == 'L' or 'l'; \( m \) must be at least zero.

- **n** specifies the number of columns of matrix \( C \), and the number of columns of matrix \( B \). It also specifies the dimensions of symmetric matrix \( A \) when `side` == 'R' or 'r'; \( n \) must be at least zero.

- **alpha** single precision scalar multiplier applied to \( A \times B \) or \( B \times A \).
Input (continued)

A  
single precision array of dimensions (lda, ka), where ka is m when side == 'L' or 'l' and is n otherwise. If side == 'L' or 'l', the leading m×m part of array A must contain the symmetric matrix, such that when uplo == 'U' or 'u', the leading m×m part stores the upper triangular part of the symmetric matrix, and the strictly lower triangular part of A is not referenced; and when uplo == 'L' or 'l', the leading m×m part stores the lower triangular part of the symmetric matrix and the strictly upper triangular part is not referenced. If side == 'R' or 'r', the leading n×n part of array A must contain the symmetric matrix, such that when uplo == 'U' or 'u', the leading n×n part stores the upper triangular part of the symmetric matrix and the strictly lower triangular part of A is not referenced; and when uplo == 'L' or 'l', the leading n×n part stores the lower triangular part of the symmetric matrix and the strictly upper triangular part is not referenced.

lda  
leading dimension of A. When side == 'L' or 'l', it must be at least max(1, m) and at least max(1, n) otherwise.

B  
single precision array of dimensions (ldb, n). On entry, the leading m×n part of the array contains the matrix B.

ldb  
leading dimension of B; ldb must be at least max(1, m).

beta  
single precision scalar multiplier applied to C. If beta is zero, C does not have to be a valid input.

C  
single precision array of dimensions (ldc, n).

ldc  
leading dimension of C; ldc must be at least max(1, m).

Output

C  
updated according to C = alpha * A * B + beta * C  or
   C = alpha * B * A + beta * C.

Reference: http://www.netlib.org/blas/ssymm.f

Error status for this function can be retrieved via cublasGetError().

Error Status

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUBLAS_STATUS_NOT_INITIALIZED</td>
<td>if CUBLAS library was not initialized</td>
</tr>
<tr>
<td>CUBLAS_STATUS_INVALID_VALUE</td>
<td>if m &lt; 0 or n &lt; 0</td>
</tr>
<tr>
<td>CUBLAS_STATUS_EXECUTION_FAILED</td>
<td>if function failed to launch on GPU</td>
</tr>
</tbody>
</table>
CHAPTER 3 BLAS2 and BLAS3 Functions

Function cublasSsyrk()

```c
void
cublasSsyrk (char uplo, char trans, int n, int k,
float alpha, const float *A, int lda,
float beta, float *C, int ldc)
```

performs one of the symmetric rank \( k \) operations

\[
C = alpha * A * A^T + beta * C \quad \text{or} \quad C = alpha * A^T * A + beta * C,
\]

where \( \alpha \) and \( \beta \) are single precision scalars. \( C \) is an \( n \times n \) symmetric matrix consisting of single precision elements and is stored in either lower or upper storage mode. \( A \) is a matrix consisting of single precision elements with dimensions of \( n \times k \) in the first case, and \( k \times n \) in the second case.

**Input**

- `uplo` specifies whether the symmetric matrix \( C \) is stored in upper or lower storage mode. If `uplo` is 'U' or 'u', only the upper triangular part of the symmetric matrix is referenced, and the elements of the strictly lower triangular part are inferred from those in the upper triangular part. If `uplo` is 'L' or 'l', only the lower triangular part of the symmetric matrix is referenced, and the elements of the strictly upper triangular part are inferred from those in the lower triangular part.

- `trans` specifies the operation to be performed. If `trans` is 'N' or 'n', \( C = alpha * A * A^T + beta * C \). If `trans` is 'T', 't', 'C', or 'c', \( C = alpha * A^T * A + beta * C \).

- `n` specifies the number of rows and the number columns of matrix \( C \). If `trans` is 'N' or 'n', \( n \) specifies the number of rows of matrix \( A \). If `trans` is 'T', 't', 'C', or 'c', \( n \) specifies the number of columns of matrix \( A \); \( n \) must be at least zero.

- `k` specifies the number of columns of matrix \( A \). If `trans` is 'N' or 'n', \( k \) specifies the number of rows of matrix \( A \); \( k \) must be at least zero.

- `alpha` is a single precision scalar multiplier applied to \( A * A^T \) or \( A^T * A \).

- `A` is a single precision array of dimensions \((\text{lda}, k)\), where \( k \) is \( k \) when `trans` is 'N' or 'n', and is \( n \) otherwise. When `trans` is 'N' or 'n', the leading \( n \times k \) part of array \( A \) contains the matrix \( A \); otherwise, the leading \( k \times n \) part of the array contains the matrix \( A \).

- `lda` is the leading dimension of \( A \). When `trans` is 'N' or 'n', \( lda \) must be at least \( \max(1, n) \). Otherwise, \( lda \) must be at least \( \max(1, k) \).
Input (continued)

beta single precision scalar multiplier applied to C.
  If beta is zero, C is not read.
C single precision array of dimensions (ldc, n). If uplo == 'U' or 'u',
  the leading n×n triangular part of the array C must contain the upper
  triangular part of the symmetric matrix C, and the strictly lower
  triangular part of C is not referenced. On exit, the upper triangular
  part of C is overwritten by the upper triangular part of the updated matrix.
  If uplo == 'L' or 'l', the leading n×n triangular part of the array C
  must contain the lower triangular part of the symmetric matrix C, and
  the strictly upper triangular part of C is not referenced. On exit, the
  lower triangular part of C is overwritten by the lower triangular part
  of the updated matrix.
ldc leading dimension of C; ldc must be at least max(1, n).

Output

C updated according to C = alpha * A * Aᵀ + beta * C or
C = alpha * Aᵀ * A + beta * C.

Reference: http://www.netlib.org/blas/ssyrk.f

Error status for this function can be retrieved via cublasGetError().

Error Status

<table>
<thead>
<tr>
<th>Error Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUBLAS_STATUS_NOT_INITIALIZED</td>
<td>if CUBLAS library was not initialized</td>
</tr>
<tr>
<td>CUBLAS_STATUS_INVALID_VALUE</td>
<td>if n &lt; 0 or k &lt; 0</td>
</tr>
<tr>
<td>CUBLAS_STATUS_EXECUTION_FAILED</td>
<td>if function failed to launch on GPU</td>
</tr>
</tbody>
</table>

Function cublasSsyr2k()

void

cublasSsyr2k (char uplo, char trans, int n, int k,
  float alpha, const float *A, int lda,
  const float *B, int ldb, float beta,
  float *C, int ldc)

performs one of the symmetric rank 2k operations

C = alpha * A * Bᵀ + alpha * B * Aᵀ + beta * C or
C = alpha * Aᵀ * B + alpha * Bᵀ * A + beta * C,

where alpha and beta are single precision scalars. C is an n×n
symmetric matrix consisting of single precision elements and is stored
in either lower or upper storage mode. A and B are matrices consisting of single precision elements with dimension of \( n \times k \) in the first case, and \( k \times n \) in the second case.

**Input**

- **uplo** specifies whether the symmetric matrix C is stored in upper or lower storage mode. If \( \text{uplo} == 'U' \) or \('u' \), only the upper triangular part of the symmetric matrix is referenced, and the elements of the strictly lower triangular part are inferred from those in the upper triangular part. If \( \text{uplo} == 'L' \) or \('l' \), only the lower triangular part of the symmetric matrix is referenced, and the elements of the strictly upper triangular part are inferred from those in the lower triangular part.

- **trans** specifies the operation to be performed. If \( \text{trans} == 'N' \) or \('n' \), \( C = \alpha \cdot A \cdot B^T + \alpha \cdot B \cdot A^T + \beta \cdot C \). If \( \text{trans} == 'T' \), \('t' \), \('C' \), or \('c' \), \( C = \alpha \cdot A^T \cdot B + \alpha \cdot B^T \cdot A + \beta \cdot C \).

- **n** specifies the number of rows and the number columns of matrix C. If \( \text{trans} == 'N' \) or \('n' \), \( n \) specifies the number of rows of matrix A. If \( \text{trans} == 'T' \), \('t' \), \('C' \), or \('c' \), \( n \) specifies the number of columns of matrix A; \( n \) must be at least zero.

- **k** specifies the number of columns of matrix C. If \( \text{trans} == 'N' \) or \('n' \), \( k \) specifies the number of columns of matrix A; \( k \) must be at least zero.

- **alpha** single precision scalar multiplier.

- **A** single precision array of dimensions \((\text{lda}, \text{ka})\), where \( \text{ka} \) is \( k \) when \( \text{trans} == 'N' \) or \('n' \), and is \( n \) otherwise. When \( \text{trans} == 'N' \) or \('n' \), the leading \( n \times k \) part of array A must contain the matrix A, otherwise the leading \( k \times n \) part of the array must contain the matrix A.

- **lda** leading dimension of A. When \( \text{trans} == 'N' \) or \('n' \), \( \text{lda} \) must be at least \( \max(1, n) \). Otherwise \( \text{lda} \) must be at least \( \max(1, k) \).

- **B** single precision array of dimensions \((\text{ldb}, \text{kb})\), where \( \text{kb} = k \) when \( \text{trans} == 'N' \) or \('n' \), and \( k = n \) otherwise. When \( \text{trans} == 'N' \) or \('n' \), the leading \( n \times k \) part of array B must contain the matrix B, otherwise the leading \( k \times n \) part of the array must contain the matrix B.

- **ldb** leading dimension of B. When \( \text{trans} == 'N' \) or \('n' \), \( \text{ldb} \) must be at least \( \max(1, n) \). Otherwise \( \text{ldb} \) must be at least \( \max(1, k) \).

- **beta** single precision scalar multiplier applied to C. If \( \beta \) is zero, C does not have to be a valid input.
CUDA

CUBLAS Library

Input (continued)

C

single precision array of dimensions (ldc, n). If uplo == 'U' or 'u',
the leading n x n triangular part of the array C must contain the upper
triangular part of the symmetric matrix C, and the strictly lower
triangular part of C is not referenced. On exit, the upper triangular part
of C is overwritten by the upper triangular part of the updated matrix.
If uplo == 'L' or 'l', the leading n x n triangular part of the array C
must contain the lower triangular part of the symmetric matrix C, and
the strictly upper triangular part of C is not referenced. On exit, the
lower triangular part of C is overwritten by the lower triangular part
of the updated matrix.

ldc

leading dimension of C; ldc must be at least max(1, n).

Output

C

updated according to

C = alpha * A * B^T + alpha * B * A^T + beta * C or
C = alpha * A^T * B + alpha * B^T * A + beta * C.

Reference:  http://www.netlib.org/blas/ssyr2k.f

Error status for this function can be retrieved via cublasGetError().

Error Status

<table>
<thead>
<tr>
<th>Error Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUBLAS_STATUS_NOT_INITIALIZED</td>
<td>if CUBLAS library was not initialized</td>
</tr>
<tr>
<td>CUBLAS_STATUS_INVALID_VALUE</td>
<td>if n &lt; 0 or k &lt; 0</td>
</tr>
<tr>
<td>CUBLAS_STATUS_EXECUTION_FAILED</td>
<td>if function failed to launch on GPU</td>
</tr>
</tbody>
</table>

Function cublasStrmm()

void

cublasStrmm (char side, char uplo, char transa,
char diag, int m, int n, float alpha,
const float *A, int lda, const float *B,
int ldb)

performs one of the matrix-matrix operations

B = alpha * op(A) * B or B = alpha * B * op(A),
where op(A) = A or op(A) = A^T,

alpha is a single precision scalar, B is an m x n matrix consisting of
single precision elements, and A is a unit or non-unit, upper or lower
triangular matrix consisting of single precision elements.
Matrices \( A \) and \( B \) are stored in column-major format, and \( \text{lda} \) and \( \text{ldb} \) are the leading dimensions of the two-dimensional arrays that contain \( A \) and \( B \), respectively.

**Input**

- **side** specifies whether \( \text{op}(A) \) multiplies \( B \) from the left or right.
  - If \( \text{side} == 'L' \) or \( 'l' \), \( B = \alpha \times \text{op}(A) \times B \).
  - If \( \text{side} == 'R' \) or \( 'r' \), \( B = \alpha \times B \times \text{op}(A) \).

- **uplo** specifies whether the matrix \( A \) is an upper or lower triangular matrix.
  - If \( \text{uplo} == 'U' \) or \( 'u' \), \( A \) is an upper triangular matrix.
  - If \( \text{uplo} == 'L' \) or \( 'l' \), \( A \) is a lower triangular matrix.

- **transa** specifies the form of \( \text{op}(A) \) to be used in the matrix multiplication.
  - If \( \text{transa} == 'N' \) or \( 'n' \), \( \text{op}(A) = A \).
  - If \( \text{transa} == 'T', 't', 'C', \) or \( 'c' \), \( \text{op}(A) = A^T \).

- **diag** specifies whether or not \( A \) is a unit triangular matrix.
  - If \( \text{diag} == 'U' \) or \( 'u' \), \( A \) is assumed to be unit triangular.
  - If \( \text{diag} == 'N' \) or \( 'n' \), \( A \) is not assumed to be unit triangular.

- **m** the number of rows of matrix \( B \); \( m \) must be at least zero.

- **n** the number of columns of matrix \( B \); \( n \) must be at least zero.

- **alpha** single precision scalar multiplier applied to \( \text{op}(A) \times B \) or \( B \times \text{op}(A) \), respectively. If \( \alpha \) is zero, no accesses are made to matrix \( A \), and no read accesses are made to matrix \( B \).

- \( A \) single precision array of dimensions \((\text{lda}, k)\). If \( \text{side} == 'L' \) or \( 'l' \), \( k = m \). If \( \text{side} == 'R' \) or \( 'r' \), \( k = n \). If \( \text{uplo} == 'U' \) or \( 'u' \), the leading \( k \times k \) upper triangular part of the array \( A \) must contain the upper triangular matrix, and the strictly lower triangular part of \( A \) is not referenced. If \( \text{uplo} == 'L' \) or \( 'l' \), the leading \( k \times k \) lower triangular part of the array \( A \) must contain the lower triangular matrix, and the strictly upper triangular part of \( A \) is not referenced. When \( \text{diag} == 'U' \) or \( 'u' \), the diagonal elements of \( A \) are not referenced and are assumed to be unity.

- **lda** leading dimension of \( A \). When \( \text{side} == 'L' \) or \( 'l' \), it must be at least \( \max(1, m) \) and at least \( \max(1, n) \) otherwise.

- \( B \) single precision array of dimensions \((\text{ldb}, n)\). On entry, the leading \( m \times n \) part of the array contains the matrix \( B \). It is overwritten with the transformed matrix on exit.

- **ldb** leading dimension of \( B \); \( \text{ldb} \) must be at least \( \max(1, m) \).

**Output**

\( B \) updated according to \( B = \alpha \times \text{op}(A) \times B \) or \( B = \alpha \times B \times \text{op}(A) \).
Reference: http://www.netlib.org/blas/strmm.f

Error status for this function can be retrieved via `cublasGetError()`.  

### Error Status

<table>
<thead>
<tr>
<th>Error Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUBLAS_STATUS_NOT_INITIALIZED</td>
<td>if CUBLAS library was not initialized</td>
</tr>
<tr>
<td>CUBLAS_STATUS_INVALID_VALUE</td>
<td>if m &lt; 0 or n &lt; 0</td>
</tr>
<tr>
<td>CUBLAS_STATUS_EXECUTION_FAILED</td>
<td>if function failed to launch on GPU</td>
</tr>
</tbody>
</table>

### Function cublasStrsm()

```c
void 
cublasStrsm (char side, char uplo, char transa,
char diag, int m, int n, float alpha,
const float *A, int lda, float *B, int ldb)
```

solves one of the matrix equations

\[
\begin{align*}
op(A) \cdot X &= \alpha \cdot B \quad \text{or} \quad X \cdot \op(A) = \alpha \cdot B, \\
\text{where} \quad \op(A) &= A \quad \text{or} \quad \op(A) = A^T,
\end{align*}
\]

\(\alpha\) is a single precision scalar, and \(X\) and \(B\) are \(m \times n\) matrices that consist of single precision elements. \(A\) is a unit or non-unit, upper or lower, triangular matrix.

The result matrix \(X\) overwrites input matrix \(B\); that is, on exit the result is stored in \(B\). Matrices \(A\) and \(B\) are stored in column-major format, and \(lda\) and \(ldb\) are the leading dimensions of the two-dimensional arrays that contain \(A\) and \(B\), respectively.

### Input

- **side**: specifies whether \(\op(A)\) appears on the left or right of \(X\):
  - side == 'L' or 'l' indicates solve \(\op(A) \cdot X = \alpha \cdot B\);
  - side == 'R' or 'r' indicates solve \(X \cdot \op(A) = \alpha \cdot B\).

- **uplo**: specifies whether the matrix \(A\) is an upper or lower triangular matrix:
  - uplo == 'U' or 'u' indicates \(A\) is an upper triangular matrix;
  - uplo == 'L' or 'l' indicates \(A\) is a lower triangular matrix.

- **transa**: specifies the form of \(\op(A)\) to be used in matrix multiplication.
  - If transa == 'N' or 'n', \(\op(A) = A\).
  - If transa == 'T', 't', 'C', or 'c', \(\op(A) = A^T\).

- **diag**: specifies whether or not \(A\) is a unit triangular matrix.
  - If diag == 'U' or 'u', \(A\) is assumed to be unit triangular.
  - If diag == 'N' or 'n', \(A\) is not assumed to be unit triangular.
### Input (continued)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>m</td>
<td>specifies the number of rows of B; m must be at least zero.</td>
</tr>
<tr>
<td>n</td>
<td>specifies the number of columns of B; n must be at least zero.</td>
</tr>
<tr>
<td>alpha</td>
<td>single precision scalar multiplier applied to B. When alpha is zero, A is not referenced and B does not have to be a valid input.</td>
</tr>
<tr>
<td>A</td>
<td>single precision array of dimensions (lda, k), where k is m when side == 'L' or 'l', and is n when side == 'R' or 'r'. If uplo == 'U' or 'u', the leading k×k upper triangular part of the array A must contain the upper triangular matrix, and the strictly lower triangular matrix of A is not referenced. When uplo == 'L' or 'l', the leading k×k lower triangular part of the array A must contain the lower triangular matrix, and the strictly upper triangular part of A is not referenced. Note that when diag == 'U' or 'u', the diagonal elements of A are not referenced and are assumed to be unity.</td>
</tr>
<tr>
<td>lda</td>
<td>leading dimension of the two-dimensional array containing A. When side == 'L' or 'l', lda must be at least max(1, m). When side == 'R' or 'r', lda must be at least max(1, n).</td>
</tr>
<tr>
<td>B</td>
<td>single precision array of dimensions (ldb, n); ldb must be at least max(1, m). The leading m×n part of the array B must contain the right-hand side matrix B. On exit B is overwritten by the solution matrix X.</td>
</tr>
<tr>
<td>ldb</td>
<td>leading dimension of the two-dimensional array containing B; ldb must be at least max(1, m).</td>
</tr>
</tbody>
</table>

### Output

B contains the solution matrix X satisfying op(A) * X = alpha * B or X * op(A) = alpha * B.

### Reference

http://www.netlib.org/blas/strsm.f

Error status for this function can be retrieved via cublasGetError().

### Error Status

- **CUBLAS_STATUS_NOT_INITIALIZED** if CUBLAS library was not initialized
- **CUBLAS_STATUS_INVALID_VALUE** if m < 0 or n < 0
- **CUBLAS_STATUS_EXECUTION_FAILED** if function failed to launch on GPU

---

### Single Precision Complex BLAS3 Functions

The only single precision complex BLAS3 function is cublasCgemm().
Function cublasCgemm()

```c
void cublasCgemm (char transa, char transb, int m, int n,
    int k, cuComplex alpha, const cuComplex *A,
    int lda, const cuComplex *B, int ldb,
    cuComplex beta, cuComplex *C, int ldc)
```

performs one of the matrix-matrix operations

\[
C = \alpha \operatorname{op}(A) \ast \operatorname{op}(B) + \beta C,
\]

where \( \operatorname{op}(X) = X, \operatorname{op}(X) = X^T, \) or \( \operatorname{op}(X) = X^\dagger; \)

and \( \alpha \) and \( \beta \) are single precision complex scalars. \( A, B, \) and \( C \) are matrices consisting of single precision complex elements, with \( \operatorname{op}(A) \) an \( m \times k \) matrix, \( \operatorname{op}(B) \) a \( k \times n \) matrix and \( C \) an \( m \times n \) matrix.

**Input**

- `transa` specifies \( \operatorname{op}(A) \). If `transa` == 'N' or 'n', \( \operatorname{op}(A) = A \).
  - If `transa` == 'T' or 't', \( \operatorname{op}(A) = A^T \).
  - If `transa` == 'C' or 'c', \( \operatorname{op}(A) = A^\dagger \).

- `transb` specifies \( \operatorname{op}(B) \). If `transb` == 'N' or 'n', \( \operatorname{op}(B) = B \).
  - If `transb` == 'T' or 't', \( \operatorname{op}(B) = B^T \).
  - If `transb` == 'C' or 'c', \( \operatorname{op}(B) = B^\dagger \).

- `m` number of rows of matrix \( \operatorname{op}(A) \) and rows of matrix \( C \); 
  - `m` must be at least zero.

- `n` number of columns of matrix \( \operatorname{op}(B) \) and number of columns of \( C \); 
  - `n` must be at least zero.

- `k` number of columns of matrix \( \operatorname{op}(A) \) and number of rows of \( \operatorname{op}(B) \); 
  - `k` must be at least zero.

- `alpha` single precision complex scalar multiplier applied to \( \operatorname{op}(A) \ast \operatorname{op}(B) \).

- `A` single precision complex array of dimension \((\text{lda}, \text{k})\) if `transa` == 'N' or 'n', and of dimension \((\text{lda}, \text{m})\) otherwise.

- `lda` leading dimension of \( A \). When `transa` == 'N' or 'n', it must be at least \( \max(1, \text{m}) \) and at least \( \max(1, \text{k}) \) otherwise.

- `B` single precision complex array of dimension \((\text{ldb}, \text{n})\) if `transb` == 'N' or 'n', and of dimension \((\text{ldb}, \text{k})\) otherwise.

- `ldb` leading dimension of \( B \). When `transb` == 'N' or 'n', it must be at least \( \max(1, \text{k}) \) and at least \( \max(1, \text{n}) \) otherwise.

- `beta` single precision complex scalar multiplier applied to \( C \). If `beta` is zero, 
  - \( C \) does not have to be a valid input.
CHAPTER 3

BLAS2 and BLAS3 Functions

Input (continued)

<table>
<thead>
<tr>
<th>C</th>
<th>single precision array of dimensions (ldc, n).</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldc</td>
<td>leading dimension of C; ldc must be at least max(1, m).</td>
</tr>
</tbody>
</table>

Output

C updated according to \( C = \alpha \cdot \text{op}(A) \cdot \text{op}(B) + \beta \cdot C \).

Reference: [http://www.netlib.org/blas/cgemm.f](http://www.netlib.org/blas/cgemm.f)

Error status for this function can be retrieved via `cublasGetError()`.  

Error Status

- `CUBLAS_STATUS_NOT_INITIALIZED` if CUBLAS library was not initialized
- `CUBLAS_STATUS_INVALID_VALUE` if \( m < 0, n < 0, \) or \( k < 0 \)
- `CUBLAS_STATUS_EXECUTION_FAILED` if function failed to launch on GPU
CUBLAS Fortran Bindings

CUBLA is implemented using the C-based CUDA toolchain and thus provides a C-style API. This makes interfacing to applications written in C or C++ trivial. In addition, there are many applications implemented in Fortran that would benefit from using CUBLAS. CUBLAS uses 1-based indexing and Fortran-style column-major storage for multidimensional data to simplify interfacing to Fortran applications. Unfortunately, Fortran-to-C calling conventions are not standardized and differ by platform and toolchain. In particular, differences may exist in the following areas:

- Symbol names (capitalization, name decoration)
- Argument passing (by value or reference)
- Passing of string arguments (length information)
- Passing of pointer arguments (size of the pointer)
- Returning compound data types (for example, the complex data type)

To provide maximum flexibility in addressing those differences, the CUBLAS Fortran interface is provided in the form of wrapper functions. These wrapper functions, written in C, are located in the file fortran.c, whose code needs to be compiled into an application for it to call the CUBLAS API functions. Providing source code allows users to make any changes necessary for a particular platform and toolchain.
The code in fortran.c has been used to demonstrate interoperability with the compilers g77 3.2.3 on 32-bit Linux and Intel Fortran 9.0 on 32-bit Microsoft Windows. Note that for g77, use of the compiler flag -fno-second-underscore is required.

Two kinds of wrapper functions are provided. The thunking wrappers allow interfacing to existing Fortran applications without any changes to the applications. During each call, the wrappers allocate GPU memory, copy source data from CPU memory space to GPU memory space, call CUBLAS, and finally copy back the results to CPU memory space and deallocate the GPU memory. As this process causes very significant call overhead, these wrappers are intended for light testing, not for production code. By default, non-thunking wrappers are used for production code. To enable the thunking wrappers, symbol CUBLAS_USE_THUNKING must be defined for the compilation of fortran.c.

The non-thunking wrappers, intended for production code, substitute device pointers for vector and matrix arguments in all BLAS functions. To use these interfaces, existing applications need to be modified slightly to allocate and deallocate data structures in GPU memory space (using CUBLAS_ALLOC and CUBLAS_FREE) and to copy data between GPU and CPU memory spaces (using CUBLAS_SET_VECTOR, CUBLAS_GET_VECTOR, CUBLAS_SET_MATRIX, and CUBLAS_GET_MATRIX). The sample wrappers provided in fortran.c map device pointers to 32-bit integers on the Fortran side.