CUDA

CUBLAS Library

PG-00000-002_V1.0
June, 2007

Published by
NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050

Table of Contents

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. Since C and C++ use row-major storage, applications cannot use the native array semantics for two-dimensional arrays. Instead, macros or inline functions should be defined to implement matrices on top of one-dimensional arrays. For Fortran code ported to C in mechanical fashion, one may chose to retain 1-based indexing to avoid the need to transform loops. In this case, the array index of a matrix element in row `i` and column `j` can be computed via the following macro:

```
#define IDX2F(i,j,ld) ((((j)-1)*(ld))+((i)-1))
```

Here, `ld` refers to the leading dimension of the matrix as allocated, which in the case of column-major storage is the number of rows. For natively written C and C++ code, one would most likely chose 0-based indexing, in which case the indexing macro becomes

```
#define IDX2C(i,j,ld) (((j)*(ld))+(i))
```

Please refer to the code examples at the end of this section, which show a tiny application implemented in Fortran on the host (Example 1. "Fortran 77 Application Executing on the Host") and show versions of the application written in C using CUBLAS for the indexing styles described above (Example 2. "Application Using C and CUBLAS: 1-based Indexing" and Example 3. "Application Using C and CUBLAS: 0-based Indexing").

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 aid in debugging that retrieves the last recorded error.

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) when building for the device, and against the DSO `cublasemu.so` (Linux) or the DLL `cublasemu.dll` (Win32) when building for device emulation.

Following these three examples, the remainder of this chapter discusses "CUBLAS Types" on page 7 and "CUBLAS Helper Functions" on page 7.

Example 1. Fortran 77 Application Executing on the Host

```
      subroutine modify (m, ldm, n, p, q, alpha, beta)
      implicit none
      integer ldm, n, p, q
      real*4 m(ldm,*), alpha, beta
      external sscal
      call sscal (n-p+1, alpha, m(p,q), ldm)
      call sscal (ldm-p+1, beta, m(p,q), 1)
      return
      end

      program matrixmod
      implicit none
      integer M, N
      parameter (M=6, N=5)
      real*4 a(M,N)
      integer i, j
      do j = 1, N
        do i = 1, M
          a(i,j) = (i-1) * M + j
        enddo
      enddo
      call modify (a, M, N, 2, 3, 16.0, 12.0)
      do j = 1, N
        do i = 1, M
          write(*,"(F7.0$)") a(i,j)
        enddo
        write (*,*) ""
      enddo
      stop
      end
```

Example 2. Application Using C and CUBLAS: 1-based Indexing

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "cublas.h"

#define IDX2F(i,j,ld) ((((j)-1)*(ld))+((i)-1))

void modify (float *m, int ldm, int n, int p, int q, float alpha,
             float beta)
{
    cublasSscal (n-p+1, alpha, &m[IDX2F(p,q,ldm)], ldm);
    cublasSscal (ldm-p+1, beta, &m[IDX2F(p,q,ldm)], 1);
}

#define M 6
#define N 5
int main (void)
{
    int i, j;
    cublasStatus stat;
    float* devPtrA;
    float* a = 0;
    a = (float *)malloc (M * N * sizeof (*a));
    if (!a) {
        printf ("host memory allocation failed");
        return EXIT_FAILURE;
    }
    for (j = 1; j <= N; j++) {
        for (i = 1; i <= M; i++) {
            a[IDX2F(i,j,M)] = (i-1) * M + j;
        }
    }
    cublasInit();
    stat = cublasAlloc (M*N, sizeof(*a), (void**)&devPtrA);
```

Example 2. Application Using C and CUBLAS: 1-based Indexing (continued)

```
    if (stat != CUBLAS_STATUS_SUCCESS) {
        printf ("device memory allocation failed");
        return EXIT_FAILURE;
    }
    cublasSetMatrix (M, N, sizeof(*a), a, M, devPtrA, M);
    modify (devPtrA, M, N, 2, 3, 16.0f, 12.0f);
    cublasGetMatrix (M, N, sizeof(*a), devPtrA, M, a, M);
    cublasFree (devPtrA);
    cublasShutdown();
    for (j = 1; j <= N; j++) {
        for (i = 1; i <= M; i++) {
            printf ("%7.0f", a[IDX2F(i,j,M)]);
        }
        printf ("\n");
    }
    return EXIT_SUCCESS;
}
```

Example 3. Application Using C and CUBLAS: 0-based Indexing

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "cublas.h"

#define IDX2C(i,j,ld) (((j)*(ld))+(i))

void modify (float *m, int ldm, int n, int p, int q, float alpha,
             float beta)
{
    cublasSscal (n-p, alpha, &m[IDX2C(p,q,ldm)], ldm);
    cublasSscal (ldm-p, beta, &m[IDX2C(p,q,ldm)], 1);
}

#define M 6
#define N 5
```

Example 3. Application Using C and CUBLAS: 0-based Indexing (continued)

```
int main (void)
{
    int i, j;
    cublasStatus stat;
    float* devPtrA;
    float* a = 0;
    a = (float *)malloc (M * N * sizeof (*a));
    if (!a) {
        printf ("host memory allocation failed");
        return EXIT_FAILURE;
    }
    for (j = 0; j < N; j++) {
        for (i = 0; i < M; i++) {
            a[IDX2C(i,j,M)] = i * M + j + 1;
        }
    }
    cublasInit();
    stat = cublasAlloc (M*N, sizeof(*a), (void**)&devPtrA);
    if (stat != CUBLAS_STATUS_SUCCESS) {
        printf ("device memory allocation failed");
        return EXIT_FAILURE;
    }
    cublasSetMatrix (M, N, sizeof(*a), a, M, devPtrA, M);
    modify (devPtrA, M, N, 1, 2, 16.0f, 12.0f);
    cublasGetMatrix (M, N, sizeof(*a), devPtrA, M, a, M);
    cublasFree (devPtrA);
    cublasShutdown();
    for (j = 0; j < N; j++) {
        for (i = 0; i < M; i++) {
            printf ("%7.0f", a[IDX2C(i,j,M)]);
        }
        printf ("\n");
    }
    return EXIT_SUCCESS;
}
```

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:

cublasStatus Values

CUBLAS_STATUS_SUCCESS	operation completed successfully
CUBLAS_STATUS_NOT_INITIALIZED	CUBLAS library not initialized
CUBLAS_STATUS_ALLOC_FAILED	resource allocation failed
CUBLAS_STATUS_INVALID_VALUE	unsupported numerical value was passed to function
CUBLAS_STATUS_MAPPING_ERROR	access to GPU memory space failed
CUBLAS_STATUS_EXECUTION_FAILED	GPU program failed to execute
CUBLAS_STATUS_INTERNAL_ERROR	an internal CUBLAS operation failed

CUBLAS Helper Functions

The following are the CUBLAS helper functions:

- "Function cublasInit()" on page 8
- "Function cublasShutdown()" on page 8
- "Function cublasGetError()" on page 8
- "Function cublasAlloc()" on page 8
- "Function cublasFree()" on page 9
- "Function cublasSetVector()" on page 9
- "Function cublasGetVector()" on page 10
- "Function cublasSetMatrix()" on page 10
- "Function cublasGetMatrix()" on page 11

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

CUBLAS_STATUS_ALLOC_FAILED	if resources could not be allocated
CUBLAS_STATUS_SUCCESS	if CUBLAS library initialized successfully

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

CUBLAS_STATUS_NOT_INITIALIZED	if CUBLAS library was not initialized
CUBLAS_STATUS_SUCCESS	CUBLAS library shut down successfully

Function cublasGetError()

```
cublasStatus
cublasGetError (void)
```

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()

```
cublasStatus
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 `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. Function **`cublasAlloc()`** is a wrapper around **`cudaMalloc()`**. Device pointers returned by **`cublasAlloc()`** can therefore be passed to any CUDA device kernels, not just CUBLAS functions.

Return Values

CUBLAS_STATUS_NOT_INITIALIZED	if CUBLAS library was not initialized
CUBLAS_STATUS_INVALID_VALUE	if `n <= 0` or `elemSize <= 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()

```
cublasStatus
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

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

CUBLAS_STATUS_NOT_INITIALIZED	if CUBLAS library was not initialized
CUBLAS_STATUS_INVALID_VALUE	if `incx`, `incy`, or `elemSize` <= `0`
CUBLAS_STATUS_MAPPING_ERROR	if error accessing GPU memory
CUBLAS_STATUS_SUCCESS	if operation completed successfully

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

CUBLAS_STATUS_NOT_INITIALIZED	if CUBLAS library was not initialized
CUBLAS_STATUS_INVALID_VALUE	if `incx`, `incy`, or `elemSize` <= `0`
CUBLAS_STATUS_MAPPING_ERROR	if error accessing GPU memory
CUBLAS_STATUS_SUCCESS	if operation completed successfully

Function cublasSetMatrix()

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

copies a tile of `rows×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

CUBLAS_STATUS_NOT_INITIALIZED	if CUBLAS library was not initialized
CUBLAS_STATUS_INVALID_VALUE	if `rows` or `cols` < `0`; or `elemSize`, `lda`, or `ldb` <= `0`
CUBLAS_STATUS_MAPPING_ERROR	if error accessing GPU memory
CUBLAS_STATUS_SUCCESS	if operation completed successfully

Function cublasGetMatrix()

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

copies a tile of `rows×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

CUBLAS_STATUS_NOT_INITIALIZED	if CUBLAS library was not initialized
CUBLAS_STATUS_INVALID_VALUE	if `rows` or `cols` < `0`; or `elemSize`, `lda`, or `ldb` <= `0`
CUBLAS_STATUS_MAPPING_ERROR	if error accessing GPU memory
CUBLAS_STATUS_SUCCESS	if operation completed successfully

CHAPTER

2

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 13
- "Single-precision Complex BLAS1 Functions" on page 25

Single-precision BLAS1 Functions

The single-precision BLAS1 functions are as follows:

Function cublasIsamax()

```
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 `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

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

Error Status

CUBLAS_STATUS_NOT_INITIALIZED	if CUBLAS library was not initialized
CUBLAS_STATUS_ALLOC_FAILED	if function could not allocate reduction buffer
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function cublasIsamin()

```
int
cublasIsamin (int n, const float *x, int incx)
```

finds the smallest index of the minimum magnitude element of single-precision vector `x`; that is, the result is the first `i`, `i = 0` to `n-1`, that minimizes `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/scilib/blass.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_ALLOC_FAILED	if function could not allocate reduction buffer
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function cublasSasum()

```
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[1 + 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 occurred)

Reference: http://www.netlib.org/blas/sasum.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_ALLOC_FAILED	if function could not allocate reduction buffer
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function cublasSaxpy()

```
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[ly + i * incy] *with* alpha * x[lx + i * incx] + y[ly + i * incy], where

`lx = 1` if `incx >= 0`, else
`lx = 1 + (1 – n) * incx;`

`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 <= 0`)

Reference: 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()

```
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[lx + i * incx]` *to* `y[ly + i * incy]`,

where

`lx = 1` if `incx >= 0`, else
`lx = 1 + (1 – n) * 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

y	contains single-precision vector x

Reference: 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()

```
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[lx + i * incx] * y[ly + i * incy],

where

lx = 1 if incx >= 0, else
lx = 1 + (1 – n) * 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

Input (continued)

`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

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

Error Status

`CUBLAS_STATUS_NOT_INITIALIZED`	if CUBLAS library was not initialized
`CUBLAS_STATUS_ALLOC_FAILED`	if function could not allocate reduction buffer
`CUBLAS_STATUS_EXECUTION_FAILED`	if function failed to execute on GPU

Function cublasSnrm2()

```
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

Reference: http://www.netlib.org/slatec/lin/snrm2.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_ALLOC_FAILED	if function could not allocate reduction buffer
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function cublasSrot()

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

multiplies a 2×2 matrix $\begin{bmatrix} sc & ss \\ -ss & sc \end{bmatrix}$ with the 2×n matrix $\begin{bmatrix} x^T \\ y^T \end{bmatrix}$.

The elements of x are in x[lx + i * incx], i = 0 to n-1, where

lx = 1 if incx >= 0, else

lx = 1 + (1 – n) * 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 <= 0)
y	rotated vector y (unchanged if n <= 0)

Reference: http://www.netlib.org/blas/srot.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 cublasSrotg()

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

constructs the Givens transformation

$$G = \begin{bmatrix} sc & ss \\ -ss & sc \end{bmatrix}, \qquad sc^2 + ss^2 = 1$$

which zeros the second entry of the 2-vector $\begin{bmatrix} sa & sb \end{bmatrix}^T$.

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

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

applies the modified Givens transformation, h, to the 2×n matrix $\begin{bmatrix} x^T \\ y^T \end{bmatrix}$

The elements of x are in x[lx + i * incx], i = 0 to n-1, where

lx = 1 if incx >= 0, else
lx = 1 + (1 – n) * incx;

y is treated similarly using ly and incy.

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

sflag = -1.0f

$$h = \begin{bmatrix} \mathtt{sh00} & \mathtt{sh01} \\ \mathtt{sh10} & \mathtt{sh11} \end{bmatrix}$$

sflag = 0.0f

$$h = \begin{bmatrix} \mathtt{1.0f} & \mathtt{sh01} \\ \mathtt{sh10} & \mathtt{1.0f} \end{bmatrix}$$

sflag = 1.0f

$$h = \begin{bmatrix} \mathtt{sh00} & \mathtt{1.0f} \\ \mathtt{-1.0f} & \mathtt{sh11} \end{bmatrix}$$

sflag = -2.0f

$$h = \begin{bmatrix} \mathtt{1.0f} & \mathtt{0.0f} \\ \mathtt{0.0f} & \mathtt{1.0f} \end{bmatrix}$$

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 2×2 rotation matrix h: sparam[1] contains sh00, sparam[2] contains sh10, sparam[3] contains sh01, and sparam[4] contains sh11.

Output

x	rotated vector x (unchanged if n <= 0)
y	rotated vector y (unchanged if n <= 0)

Reference: http://www.netlib.org/blas/srotm.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 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 $(\sqrt{sd1} * sx1, \sqrt{sd2} * sy1)^T$.

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.

Output

sd1	changed to represent the effect of the transformation.
sd2	changed to represent the effect of the transformation.
sx1	changed to represent the effect of the transformation.
sparam	5-element vector. sparam[0] is sflag described above. sparam[1] through sparam[4] contain the 2×2 rotation matrix h: sparam[1] contains sh00, sparam[2] contains sh10, sparam[3] contains sh01, and sparam[4] contains sh11.

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

This function does not set any error status.

Function cublasSscal()

```
void
cublasSscal (int n, float alpha, float *x, int incx)
```

replaces single-precision vector x with single-precision alpha * x. For i = 0 to n-1, it replaces

x[lx + i * incx] *with* alpha * x[lx + i * incx],

where

lx = 1 if incx >= 0, else
lx = 1 + (1 – n) * incx.

Input

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

Output

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

Reference: http://www.netlib.org/blas/sscal.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 cublasSswap()

```
void
cublasSswap (int n, float *x, int incx, float *y,
             int incy)
```

interchanges single-precision vector x with single-precision vector y. For i = 0 to n-1, it interchanges

x[lx + i * incx] *with* y[ly + i * incy],

where

lx = 1 if incx >= 0, else
lx = 1 + (1 – n) * incx;

ly is defined in a similar manner 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

x	input vector y (unchanged if n <= 0)
y	input vector x (unchanged if n <= 0)

Reference: http://www.netlib.org/blas/sswap.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

Single-precision Complex BLAS1 Functions

The single-precision complex BLAS1 functions are as follows:

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]` *with* `alpha * x[lx + i * incx] + y[ly + i * incy]`,

where

`lx = 0` if `incx >= 0`, else

`lx = 1 + (1 – n) * incx;`

ly is defined in a similar way using incy.

Input

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

Output

y	single-precision complex result (unchanged if n <= 0)

Reference: http://www.netlib.org/blas/caxpy.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 cublasCcopy()

```
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 * incx] *to* y[ly + i * incy],

where

lx = 1 if incx >= 0, else
lx = 1 + (1 – n) * incx;

ly is defined in a similar way using incy.

Input

n	number of elements in input vectors
x	single-precision complex vector with n elements

Input (continued)

incx	storage spacing between elements of x
y	single-precision complex vector with n elements
incy	storage spacing between elements of y

Output

y	contains single-precision complex vector x

Reference: http://www.netlib.org/blas/ccopy.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 cublasCdotc()

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

computes the dot product of two single-precision complex vectors, the first of which is conjugated. It returns the dot product of the single-precision complex vectors x and y if successful, and complex zero otherwise. For i = 0 to n-1, it sums the products

x[lx + i * incx] * y[ly + i * incy],

where

lx = 1 if incx >= 0, else
lx = 1 + (1 – n) * incx;

ly is defined in a similar way using incy.

Input

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

Output

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

Reference: http://www.netlib.org/blas/cdotc.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_ALLOC_FAILED	if function could not allocate reduction buffer
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function cublasCdotu()

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

computes the dot product of two single-precision complex vectors. It returns the dot product of the single-precision complex vectors x and y if successful, and complex zero otherwise. For i = 0 to n-1, it sums the products

x[lx + i * incx] * y[ly + i * incy],

where

lx = 1 if incx >= 0, else

lx = 1 + (1 – n) * incx;

ly is defined in a similar way using incy.

Input

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

Output

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

Reference: http://www.netlib.org/blas/cdotu.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_ALLOC_FAILED	if function could not allocate reduction buffer
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function cublasCscal()

```
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]` *with* `alpha * x[lx + i * incx]`,

where

`lx = 1` if `incx >= 0`, else

`lx = 1 + (1 – n) * 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

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 cublasCsscal()

```
void
cublasCsscal (int n, float 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] *with* alpha * x[lx + i * incx],

where

lx = 1 if incx >= 0, else
lx = 1 + (1 – n) * 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 <= 0 or incx <= 0)

Reference: 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 * incx] *with* y[ly + i * incy], where

`lx = 1` if `incx >= 0`, else
`lx = 1 + (1 – n) * incx;`

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

Input

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

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 cublasIcamax()

```
int
cublasIcamax (int n, const cuComplex *x, int incx)
```

finds the smallest index of the maximum magnitude element of single-precision complex vector `x`; that is, the result is the first `i`, `i = 0` to `n-1`, that maximizes `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 complex 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/icamax.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_ALLOC_FAILED`	if function could not allocate reduction buffer
`CUBLAS_STATUS_EXECUTION_FAILED`	if function failed to launch on GPU

Function cublasIcamin()

```
int
cublasIcamin (int n, const cuComplex *x, int incx)
```

finds the smallest index of the minimum magnitude element of single-precision complex vector `x`; that is, the result is the first `i`, `i = 0` to `n-1`, that minimizes `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 complex 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: Analogous to http://www.netlib.org/blas/icamax.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_ALLOC_FAILED`	if function could not allocate reduction buffer
`CUBLAS_STATUS_EXECUTION_FAILED`	if function failed to launch on GPU

Function cublasScasum()

```
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

$$\mathtt{abs(real(x[lx+i*incx]))+abs(imag(x[lx+i*incx]))},$$

where

`lx = 1` if `incx <= 0`, else
`lx = 1 + (1 – n) * incx`.

Input

`n`	number of elements in input vector
`x`	single-precision complex vector with `n` elements
`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 <= 0`, `incx <= 0`, or if an error occurred)

Reference: 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_ALLOC_FAILED`	if function could not allocate reduction buffer
`CUBLAS_STATUS_EXECUTION_FAILED`	if function failed to launch on GPU

Function cublasScnrm2()

```
float
cublasScnrm2 (int n, const cuComplex *x, int incx)
```

computes the Euclidean norm of single-precision complex n-vector x. This implementation uses simple scaling to avoid intermediate underflow and overflow.

Input

n	number of elements in input vector
x	single-precision complex 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/scnrm2.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_ALLOC_FAILED	if function could not allocate reduction buffer
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 36
- "Single-precision Complex BLAS2 Functions" on page 57 (*Not yet implemented*)
- "Single-precision BLAS3 Functions" on page 58
- "Single-precision Complex BLAS3 Functions" on page 69

Single-precision BLAS2 Functions

The single-precision BLAS2 functions are as follows:

Function cublasSgbmv()

```
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

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

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×n band matrix consisting of single-precision elements with kl subdiagonals and ku superdiagonals.

Input

trans	specifies op(A). If trans == 'N' or 'n', op(A) = A. If trans == 'T', 't', 'C', or 'c', 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 op(A).
A	single-precision array of dimensions (lda, n). The leading (kl + ku + 1) × 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×ku triangle) are not referenced.
lda	leading dimension of A; lda must be at least kl + ku + 1.
x	single-precision array of length at least (1 + (n – 1) * abs(incx)) when trans == 'N' or 'n', and at least (1 + (m – 1) * abs(incx)) otherwise.
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 + (m – 1) * abs(incy)) when trans == 'N' or 'n' and at least (1 + (n – 1) * abs(incy)) otherwise. 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 * op(A) * x + beta * y.

Reference: http://www.netlib.org/blas/sgbmv.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, kl < 0, ku < 0, incx == 0, or incy == 0
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function cublasSgemv()

```
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)
```

performs one of the matrix-vector operations

y = alpha * op(A) * x + beta * y,

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

alpha and beta are single-precision scalars, and x and y are single-precision vectors. A is an m×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 in which A is stored.

Input

trans	specifies op(A). If trans == 'N' or 'n', op(A) = A. If trans == 'T', 't', 'C', or 'c', 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 op(A).
A	single-precision array of dimensions (lda, n) if trans == 'N' or 'n', of dimensions (lda, m) otherwise; lda must be at least max(1, m) if trans == 'N' or 'n' and at least max(1, n) otherwise.
lda	leading dimension of two-dimensional array used to store matrix A.
x	single-precision array of length at least (1 + (n – 1) * abs(incx)) if trans == 'N' or 'n', else at least (1 + (m – 1) * abs(incx)).
incx	specifies the storage spacing for 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.

Input (continued)

y	single-precision array of length at least (1 + (m – 1) * abs(incy)) if trans == 'N' or 'n', else at least (1 + (n – 1) * abs(incy)).
incy	the storage spacing between elements of y; incy must not be zero.

Output

y	updated according to y = alpha * op(A) * x + beta * y.

Reference: http://www.netlib.org/blas/sgemv.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, incx == 0, or incy == 0
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function cublasSger()

```
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×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) * abs(incx)).
incx	the storage spacing between elements of x; incx must not be zero.
y	single-precision array of length at least (1 + (n – 1) * abs(incy)).

Input (continued)

`incy`	the storage spacing between elements of `y`; `incy` must not be zero.
`A`	single-precision array of dimensions `(lda, n)`.
`lda`	leading dimension of two-dimensional array used to store matrix `A`.

Output

`A`	updated according to $A = alpha * x * y^T + A$.

Reference: http://www.netlib.org/blas/sger.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`, `incx == 0`, or `incy == 0`
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function cublasSsbmv()

```
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)
```

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 an `n×n` symmetric band matrix consisting of single-precision elements, with `k` superdiagonals and the same number of subdiagonals.

Input

`uplo`	specifies whether the upper or lower triangular part of the symmetric band matrix `A` is being supplied. If `uplo == 'U'` or `'u'`, the upper triangular part is being supplied. If `uplo == 'L'` or `'l'`, the lower triangular part is being supplied.
`n`	specifies the number of rows and the number of columns of the symmetric matrix `A`; `n` must be at least zero.

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) * 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/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()

```
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×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 array 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 A * x.
AP	single-precision array with at least $(n * (n+1))/2$ 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 $AP[i+((2*n-j+1)*j)/2]$.
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/sspmv.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, incx == 0, or incy == 0
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function cublasSspr()

```
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×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 array 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 $(1 + (n - 1) * abs(incx))$.
incx	storage spacing between elements of x; incx must not be zero.
AP	single-precision array with at least $(n * (n + 1))/2$ 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

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 incx == 0
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function cublasSspr2()

```
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

$$A = alpha * x * y^T + alpha * y * x^T + A,$$

where `alpha` is a single-precision scalar, and `x` and `y` are `n`-element single-precision vectors. `A` is a symmetric `n`×`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 array `A`. If `uplo` == `'U'` or `'u'`, only the upper triangular part of `A` may be referenced and the lower triangular part of `A` is inferred. If `uplo` == `'L'` or `'l'`, only the lower triangular part of `A` may be referenced and the upper triangular part of `A` is inferred.
`n`	the number of rows and columns of matrix `A`; `n` must be at least zero.
`alpha`	single-precision scalar multiplier applied to $x * y^T + alpha * y * x^T$.
`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.
`y`	single-precision array of length at least $(1 + (n - 1) * abs(incy))$.
`incy`	storage spacing between elements of `y`; `incy` must not be zero.
`AP`	single-precision array with at least $(n * (n + 1))/2$ 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 $A = alpha * x * y^T + alpha * y * x^T + A$.

Reference: 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, incx == 0, or incy == 0
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function cublasSsymv()

```
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

$$y = alpha * A * x + beta * y,$$

where alpha and beta are single-precision scalars, and x and y are n-element single-precision vectors. A is a symmetric n×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 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 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 * 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

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`, `incx` == `0`, or `incy` == `0`
`CUBLAS_STATUS_EXECUTION_FAILED`	if function failed to launch on GPU

Function cublasSsyr()

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

performs the symmetric rank 1 operation

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

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 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 uplo == 'U' or 'u', only the upper triangular part of A is referenced. If uplo == 'L' or 'l', only the lower triangular part of A is referenced.
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) * abs(incx)).
incx	the storage spacing between elements of x; incx must not be zero.
A	single-precision array of dimensions (lda, n). If uplo == 'U' or 'u', A contains the upper triangular part of the symmetric matrix, and the strictly lower triangular part is not referenced. If 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).

Output

A	updated according to A = alpha * x * x^T + A.

Reference: 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 incx == 0
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function cublasSsyr2()

```
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 * x * y^T + alpha * y * x^T + A, where alpha is a single-precision scalar, x and y are n-element single-precision vectors, and A is an n×n symmetric matrix consisting of single-precision elements.

Input

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

Output

A	updated according to $A = alpha * x * y^T + alpha * y * x^T + A$.

Reference: 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, incx == 0, or incy == 0
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function cublasStbmv()

```
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) * x,

where op(A) = A *or* 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 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)×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×k triangle of the array A is not referenced. If uplo == 'L' or 'l', the leading (k+1)×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×k triangle of the array is not referenced.
lda	is the leading dimension of A; lda must be at least k+1.

Input (continued)

`x`	single-precision array of length at least `(1 + (n – 1) * 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 = op(A) * x`.

Reference: http://www.netlib.org/blas/stbmv.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`, `n > 4070`, `k < 0`, or `incx == 0`
`CUBLAS_STATUS_EXECUTION_FAILED`	if function failed to launch on GPU

Function cublasStbsv()

```
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

`op(A) * x = b,`

where `op(A) = A` *or* `op(A) =` A^T,

`b` and `x` are `n`-element vectors, and `A` is an `n×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 `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.

Input (continued)

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	the number of rows and columns of matrix A; n must be at least zero.
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 be at least zero.
A	single-precision array of dimension (lda, n). If uplo == 'U' or 'u', the leading (k+1)×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×k triangle of the array A is not referenced. If uplo == 'L' or 'l', the leading (k+1)×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×k triangle of the array is not referenced.
x	single-precision array of length at least (1 + (n – 1) * abs(incx)). On entry, x contains the n-element right-hand side vector b. On exit, it is overwritten with the solution vector x.
incx	storage spacing between elements of x; incx must not be zero.

Output

x	updated to contain the solution vector x that solves op(A) * x = b.

Reference: http://www.netlib.org/blas/stbsv.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 incx == 0, n < 0, or n > 4070
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function cublasStpmv()

```
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 = op(A) * x,

where op(A) = A *or* 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

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 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.
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 <= 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 >= 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) * 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 = op(A) * x.

Reference: http://www.netlib.org/blas/stpmv.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 incx == 0, n < 0, or n > 4070
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function cublasStpsv()

```
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 op(A) = A *or* op(A) = A^T,

b and x are n-element single-precision vectors, and A is an n×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 op(A). If trans == 'N' or 'n', op(A) = A. If trans == 'T', 't', 'C', or 'c', 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.

Input (continued)

AP	single-precision array with at least `(n * (n + 1))/2` elements. If `uplo == 'U'` or `'u'`, array `AP` contains the upper triangular 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'`, array `AP` contains the lower triangular 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]`. When `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) * abs(incx))`. On entry, `x` contains the `n`-element right-hand side vector `b`. On exit, it is overwritten with the solution vector `x`.
incx	storage spacing between elements of `x`; `incx` must not be zero.

Output

x	updated to contain the solution vector `x` that solves `op(A) * x = b`.

Reference: http://www.netlib.org/blas/stpsv.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 `incx == 0`, `n < 0`, or `n > 4070`
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function cublasStrmv()

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

performs one of the matrix-vector operations

`x = op(A) * x`,

where `op(A) = A` *or* `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

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 an lower triangular 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 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.
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 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.
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)). On entry, x contains the source vector. On exit, x is overwritten with the result vector.
incx	the storage spacing between elements of x; incx must not be zero.

Output

x	updated according to x = op(A) * x.

Reference: http://www.netlib.org/blas/strmv.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 incx == 0, n < 0, or n > 4070
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function cublasStrsv()

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

solves a system of equations

op(A) * x = b,

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

b and x are n-element single-precision vectors, and A is an n×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 uplo == 'U' or 'u', only the upper triangular part of A may be referenced. If uplo == 'L' or 'l', only the lower triangular part of A may be referenced.
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 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.
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 uplo == 'U' or 'u', A contains the upper triangular part of the symmetric matrix, and the strictly lower triangular part is not referenced. If 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) * abs(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

x	updated to contain the solution vector x that solves op(A) * x = b.

Reference: http://www.netlib.org/blas/strsv.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 incx == 0, n < 0, or n > 4070
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

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()

```
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 * op(A) * op(B) + beta * C,

where op(X) = X *or* op(X) = X^T,

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

Input

transa	specifies op(A). If transa == 'N' or 'n', op(A) = A. If transa == 'T', 't', 'C', or 'c', op(A) = A^T.
transb	specifies op(B). If transb == 'N' or 'n', op(B) = B. If transb == 'T', 't', 'C', or 'c', op(B) = B^T.
m	number of rows of matrix op(A) and rows of matrix C; m must be at least zero.

Input (continued)

n	number of columns of matrix op(B) and number of columns of C; n must be at least zero.
k	number of columns of matrix op(A) and number of rows of op(B); k must be at least zero.
alpha	single-precision scalar multiplier applied to op(A) * op(B).
A	single-precision array of dimensions (lda, k) if transa == 'N' or 'n', and of dimensions (lda, m) otherwise. If transa == 'N' or 'n', lda must be at least max(1, m); otherwise, lda must be at least max(1, k).
lda	leading dimension of two-dimensional array used to store matrix A.
B	single-precision array of dimensions (ldb, n) if transb == 'N' or 'n', and of dimensions (ldb, k) otherwise. If transb == 'N' or 'n', ldb must be at least max(1, k); otherwise, ldb must be at least max(1, n).
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 (ldc, n); ldc must be at least max (1, m).
ldc	leading dimension of two-dimensional array used to store matrix C.

Output

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

Reference: 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

Function cublasSsymm()

```
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 * B + beta * C *or* C = alpha * B * 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×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 * B + beta * C. If side == 'R' or 'r', C = alpha * B * 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 * B or B * 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

`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

Function cublasSsyrk()

```
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

$$C = alpha * A * A^T + beta * C \textit{ or } C = alpha * A^T * A + beta * C,$$

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 is a matrix consisting of single-precision elements with dimensions of n×k in the first case, and k×n in the second case.

Input

uplo	specifies whether the symmetric matrix C 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.
trans	specifies the operation to be performed. If trans == 'N' or 'n', $C = alpha * A * A^T + beta * C$. If trans == '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 == 'N' or 'n', n specifies the number of rows of matrix A. If trans == 'T', 't', 'C', or 'c', n specifies the number of columns of matrix A; n must be at least zero.
k	If trans == 'N' or 'n', k specifies the number of columns of matrix A. If trans == 'T', 't', 'C', or 'c', k specifies the number of rows of matrix A; k must be at least zero.
alpha	single-precision scalar multiplier applied to $A * A^T$ or $A^T * A$.
A	single-precision array of dimensions (lda, ka), where ka is k when trans == 'N' or 'n', and is n otherwise. When trans == 'N' or 'n', the leading n×k part of array A contains the matrix A; otherwise, the leading k×n part of the array contains the matrix A.
lda	leading dimension of A. When trans == '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 $C = alpha * A * A^T + beta * C$ *or* $C = alpha * A^T * A + beta * C$.

Reference: http://www.netlib.org/blas/ssyrk.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 `k < 0`
`CUBLAS_STATUS_EXECUTION_FAILED`	if function failed to launch on GPU

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

$$C = alpha * A * B^T + alpha * B * A^T + beta * C \text{ or}$$

$$C = alpha * A^T * B + alpha * B^T * A + beta * C,$$

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×k in the first case, and k×n in the second case.

Input

uplo	specifies whether the symmetric matrix C 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.
trans	specifies the operation to be performed. If trans == 'N' or 'n', $C = alpha * A * B^T + alpha * B * A^T + beta * C$. If trans == 'T', 't', 'C', or 'c', $C = alpha * A^T * B + alpha * B^T * A + beta * C$.
n	specifies the number of rows and the number columns of matrix C. If trans == 'N' or 'n'', n specifies the number of rows of matrix A. If trans == 'T', 't', 'C', or 'c', n specifies the number of columns of matrix A; n must be at least zero.
k	If trans == 'N' or 'n', k specifies the number of columns of matrix A. If trans == 'T', 't', 'C', or 'c', k specifies the number of rows of matrix A; k must be at least zero.
alpha	single-precision scalar multiplier.
A	single-precision array of dimensions (lda, ka), where ka is k when trans == 'N' or 'n', and is n otherwise. When trans == 'N' or 'n', the leading n×k part of array A must contain the matrix A, otherwise the leading k×n part of the array must contain the matrix A.
lda	leading dimension of A. When trans == 'N' or 'n', lda must be at least max(1, n). Otherwise lda must be at least max(1, k).
B	single-precision array of dimensions (lda, kb), where kb = k when trans == 'N' or 'n', and k = n otherwise. When trans == 'N' or 'n', the leading n×k part of array B must contain the matrix B, otherwise the leading k×n part of the array must contain the matrix B.
ldb	leading dimension of B. When trans == 'N' or 'n', ldb must be at least max(1, n). Otherwise 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.

Input (continued)

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; idc 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

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

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×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 lda and ldb are the leading dimensions of the two-dimensional arrays that contain A and B, respectively.

Input

side	specifies whether op(A) multiplies B from the left or right. If side == 'L' or 'l', B = alpha * op(A) * B. If side == 'R' or 'r', B = alpha * B * op(A).
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.
transa	specifies the form of op(A) to be used in the 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.
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 op(A) * B or B * 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 (lda, k). If side == 'L' or 'l', k = m. If side == 'R' or 'r', k = n. 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 part of A is not referenced. If 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. When diag == 'U' or 'u', the diagonal elements of A are not referenced and are assumed to be unity.
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. It is overwritten with the transformed matrix on exit.
ldb	leading dimension of B; ldb must be at least max(1, m).

Output

B	updated according to B = alpha * op(A) * B *or* B = alpha * B * op(A).

Reference: http://www.netlib.org/blas/strmm.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

Function cublasStrsm()

```
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

op(A) * X = alpha * B *or* X * op(A) = alpha * B,

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

alpha is a single-precision scalar, and X and B are m×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) * X = alpha * B; side == 'R' or 'r' indicates solve X * op(A) = alpha * 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)

`m`	specifies the number of rows of `B`; `m` must be at least zero.
`n`	specifies the number of columns of `B`; `n` must be at least zero.
`alpha`	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.
`A`	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.
`lda`	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)`.
`B`	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`.
`ldb`	leading dimension of the two-dimensional array containing `B`; `ldb` must be at least `max(1, m)`.

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()

```
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 * op(A) * op(B) + beta * C,

where op(X) = X, op(X) = X^T, *or* op(X) = X^H;

and alpha and beta are single-precision complex scalars. A, B, and C are matrices consisting of single-precision complex elements, with op(A) an m×k matrix, op(B) a k×n matrix and C an m×n matrix.

Input

transa	specifies op(A). If transa == 'N' or 'n', op(A) = A. If transa == 'T' or 't', op(A) = A^T. If transa == 'C' or 'c', op(A) = A^H.
transb	specifies op(B). If transb == 'N' or 'n', op(B) = B. If transb == 'T' or 't', op(B) = B^T. If transb == 'C' or 'c', op(B) = B^H.
m	number of rows of matrix op(A) and rows of matrix C; m must be at least zero.
n	number of columns of matrix op(B) and number of columns of C; n must be at least zero.
k	number of columns of matrix op(A) and number of rows of op(B); k must be at least zero.
alpha	single-precision complex scalar multiplier applied to op(A) * op(B).
A	single-precision complex array of dimension (lda, k) if transa == 'N' or 'n', and of dimension (lda, m) otherwise.
lda	leading dimension of A. When transa == 'N' or 'n', it must be at least max(1, m) and at least max(1, k) otherwise.
B	single-precision complex array of dimension (ldb, n) if transb == 'N' or 'n', and of dimension (ldb, k) otherwise.

Input (continued)

`ldb`	leading dimension of `B`. When `transb` == `'N'` or `'n'`, it must be at least `max(1, k)` and at least `max(1, n)` otherwise.
`beta`	single-precision complex 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`; `idc` must be at least `max(1, m)`.

Output

`C`	updated according to `C = alpha * op(A) * op(B) + beta * C`.

Reference: 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

APPENDIX

A

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 floating-point or compound data types (for example, single-precision or complex data types)

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, g77 3.4.5 on 64-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 to use these wrappers as provided. Also, the use of the default calling conventions with regard to argument and return value passing is expected. Using the flag `-fno-f2c` changes the default calling convention with respect to these two items.

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, regardless of whether the host platform is a 32-bit or 64-bit platform.

One approach to deal with index arithmetic on device pointers in Fortran code is to use C-style macros, and use the C preprocessor to expand these, as shown in the example below. On Linux, one way of pre-processing is to invoke `'g77 -E -x f77-cpp-input'`. On Windows platforms with Microsoft Visual C/C++, using `'cl -EP'` achieves similar results.

When traditional fixed-form Fortran 77 code is ported to CUBLAS, line length often increases when the BLAS calls are exchanged for CUBLAS calls. Longer function names and possible macro expansion are contributing factors. Inadvertently exceeding the maximum line length can lead to run-time errors that are difficult to find, so care should be taken not to exceed the 72-column limit if fixed form is retained.

The following examples show a small application implemented in Fortran 77 on the host, and show the same application using the non-thunking wrappers after it has been ported to use CUBLAS.

Example A.1. Fortran 77 Application Executing on the Host

```
      subroutine modify (m, ldm, n, p, q, alpha, beta)
      implicit none
      integer ldm, n, p, q
      real*4 m(ldm,*), alpha, beta
      external sscal
      call sscal (n-p+1, alpha, m(p,q), ldm)
      call sscal (ldm-p+1, beta, m(p,q), 1)
      return
      end

      program matrixmod
      implicit none
      integer M, N
      parameter (M=6, N=5)
      real*4 a(M,N)
      integer i, j
      do j = 1, N
        do i = 1, M
          a(i,j) = (i-1) * M + j
        enddo
      enddo
      call modify (a, M, N, 2, 3, 16.0, 12.0)
      do j = 1, N
        do i = 1, M
          write(*,"(F7.0$)") a(i,j)
        enddo
        write (*,*) ""
```

Example A.1. Fortran 77 Application Executing on the Host (continued)

```
      enddo
      stop
      end
```

Example A.2. Same Application Using Non-thunking CUBLAS Calls

```
#define IDX2F(i,j,ld) ((((j)-1)*(ld))+((i)-1))

      subroutine modify (devPtrM, ldm, n, p, q, alpha, beta)
      implicit none
      integer sizeof_real
      parameter (sizeof_real=4)
      integer ldm, n, p, q, devPtrM
      real*4 alpha, beta
      call cublas_sscal (n-p+1, alpha,
     1                   devPtrM+IDX2F(p,q,ldm)*sizeof_real,
     2                   ldm)
      call cublas_sscal (ldm-p+1, beta,
     1                   devPtrM+IDX2F(p,q,ldm)*sizeof_real,
     2                   1)
      return
      end

      program matrixmod
      implicit none
      integer M, N, sizeof_real, devPtrA
      parameter (M=6, N=5, sizeof_real=4)
      real*4 a(M,N)
      integer i, j, stat
      external cublas_init, cublas_set_matrix, cublas_get_matrix
      external cublas_shutdown, cublas_alloc
      integer cublas_alloc
      do j = 1, N
        do i = 1, M
          a(i,j) = (i-1) * M + j
        enddo
      enddo
```

Example A.2. Same Application Using Non-thunking CUBLAS Calls (continued)

```
      call cublas_init
      stat = cublas_alloc(M*N, sizeof_real, devPtrA)
      if (stat .NE. 0) then
        write(*,*) "device memory allocation failed"
        stop
      endif
      call cublas_set_matrix (M, N, sizeof_real, a, M, devPtrA, M)
      call modify (devPtrA, M, N, 2, 3, 16.0, 12.0)
      call cublas_get_matrix (M, N, sizeof_real, devPtrA, M, a, M)
      call cublas_free (devPtrA)
      call cublas_shutdown
      do j = 1, N
        do i = 1, M
          write(*,"(F7.0$)") a(i,j)
        enddo
        write (*,*) ""
      enddo
      stop
      end
```