



CUDA **CUBLAS Library**

PG-00000-002_V1.1
September, 2007

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

Notice

ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE.

Information furnished is believed to be accurate and reliable. However, NVIDIA Corporation assumes no responsibility for the consequences of use of such information or for any infringement of patents or other rights of third parties that may result from its use. No license is granted by implication or otherwise under any patent or patent rights of NVIDIA Corporation. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all information previously supplied. NVIDIA Corporation products are not authorized for use as critical components in life support devices or systems without express written approval of NVIDIA Corporation.

Trademarks

NVIDIA, CUDA, and the NVIDIA logo are trademarks or registered trademarks of NVIDIA Corporation in the United States and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

© 2005–2007 by NVIDIA Corporation. All rights reserved.

Table of Contents

The CUBLAS Library	1
CUBLAS Types	7
Type cublasStatus	7
CUBLAS Helper Functions	7
Function cublasInit()	8
Function cublasShutdown()	8
Function cublasGetError()	8
Function cublasAlloc()	8
Function cublasFree()	9
Function cublasSetVector()	9
Function cublasGetVector()	10
Function cublasSetMatrix()	10
Function cublasGetMatrix()	11
BLAS1 Functions	12
Single-precision BLAS1 Functions	13
Function cublasIsamax()	13
Function cublasIsamin()	14
Function cublasSasum()	15
Function cublasSaxpy()	15
Function cublasScopy()	16
Function cublasSdot()	17
Function cublasSnrm2()	18
Function cublasSrot()	19
Function cublasSrotg()	20
Function cublasSrotm()	21
Function cublasSrotmg()	22
Function cublasSscal()	23
Function cublasSswap()	24
Single-precision Complex BLAS1 Functions	25
Function cublasCaxpy()	25
Function cublasCcopy()	26
Function cublasCdotc()	27
Function cublasCdotu()	28
Function cublasCrot()	29

Function cublasCrotg()	30
Function cublasCscal()	31
Function cublasCsrot()	31
Function cublasCsscal()	32
Function cublasCswap()	33
Function cublasIcamax()	34
Function cublasIcamin()	35
Function cublasScasum()	35
Function cublasScnrm2()	36
BLAS2 and BLAS3 Functions	38
Single-precision BLAS2 Functions	39
Function cublasSgbmv()	39
Function cublasSgemv()	41
Function cublasSger()	42
Function cublasSsbmv()	43
Function cublasSspmv()	45
Function cublasSspr()	46
Function cublasSspr2()	47
Function cublasSsymv()	48
Function cublasSsyr()	49
Function cublasSsyr2()	50
Function cublasStbmv()	52
Function cublasStbsv()	53
Function cublasStpmv()	55
Function cublasStpsv()	56
Function cublasStrmv()	57
Function cublasStrsv()	59
Single-precision Complex BLAS2 Functions.	60
Single-precision BLAS3 Functions	61
Function cublasSgemm()	61
Function cublasSsymm()	63
Function cublasSsyrk()	65
Function cublasSsyr2k()	66
Function cublasStrmm()	68
Function cublasStrsm()	70
Single-precision Complex BLAS3 Functions.	72
Function cublasCgemm()	72
CUBLAS Fortran Bindings	75

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, &[IDX2F(p,q,ldm)], ldm);
    cublasSscal (ldm-p+1, beta, &[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

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

<code>CUBLAS_STATUS_NOT_INITIALIZED</code>	if CUBLAS library was not initialized
<code>CUBLAS_STATUS_INVALID_VALUE</code>	if $n \leq 0$ or <code>elemSize</code> ≤ 0
<code>CUBLAS_STATUS_ALLOC_FAILED</code>	if the object could not be allocated due to lack of resources.
<code>CUBLAS_STATUS_SUCCESS</code>	if storage was successfully allocated

Function `cublasFree()`

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

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

Return Values

<code>CUBLAS_STATUS_NOT_INITIALIZED</code>	if CUBLAS library was not initialized
<code>CUBLAS_STATUS_INTERNAL_ERROR</code>	if the object could not be deallocated
<code>CUBLAS_STATUS_SUCCESS</code>	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

<code>CUBLAS_STATUS_NOT_INITIALIZED</code>	if CUBLAS library was not initialized
<code>CUBLAS_STATUS_INVALID_VALUE</code>	if <code>incx</code> , <code>incy</code> , or <code>elemSize</code> \leq 0
<code>CUBLAS_STATUS_MAPPING_ERROR</code>	if error accessing GPU memory
<code>CUBLAS_STATUS_SUCCESS</code>	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

<code>CUBLAS_STATUS_NOT_INITIALIZED</code>	if CUBLAS library was not initialized
<code>CUBLAS_STATUS_INVALID_VALUE</code>	if <code>incx</code> , <code>incy</code> , or <code>elemSize</code> \leq 0
<code>CUBLAS_STATUS_MAPPING_ERROR</code>	if error accessing GPU memory
<code>CUBLAS_STATUS_SUCCESS</code>	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

<code>CUBLAS_STATUS_NOT_INITIALIZED</code>	if CUBLAS library was not initialized
<code>CUBLAS_STATUS_INVALID_VALUE</code>	if <code>rows</code> or <code>cols</code> < 0; or <code>elemSize</code> , <code>lda</code> , or <code>ldb</code> <= 0
<code>CUBLAS_STATUS_MAPPING_ERROR</code>	if error accessing GPU memory
<code>CUBLAS_STATUS_SUCCESS</code>	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

<code>CUBLAS_STATUS_NOT_INITIALIZED</code>	if CUBLAS library was not initialized
<code>CUBLAS_STATUS_INVALID_VALUE</code>	if <code>rows</code> or <code>cols</code> < 0; or <code>elemSize</code> , <code>lda</code> , or <code>ldb</code> <= 0
<code>CUBLAS_STATUS_MAPPING_ERROR</code>	if error accessing GPU memory
<code>CUBLAS_STATUS_SUCCESS</code>	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()`” on page 13
- “Function `cublasIsamin()`” on page 14
- “Function `cublasSasum()`” on page 15
- “Function `cublasSaxpy()`” on page 15
- “Function `cublasScopy()`” on page 16
- “Function `cublasSdot()`” on page 17
- “Function `cublasSnrm2()`” on page 18
- “Function `cublasSrot()`” on page 19
- “Function `cublasSrotg()`” on page 20
- “Function `cublasSrotm()`” on page 21
- “Function `cublasSrotmg()`” on page 22
- “Function `cublasSscal()`” on page 23
- “Function `cublasSswap()`” on page 24

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

<code>n</code>	number of elements in input vector
<code>x</code>	single-precision vector with <code>n</code> elements
<code>incx</code>	storage spacing between elements of <code>x</code>

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

<code>CUBLAS_STATUS_NOT_INITIALIZED</code>	if CUBLAS library was not initialized
<code>CUBLAS_STATUS_ALLOC_FAILED</code>	if function could not allocate reduction buffer
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	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

<code>n</code>	number of elements in input vector
<code>x</code>	single-precision vector with <code>n</code> elements
<code>incx</code>	storage spacing between elements of <code>x</code>

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

<code>CUBLAS_STATUS_NOT_INITIALIZED</code>	if CUBLAS library was not initialized
<code>CUBLAS_STATUS_ALLOC_FAILED</code>	if function could not allocate reduction buffer
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	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 $\text{abs}(x[1 + i * \text{incx}])$.

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 $n \leq 0$ or $\text{incx} \leq 0$, 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

<code>CUBLAS_STATUS_NOT_INITIALIZED</code>	if CUBLAS library was not initialized
<code>CUBLAS_STATUS_ALLOC_FAILED</code>	if function could not allocate reduction buffer
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	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 α 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

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

where

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

$$lx = 1 + (1 - n) * incx;$$

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

Input

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

Output

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

Reference: <http://www.netlib.org/blas/saxpy.f>

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

Error Status

<code>CUBLAS_STATUS_NOT_INITIALIZED</code>	if CUBLAS library was not initialized
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	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

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

where

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

$$lx = 1 + (1 - n) * incx;$$

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

<code>CUBLAS_STATUS_NOT_INITIALIZED</code>	if CUBLAS library was not initialized
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	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

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

$$lx = 1 + (1 - n) * incx;$$

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)

<code>y</code>	single-precision vector with <code>n</code> elements
<code>incy</code>	storage spacing between elements of <code>y</code>

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

<code>CUBLAS_STATUS_NOT_INITIALIZED</code>	if CUBLAS library was not initialized
<code>CUBLAS_STATUS_ALLOC_FAILED</code>	if function could not allocate reduction buffer
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	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

<code>n</code>	number of elements in input vector
<code>x</code>	single-precision vector with <code>n</code> elements
<code>incx</code>	storage spacing between elements of <code>x</code>

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

<code>CUBLAS_STATUS_NOT_INITIALIZED</code>	if CUBLAS library was not initialized
<code>CUBLAS_STATUS_ALLOC_FAILED</code>	if function could not allocate reduction buffer
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	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 $2 \times n$ matrix $\begin{bmatrix} x^T \\ y^T \end{bmatrix}$.

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

$1x = 1$ if $incx \geq 0$, else

$1x = 1 + (1 - n) * incx$;

y is treated similarly using $1y$ and $incy$.

Input

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

Output

<code>x</code>	rotated vector x (unchanged if $n \leq 0$)
<code>y</code>	rotated vector y (unchanged if $n \leq 0$)

Reference: <http://www.netlib.org/blas/srot.f>

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

Error Status

<code>CUBLAS_STATUS_NOT_INITIALIZED</code>	if CUBLAS library was not initialized
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	if function failed to launch on GPU

Function `cublasSrotg()`

`void cublasSrotg (float *sa, float *sb, float *sc, float *ss)` constructs the Givens transformation

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

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

<code>sa</code>	single-precision scalar
<code>sb</code>	single-precision scalar

Output

<code>sa</code>	single-precision r
<code>sb</code>	single-precision z
<code>sc</code>	single-precision result
<code>ss</code>	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 $2 \times n$ 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:

$sflag = -1.0f$	$sflag = 0.0f$
$h = \begin{bmatrix} sh00 & sh01 \\ sh10 & sh11 \end{bmatrix}$	$h = \begin{bmatrix} 1.0f & sh01 \\ sh10 & 1.0f \end{bmatrix}$
$sflag = 1.0f$	$sflag = -2.0f$
$h = \begin{bmatrix} sh00 & 1.0f \\ -1.0f & sh11 \end{bmatrix}$	$h = \begin{bmatrix} 1.0f & 0.0f \\ 0.0f & 1.0f \end{bmatrix}$

Input

n	number of elements in input vectors.
x	single-precision vector with n elements.
$incx$	storage spacing between elements of x .
y	single-precision vector with n elements.
$incy$	storage spacing between elements of y .
$sparam$	5-element vector. $sparam[0]$ is $sflag$ described above. $sparam[1]$ through $sparam[4]$ contain the 2×2 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 $n \leq 0$)
y	rotated vector y (unchanged if $n \leq 0$)

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

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

Error Status

<code>CUBLAS_STATUS_NOT_INITIALIZED</code>	if CUBLAS library was not initialized
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	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

<code>sd1</code>	single-precision scalar.
<code>sd2</code>	single-precision scalar.
<code>sx1</code>	single-precision scalar.
<code>sy1</code>	single-precision scalar.

Output

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

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

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

Output

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

Reference: <http://www.netlib.org/blas/sscal.f>

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

Error Status

<code>CUBLAS_STATUS_NOT_INITIALIZED</code>	if CUBLAS library was not initialized
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	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 $incx \geq 0$, 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 $n \leq 0$)
y	input vector x (unchanged if $n \leq 0$)

Reference: <http://www.netlib.org/blas/sswap.f>

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

Error Status

<code>CUBLAS_STATUS_NOT_INITIALIZED</code>	if CUBLAS library was not initialized
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	if function failed to launch on GPU

Single-precision Complex BLAS1 Functions

The single-precision complex BLAS1 functions are as follows:

- “Function `cublasCaxpy()`” on page 25
- “Function `cublasCcopy()`” on page 26
- “Function `cublasCdotc()`” on page 27
- “Function `cublasCdotu()`” on page 28
- “Function `cublasCrot()`” on page 29
- “Function `cublasCrotg()`” on page 30
- “Function `cublasCscal()`” on page 31
- “Function `cublasCsrot()`” on page 31
- “Function `cublasCsscal()`” on page 32
- “Function `cublasCswap()`” on page 33
- “Function `cublasIcamax()`” on page 34
- “Function `cublasIcamin()`” on page 35
- “Function `cublasScasum()`” on page 35
- “Function `cublasScnrm2()`” on page 36

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
α	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 $n \leq 0$)
-----	--

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

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

Error Status

<code>CUBLAS_STATUS_NOT_INITIALIZED</code>	if CUBLAS library was not initialized
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	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

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

where

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

$$lx = 1 + (1 - n) * incx;$$

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)

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

<code>CUBLAS_STATUS_NOT_INITIALIZED</code>	if CUBLAS library was not initialized
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	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

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

$$lx = 1 + (1 - n) * incx;$$

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

Output

 returns single-precision complex dot product (zero if $n \leq 0$)

 Reference: <http://www.netlib.org/blas/cdotc.f>

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

<code>CUBLAS_STATUS_NOT_INITIALIZED</code>	if CUBLAS library was not initialized
<code>CUBLAS_STATUS_ALLOC_FAILED</code>	if function could not allocate reduction buffer
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	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

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

$$lx = 1 + (1 - n) * incx;$$

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 $n \leq 0$)

Reference: <http://www.netlib.org/blas/cdotu.f>

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

Error Status

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

Function `cublasCrot()`

```
void
cublasCrot (int n, cuComplex *x, int incx, cuComplex *y,
            int incy, float sc, cuComplex cs)
```

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

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

$lx = 1$ if $incx \geq 0$, 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 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
sc	single-precision cosine component of rotation matrix
cs	single-precision complex sine component of rotation matrix

Output

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

Reference: <http://netlib.org/lapack/explore-html/crot.f.html>

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

Error Status

<code>CUBLAS_STATUS_NOT_INITIALIZED</code>	if CUBLAS library was not initialized
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	if function failed to launch on GPU

Function `cublasCrotg()`

```
void
cublasCrotg (cuComplex *ca, cuComplex cb, float *sc,
            float *cs)
```

constructs the complex Givens transformation

$$G = \begin{bmatrix} sc & cs \\ -\overline{cs} & sc \end{bmatrix}, \quad sc*sc + cs*\overline{cs} = 1$$

which zeros the second entry of the complex 2-vector $[ca \ cb]^T$.

The quantity $ca/|ca|*||ca, \ cb||$ overwrites ca in storage. In this case,

$||ca, \ cb|| = scale*\sqrt{|ca/scale|^2 + |cb/scale|^2}$, where
 $scale = |ca| + |cb|$.

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

Input

<code>ca</code>	single-precision complex scalar
<code>cb</code>	single-precision complex scalar

Output

<code>ca</code>	single-precision complex $ca/ ca * ca, \ cb $
<code>sc</code>	single-precision cosine component of rotation matrix
<code>cs</code>	single-precision complex sine component of rotation matrix

Reference: <http://www.netlib.org/blas/crotg.f>

This function does not set any error status.

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

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

where

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

$$lx = 1 + (1 - n) * incx.$$

Input

n	number of elements in input vector
α	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 $n \leq 0$ or $incx \leq 0$)
-----	---

Reference: <http://www.netlib.org/blas/cscal.f>

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

Error Status

<code>CUBLAS_STATUS_NOT_INITIALIZED</code>	if CUBLAS library was not initialized
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	if function failed to launch on GPU

Function cublasCsrot()

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

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

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

$1x = 1$ if $incx \geq 0$, else

$1x = 1 + (1 - n) * incx$;

y is treated similarly using $1y$ and $incy$.

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
sc	single-precision cosine component of rotation matrix
ss	single-precision sine component of rotation matrix

Output

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

Reference: <http://www.netlib.org/blas/csrot.f>

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

Error Status

<code>CUBLAS_STATUS_NOT_INITIALIZED</code>	if CUBLAS library was not initialized
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	if function failed to launch on GPU

Function cublasCscal()

void

cublasCscal (int n , float α , cuComplex $*x$, int $incx$)

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

$x[1x + i * incx]$ with $\alpha * x[1x + i * incx]$,

where

$1x = 1$ if $incx \geq 0$, else

$1x = 1 + (1 - n) * incx$.

Input

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

Output

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

Reference: <http://www.netlib.org/blas/cssscal.f>

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

Error Status

<code>CUBLAS_STATUS_NOT_INITIALIZED</code>	if CUBLAS library was not initialized
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	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[1x + i * incx]` with `y[1y + i * incy]`,

where

```
1x = 1 if incx >= 0, else
1x = 1 + (1 - n) * incx;
```

`1y` is defined in a similar way using `incy`.

Input

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

Output

<code>x</code>	contains single-precision complex vector <code>y</code>
<code>y</code>	contains single-precision complex vector <code>x</code>

Reference: <http://www.netlib.org/blas/cswap.f>

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

Error Status

<code>CUBLAS_STATUS_NOT_INITIALIZED</code>	if CUBLAS library was not initialized
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	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

<code>n</code>	number of elements in input vector
<code>x</code>	single-precision complex vector with <code>n</code> elements
<code>incx</code>	storage spacing between elements of <code>x</code>

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

<code>CUBLAS_STATUS_NOT_INITIALIZED</code>	if CUBLAS library was not initialized
<code>CUBLAS_STATUS_ALLOC_FAILED</code>	if function could not allocate reduction buffer
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	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

<code>n</code>	number of elements in input vector
<code>x</code>	single-precision complex vector with <code>n</code> elements
<code>incx</code>	storage spacing between elements of <code>x</code>

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

<code>CUBLAS_STATUS_NOT_INITIALIZED</code>	if CUBLAS library was not initialized
<code>CUBLAS_STATUS_ALLOC_FAILED</code>	if function could not allocate reduction buffer
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	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

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

where

$$1x = 1 \text{ if } \text{incx} \leq 0, \text{ else}$$

$$1x = 1 + (1 - n) * \text{incx}.$$

Input

<code>n</code>	number of elements in input vector
<code>x</code>	single-precision complex vector with <code>n</code> elements
<code>incx</code>	storage spacing between elements of <code>x</code>

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

<code>CUBLAS_STATUS_NOT_INITIALIZED</code>	if CUBLAS library was not initialized
<code>CUBLAS_STATUS_ALLOC_FAILED</code>	if function could not allocate reduction buffer
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	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

<code>n</code>	number of elements in input vector
<code>x</code>	single-precision complex vector with <code>n</code> elements
<code>incx</code>	storage spacing between elements of <code>x</code>

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

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

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 39
- “Single-precision Complex BLAS2 Functions” on page 60
(*Not yet implemented*)
- “Single-precision BLAS3 Functions” on page 61
- “Single-precision Complex BLAS3 Functions” on page 72

Single-precision BLAS2 Functions

The single-precision BLAS2 functions are as follows:

- “Function `cublasSgbmv()`” on page 39
- “Function `cublasSgemv()`” on page 41
- “Function `cublasSger()`” on page 42
- “Function `cublasSsbmv()`” on page 43
- “Function `cublasSspmv()`” on page 45
- “Function `cublasSspr()`” on page 46
- “Function `cublasSspr2()`” on page 47
- “Function `cublasSsymv()`” on page 48
- “Function `cublasSsyr()`” on page 49
- “Function `cublasSsyr2()`” on page 50
- “Function `cublasStbmv()`” on page 52
- “Function `cublasStbsv()`” on page 53
- “Function `cublasStpmv()`” on page 55
- “Function `cublasStpsv()`” on page 56
- “Function `cublasStrmv()`” on page 57
- “Function `cublasStrsv()`” on page 59

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

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

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

alpha and beta are single-precision scalars, and x and y are single-precision vectors. A is an $m \times n$ band matrix consisting of single-precision elements with k_l subdiagonals and k_u superdiagonals.

Input

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

Reference: <http://www.netlib.org/blas/sgbmv.f>

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

Error Status

<code>CUBLAS_STATUS_NOT_INITIALIZED</code>	if CUBLAS library was not initialized
<code>CUBLAS_STATUS_INVALID_VALUE</code>	if $m < 0$, $n < 0$, $k1 < 0$, $ku < 0$, $incx == 0$, or $incy == 0$
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	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

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

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

α and β are single-precision scalars, and x and y are single-precision vectors. A is an $m \times n$ matrix consisting of single-precision elements. Matrix A is stored in column-major format, and lda is the leading dimension of the two-dimensional array in which A is stored.

Input

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

Input (continued)

<code>y</code>	single-precision array of length at least $(1 + (m - 1) * \text{abs}(\text{incy}))$ if <code>trans == 'N'</code> or <code>'n'</code> , else at least $(1 + (n - 1) * \text{abs}(\text{incy}))$.
<code>incy</code>	the storage spacing between elements of <code>y</code> ; <code>incy</code> must not be zero.

Output

<code>y</code>	updated according to $y = \text{alpha} * \text{op}(A) * x + \text{beta} * y$.
----------------	--

Reference: <http://www.netlib.org/blas/sgemv.f>

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

Error Status

<code>CUBLAS_STATUS_NOT_INITIALIZED</code>	if CUBLAS library was not initialized
<code>CUBLAS_STATUS_INVALID_VALUE</code>	if $m < 0$, $n < 0$, <code>incx == 0</code> , or <code>incy == 0</code>
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	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

$$A = \text{alpha} * x * y^T + A,$$

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

<code>m</code>	specifies the number of rows of the matrix <code>A</code> ; <code>m</code> must be at least zero.
<code>n</code>	specifies the number of columns of matrix <code>A</code> ; <code>n</code> must be at least zero.
<code>alpha</code>	single-precision scalar multiplier applied to $x * y^T$.
<code>x</code>	single-precision array of length at least $(1 + (m - 1) * \text{abs}(\text{incx}))$.
<code>incx</code>	the storage spacing between elements of <code>x</code> ; <code>incx</code> must not be zero.
<code>y</code>	single-precision array of length at least $(1 + (n - 1) * \text{abs}(\text{incy}))$.

Input (continued)

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

Output

<code>A</code>	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

<code>CUBLAS_STATUS_NOT_INITIALIZED</code>	if CUBLAS library was not initialized
<code>CUBLAS_STATUS_INVALID_VALUE</code>	if $m < 0$, $n < 0$, $incx == 0$, or $incy == 0$
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	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 α and β are single-precision scalars, and x and y are n -element single-precision vectors. A is an $n \times n$ symmetric band matrix consisting of single-precision elements, with k superdiagonals and the same number of subdiagonals.

Input

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

Input (continued)

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

Output

`y` updated according to $y = \text{alpha} * A * x + \text{beta} * y$.

Reference: <http://www.netlib.org/blas/ssbmv.f>

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

Error Status

<code>CUBLAS_STATUS_NOT_INITIALIZED</code>	if CUBLAS library was not initialized
<code>CUBLAS_STATUS_INVALID_VALUE</code>	if $k < 0$, $n < 0$, $\text{incx} == 0$, OR $\text{incy} == 0$
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	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

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

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

Input

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

Output

<code>y</code>	updated according to $y = \text{alpha} * A * x + \text{beta} * y$.
----------------	---

Reference: <http://www.netlib.org/blas/sspmv.f>

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

Error Status

<code>CUBLAS_STATUS_NOT_INITIALIZED</code>	if CUBLAS library was not initialized
<code>CUBLAS_STATUS_INVALID_VALUE</code>	if $n < 0$, $incx == 0$, or $incy == 0$
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	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

$$A = \text{alpha} * x * x^T + A,$$

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

Input

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

<code>A</code>	updated according to $A = \text{alpha} * x * x^T + A$.
----------------	---

Reference: <http://www.netlib.org/blas/sspr.f>

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

Error Status

<code>CUBLAS_STATUS_NOT_INITIALIZED</code>	if CUBLAS library was not initialized
<code>CUBLAS_STATUS_INVALID_VALUE</code>	if $n < 0$ or $incx == 0$
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	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

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

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

Input

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

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

<code>CUBLAS_STATUS_NOT_INITIALIZED</code>	if CUBLAS library was not initialized
<code>CUBLAS_STATUS_INVALID_VALUE</code>	if $n < 0$, $incx == 0$, or $incy == 0$
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	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 $n \times n$ matrix that consists of single-precision elements and is stored in either upper or lower storage mode.

Input

<code>uplo</code>	specifies whether the upper or lower triangular part of the array <code>A</code> is referenced. If <code>uplo == 'U'</code> or <code>'u'</code> , the symmetric matrix <code>A</code> is stored in upper storage mode; that is, only the upper triangular part of <code>A</code> is referenced while the lower triangular part of <code>A</code> is inferred. If <code>uplo == 'L'</code> or <code>'l'</code> , the symmetric matrix <code>A</code> is stored in lower storage mode; that is, only the lower triangular part of <code>A</code> is referenced while the upper triangular part of <code>A</code> is inferred.
<code>n</code>	specifies the number of rows and the number of columns of the symmetric matrix <code>A</code> ; <code>n</code> must be at least zero.
<code>alpha</code>	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 $y = \text{alpha} * A * x + \text{beta} * y$.

Reference: <http://www.netlib.org/blas/ssymv.f>

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

Error Status

<code>CUBLAS_STATUS_NOT_INITIALIZED</code>	if CUBLAS library was not initialized
<code>CUBLAS_STATUS_INVALID_VALUE</code>	if $n < 0$, $\text{incx} == 0$, or $\text{incy} == 0$
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	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

$$A = \text{alpha} * x * x^T + A,$$

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 $(1 + (n - 1) * \text{abs}(\text{incx}))$.
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 $A = \text{alpha} * x * x^T + A$.
---	---

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 $\text{incx} == 0$
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

$$A = \text{alpha} * x * y^T + \text{alpha} * y * x^T + A,$$

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

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

Output

<code>A</code>	updated according to $A = \text{alpha} * x * y^T + \text{alpha} * y * x^T + A$.
----------------	--

Reference: <http://www.netlib.org/blas/ssyr2.f>

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

Error Status

<code>CUBLAS_STATUS_NOT_INITIALIZED</code>	if CUBLAS library was not initialized
<code>CUBLAS_STATUS_INVALID_VALUE</code>	if <code>n < 0</code> , <code>incx == 0</code> , or <code>incy == 0</code>
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	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

$$x = \text{op}(A) * x,$$

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

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

Input

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

Input (continued)

<code>x</code>	single-precision array of length at least $(1 + (n - 1) * \text{abs}(\text{incx}))$. On entry, <code>x</code> contains the source vector. On exit, <code>x</code> is overwritten with the result vector.
<code>incx</code>	specifies the storage spacing for elements of <code>x</code> ; <code>incx</code> must not be zero.

Output

<code>x</code>	updated according to $x = \text{op}(A) * x$.
----------------	---

Reference: <http://www.netlib.org/blas/stbmv.f>

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

Error Status

<code>CUBLAS_STATUS_NOT_INITIALIZED</code>	if CUBLAS library was not initialized
<code>CUBLAS_STATUS_INVALID_VALUE</code>	if $n < 0$, $n > 4070$, $k < 0$, or $\text{incx} == 0$
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	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

$$\text{op}(A) * x = b,$$

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

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

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

Input

<code>uplo</code>	specifies whether the matrix is an upper or lower triangular band matrix: If <code>uplo == 'U' or 'u'</code> , <code>A</code> is an upper triangular band matrix. If <code>uplo == 'L' or 'l'</code> , <code>A</code> is a lower triangular band matrix.
<code>trans</code>	specifies <code>op(A)</code> . If <code>trans == 'N' or 'n'</code> , <code>op(A) = A</code> . If <code>trans == 'T', 't', 'C', or 'c'</code> , <code>op(A) = A^T</code> .

Input (continued)

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

Output

<code>x</code>	updated to contain the solution vector <code>x</code> 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

<code>CUBLAS_STATUS_NOT_INITIALIZED</code>	if CUBLAS library was not initialized
<code>CUBLAS_STATUS_INVALID_VALUE</code>	if <code>incx == 0</code> , <code>n < 0</code> , or <code>n > 4070</code>
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	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

$$x = \text{op}(A) * x,$$

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

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

Input

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

Output

<code>x</code>	updated according to $x = \text{op}(A) * x$.
----------------	---

Reference: <http://www.netlib.org/blas/stpmv.f>

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

Error Status

<code>CUBLAS_STATUS_NOT_INITIALIZED</code>	if CUBLAS library was not initialized
<code>CUBLAS_STATUS_INVALID_VALUE</code>	if <code>incx == 0</code> , <code>n < 0</code> , or <code>n > 4070</code>
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	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

$$\text{op}(A) * x = b,$$

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

`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

<code>uplo</code>	specifies whether the matrix is an upper or lower triangular matrix. If <code>uplo == 'U' or 'u'</code> , <code>A</code> is an upper triangular matrix. If <code>uplo == 'L' or 'l'</code> , <code>A</code> is a lower triangular matrix.
<code>trans</code>	specifies <code>op(A)</code> . If <code>trans == 'N' or 'n'</code> , <code>op(A) = A</code> . If <code>trans == 'T', 't', 'C', or 'c'</code> , <code>op(A) = A^T</code> .
<code>diag</code>	specifies whether <code>A</code> is unit triangular. If <code>diag == 'U' or 'u'</code> , <code>A</code> is assumed to be unit triangular; that is, diagonal elements are not read and are assumed to be unity. If <code>diag == 'N' or 'n'</code> , <code>A</code> is not assumed to be unit triangular.
<code>n</code>	specifies the number of rows and columns of the matrix <code>A</code> ; <code>n</code> must be at least zero. In the current implementation <code>n</code> must not exceed 4070.

Input (continued)

<code>AP</code>	single-precision array with at least $(n * (n + 1)) / 2$ elements. If <code>uplo == 'U' or 'u'</code> , array <code>AP</code> contains the upper triangular matrix <code>A</code> , packed sequentially, column by column; that is, if $i \leq j$, <code>A[i, j]</code> is stored in <code>AP[i + (j * (j + 1)) / 2]</code> . If <code>uplo == 'L' or 'l'</code> , array <code>AP</code> contains the lower triangular matrix <code>A</code> , packed sequentially, column by column; that is, if $i \geq j$, <code>A[i, j]</code> is stored in <code>AP[i + ((2 * n - j + 1) * j) / 2]</code> . When <code>diag == 'U' or 'u'</code> , the diagonal elements of <code>A</code> are not referenced and are assumed to be unity.
<code>x</code>	single-precision array of length at least $(1 + (n - 1) * \text{abs}(\text{incx}))$. On entry, <code>x</code> contains the n -element right-hand side vector <code>b</code> . On exit, it is overwritten with the solution vector <code>x</code> .
<code>incx</code>	storage spacing between elements of <code>x</code> ; <code>incx</code> must not be zero.

Output

<code>x</code>	updated to contain the solution vector <code>x</code> that solves $\text{op}(A) * x = b$.
----------------	--

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

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

Error Status

<code>CUBLAS_STATUS_NOT_INITIALIZED</code>	if CUBLAS library was not initialized
<code>CUBLAS_STATUS_INVALID_VALUE</code>	if <code>incx == 0</code> , <code>n < 0</code> , or <code>n > 4070</code>
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	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

$$x = \text{op}(A) * x,$$

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

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

Input

<code>uplo</code>	specifies whether the matrix A is an upper or lower triangular matrix. If <code>uplo == 'U' or 'u'</code> , A is an upper triangular matrix. If <code>uplo == 'L' or 'l'</code> , A is an lower triangular matrix.
<code>trans</code>	specifies $op(A)$. If <code>trans == 'N' or 'n'</code> , $op(A) = A$. If <code>trans == 'T', 't', 'C', or 'c'</code> , $op(A) = A^T$.
<code>diag</code>	specifies whether or not A is a unit triangular matrix. If <code>diag == 'U' or 'u'</code> , A is assumed to be unit triangular. If <code>diag == 'N' or 'n'</code> , A is not assumed to be unit triangular.
<code>n</code>	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.
<code>A</code>	single-precision array of dimensions (lda, n) . If <code>uplo == 'U' or 'u'</code> , the leading $n \times n$ 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 <code>uplo == 'L' or 'l'</code> , the leading $n \times n$ 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 <code>diag == 'U' or 'u'</code> , the diagonal elements of A are not referenced either, but are assumed to be unity.
<code>lda</code>	leading dimension of A ; <code>lda</code> must be at least $\max(1, n)$.
<code>x</code>	single-precision array of length at least $(1 + (n - 1) * \text{abs}(\text{incx}))$. On entry, x contains the source vector. On exit, x is overwritten with the result vector.
<code>incx</code>	the storage spacing between elements of x ; <code>incx</code> must not be zero.

Output

<code>x</code>	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

<code>CUBLAS_STATUS_NOT_INITIALIZED</code>	if CUBLAS library was not initialized
<code>CUBLAS_STATUS_INVALID_VALUE</code>	if <code>incx == 0</code> , <code>n < 0</code> , or <code>n > 4070</code>
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	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

$$\text{op}(A) * x = b,$$

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

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

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

Input

<code>uplo</code>	specifies whether the matrix data is stored in the upper or the lower triangular part of array A . If <code>uplo == 'U' or 'u'</code> , only the upper triangular part of A may be referenced. If <code>uplo == 'L' or 'l'</code> , only the lower triangular part of A may be referenced.
<code>trans</code>	specifies $\text{op}(A)$. If <code>trans == 'N' or 'n'</code> , $\text{op}(A) = A$. If <code>trans == 'T', 't', 'C', or 'c'</code> , $\text{op}(A) = A^T$.
<code>diag</code>	specifies whether or not A is a unit triangular matrix. If <code>diag == 'U' or 'u'</code> , A is assumed to be unit triangular. If <code>diag == 'N' or 'n'</code> , A is not assumed to be unit triangular.
<code>n</code>	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.
<code>A</code>	single-precision array of dimensions (lda, n) . If <code>uplo == 'U' or 'u'</code> , A contains the upper triangular part of the symmetric matrix, and the strictly lower triangular part is not referenced. If <code>uplo == 'L' or 'l'</code> , A contains the lower triangular part of the symmetric matrix, and the strictly upper triangular part is not referenced.
<code>lda</code>	leading dimension of the two-dimensional array containing A ; lda must be at least $\max(1, n)$.
<code>x</code>	single-precision array of length at least $(1 + (n - 1) * \text{abs}(\text{incx}))$. On entry, x contains the n -element, right-hand-side vector b . On exit, it is overwritten with the solution vector x .
<code>incx</code>	the storage spacing between elements of x ; <code>incx</code> 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 <code>incx == 0</code> , <code>n < 0</code> , or <code>n > 4070</code>
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()`” on page 61
- “Function `cublasSsymm()`” on page 63
- “Function `cublasSsyrk()`” on page 65
- “Function `cublasSsyr2k()`” on page 66
- “Function `cublasStrmm()`” on page 68
- “Function `cublasStrsm()`” on page 70

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:

$$C = \alpha * \text{op}(A) * \text{op}(B) + \beta * C,$$

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

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

Input

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

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

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

Input (continued)

<code>n</code>	number of columns of matrix $op(B)$ and number of columns of <code>c</code> ; <code>n</code> must be at least zero.
<code>k</code>	number of columns of matrix $op(A)$ and number of rows of $op(B)$; <code>k</code> must be at least zero.
<code>alpha</code>	single-precision scalar multiplier applied to $op(A) * op(B)$.
<code>A</code>	single-precision array of dimensions (lda, k) if <code>transa == 'N'</code> or <code>'n'</code> , and of dimensions (lda, m) otherwise. If <code>transa == 'N'</code> or <code>'n'</code> , <code>lda</code> must be at least $\max(1, m)$; otherwise, <code>lda</code> must be at least $\max(1, k)$.
<code>lda</code>	leading dimension of two-dimensional array used to store matrix <code>A</code> .
<code>B</code>	single-precision array of dimensions (ldb, n) if <code>transb == 'N'</code> or <code>'n'</code> , and of dimensions (ldb, k) otherwise. If <code>transb == 'N'</code> or <code>'n'</code> , <code>ldb</code> must be at least $\max(1, k)$; otherwise, <code>ldb</code> must be at least $\max(1, n)$.
<code>ldb</code>	leading dimension of two-dimensional array used to store matrix <code>B</code> .
<code>beta</code>	single-precision scalar multiplier applied to <code>c</code> . If zero, <code>c</code> does not have to be a valid input.
<code>C</code>	single-precision array of dimensions (ldc, n) ; <code>ldc</code> must be at least $\max(1, m)$.
<code>ldc</code>	leading dimension of two-dimensional array used to store matrix <code>c</code> .

Output

<code>C</code>	updated based on $C = \text{alpha} * op(A) * op(B) + \text{beta} * C$.
----------------	---

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

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

where α and β are single-precision scalars, A is a symmetric matrix consisting of single-precision elements and is stored in either lower or upper storage mode. B and C are $m \times n$ matrices consisting of single-precision elements.

Input

<code>side</code>	specifies whether the symmetric matrix A appears on the left-hand side or right-hand side of matrix B . If <code>side == 'L' OR 'l'</code> , $C = \alpha * A * B + \beta * C$. If <code>side == 'R' OR 'r'</code> , $C = \alpha * B * A + \beta * C$.
<code>uplo</code>	specifies whether the symmetric matrix A is stored in upper or lower storage mode. If <code>uplo == 'U' OR 'u'</code> , 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 <code>uplo == 'L' OR 'l'</code> , 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.
<code>m</code>	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 <code>side == 'L' OR 'l'</code> ; m must be at least zero.
<code>n</code>	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 <code>side == 'R' OR 'r'</code> ; n must be at least zero.
<code>alpha</code>	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 $m \times m$ part of array A must contain the symmetric matrix, such that when $uplo == 'U'$ or $'u'$, the leading $m \times m$ 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 $m \times m$ 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 $n \times n$ part of array A must contain the symmetric matrix, such that when $uplo == 'U'$ or $'u'$, the leading $n \times n$ 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 $n \times n$ 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 $m \times n$ 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 cublasSsyrrk()

```
void
cublasSsyrrk (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

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

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

Input

<code>uplo</code>	specifies whether the symmetric matrix <code>C</code> is stored in upper or lower storage mode. If <code>uplo == 'U' or 'u'</code> , 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 <code>uplo == 'L' or 'l'</code> , 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.
<code>trans</code>	specifies the operation to be performed. If <code>trans == 'N' or 'n'</code> , $C = \alpha * A * A^T + \beta * C$. If <code>trans == 'T', 't', 'C', or 'c'</code> , $C = \alpha * A^T * A + \beta * C$.
<code>n</code>	specifies the number of rows and the number columns of matrix <code>C</code> . If <code>trans == 'N' or 'n'</code> , <code>n</code> specifies the number of rows of matrix <code>A</code> . If <code>trans == 'T', 't', 'C', or 'c'</code> , <code>n</code> specifies the number of columns of matrix <code>A</code> ; <code>n</code> must be at least zero.
<code>k</code>	If <code>trans == 'N' or 'n'</code> , <code>k</code> specifies the number of columns of matrix <code>A</code> . If <code>trans == 'T', 't', 'C', or 'c'</code> , <code>k</code> specifies the number of rows of matrix <code>A</code> ; <code>k</code> must be at least zero.
<code>alpha</code>	single-precision scalar multiplier applied to $A * A^T$ or $A^T * A$.
<code>A</code>	single-precision array of dimensions (lda, ka) , where <code>ka</code> is <code>k</code> when <code>trans == 'N' or 'n'</code> , and is <code>n</code> otherwise. When <code>trans == 'N' or 'n'</code> , the leading $n \times k$ part of array <code>A</code> contains the matrix <code>A</code> ; otherwise, the leading $k \times n$ part of the array contains the matrix <code>A</code> .
<code>lda</code>	leading dimension of <code>A</code> . When <code>trans == 'N' or 'n'</code> , <code>lda</code> must be at least $\max(1, n)$. Otherwise <code>lda</code> 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 <code>uplo == 'U' or 'u'</code> , the leading $n \times n$ 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 <code>uplo == 'L' or 'l'</code> , the leading $n \times n$ 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

<code>CUBLAS_STATUS_NOT_INITIALIZED</code>	if CUBLAS library was not initialized
<code>CUBLAS_STATUS_INVALID_VALUE</code>	if $n < 0$ or $k < 0$
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	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 $n \times n$ symmetric matrix consisting of single-precision elements and is stored

in either lower or upper storage mode. A and B are matrices consisting of single-precision elements with dimension of $n \times k$ in the first case, and $k \times n$ in the second case.

Input

<code>uplo</code>	specifies whether the symmetric matrix <code>c</code> is stored in upper or lower storage mode. If <code>uplo == 'U' or 'u'</code> , 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 <code>uplo == 'L' or 'l'</code> , 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.
<code>trans</code>	specifies the operation to be performed. If <code>trans == 'N' or 'n'</code> , $C = \alpha * A * B^T + \alpha * B * A^T + \beta * C$. If <code>trans == 'T', 't', 'C', or 'c'</code> , $C = \alpha * A^T * B + \alpha * B^T * A + \beta * C$.
<code>n</code>	specifies the number of rows and the number columns of matrix <code>c</code> . If <code>trans == 'N' or 'n'</code> , <code>n</code> specifies the number of rows of matrix A. If <code>trans == 'T', 't', 'C', or 'c'</code> , <code>n</code> specifies the number of columns of matrix A; <code>n</code> must be at least zero.
<code>k</code>	If <code>trans == 'N' or 'n'</code> , <code>k</code> specifies the number of columns of matrix A. If <code>trans == 'T', 't', 'C', or 'c'</code> , <code>k</code> specifies the number of rows of matrix A; <code>k</code> must be at least zero.
<code>alpha</code>	single-precision scalar multiplier.
<code>A</code>	single-precision array of dimensions (lda, ka) , where ka is k when <code>trans == 'N' or 'n'</code> , and is n otherwise. When <code>trans == 'N' or 'n'</code> , the leading $n \times k$ part of array A must contain the matrix A, otherwise the leading $k \times n$ part of the array must contain the matrix A.
<code>lda</code>	leading dimension of A. When <code>trans == 'N' or 'n'</code> , <code>lda</code> must be at least $\max(1, n)$. Otherwise <code>lda</code> must be at least $\max(1, k)$.
<code>B</code>	single-precision array of dimensions (lda, kb) , where $kb = k$ when <code>trans == 'N' or 'n'</code> , and $k = n$ otherwise. When <code>trans == 'N' or 'n'</code> , the leading $n \times k$ part of array B must contain the matrix B, otherwise the leading $k \times n$ part of the array must contain the matrix B.
<code>ldb</code>	leading dimension of B. When <code>trans == 'N' or 'n'</code> , <code>ldb</code> must be at least $\max(1, n)$. Otherwise <code>ldb</code> must be at least $\max(1, k)$.
<code>beta</code>	single-precision scalar multiplier applied to <code>c</code> . If <code>beta</code> is zero, <code>c</code> does not have to be a valid input.

Input (continued)

<code>C</code>	single-precision array of dimensions (ldc, n) . If <code>uplo == 'U' or 'u'</code> , the leading $n \times n$ triangular part of the array <code>C</code> must contain the upper triangular part of the symmetric matrix <code>C</code> , and the strictly lower triangular part of <code>C</code> is not referenced. On exit, the upper triangular part of <code>C</code> is overwritten by the upper triangular part of the updated matrix. If <code>uplo == 'L' or 'l'</code> , the leading $n \times n$ triangular part of the array <code>C</code> must contain the lower triangular part of the symmetric matrix <code>C</code> , and the strictly upper triangular part of <code>C</code> is not referenced. On exit, the lower triangular part of <code>C</code> is overwritten by the lower triangular part of the updated matrix.
<code>ldc</code>	leading dimension of <code>C</code> ; <code>ldc</code> must be at least $\max(1, n)$.

Output

<code>C</code>	updated according to $C = \alpha * A * B^T + \alpha * B * A^T + \beta * C$ <i>or</i> $C = \alpha * A^T * B + \alpha * B^T * A + \beta * C$
----------------	--

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

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

Error Status

<code>CUBLAS_STATUS_NOT_INITIALIZED</code>	if CUBLAS library was not initialized
<code>CUBLAS_STATUS_INVALID_VALUE</code>	if $n < 0$ or $k < 0$
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	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

$$B = \alpha * \text{op}(A) * B \text{ or } B = \alpha * B * \text{op}(A),$$

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

`alpha` is a single-precision scalar, `B` is an $m \times n$ 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

<code>side</code>	specifies whether $op(A)$ multiplies B from the left or right. If <code>side == 'L' OR 'l'</code> , $B = \alpha * op(A) * B$. If <code>side == 'R' OR 'r'</code> , $B = \alpha * B * op(A)$.
<code>uplo</code>	specifies whether the matrix A is an upper or lower triangular matrix. If <code>uplo == 'U' OR 'u'</code> , A is an upper triangular matrix. If <code>uplo == 'L' OR 'l'</code> , A is a lower triangular matrix.
<code>transa</code>	specifies the form of $op(A)$ to be used in the matrix multiplication. If <code>transa == 'N' OR 'n'</code> , $op(A) = A$. If <code>transa == 'T', 't', 'C', OR 'c'</code> , $op(A) = A^T$.
<code>diag</code>	specifies whether or not A is a unit triangular matrix. If <code>diag == 'U' OR 'u'</code> , A is assumed to be unit triangular. If <code>diag == 'N' OR 'n'</code> , A is not assumed to be unit triangular.
<code>m</code>	the number of rows of matrix B ; m must be at least zero.
<code>n</code>	the number of columns of matrix B ; n must be at least zero.
<code>alpha</code>	single-precision scalar multiplier applied to $op(A) * B$ OR $B * op(A)$, respectively. If <code>alpha</code> is zero, no accesses are made to matrix A , and no read accesses are made to matrix B .
<code>A</code>	single-precision array of dimensions (lda, k) . If <code>side == 'L' OR 'l'</code> , $k = m$. If <code>side == 'R' OR 'r'</code> , $k = n$. If <code>uplo == 'U' OR 'u'</code> , the leading $k \times k$ upper triangular part of the array A must contain the upper triangular matrix, and the strictly lower triangular part of A is not referenced. If <code>uplo == 'L' OR 'l'</code> , the leading $k \times k$ lower triangular part of the array A must contain the lower triangular matrix, and the strictly upper triangular part of A is not referenced. When <code>diag == 'U' OR 'u'</code> , the diagonal elements of A are not referenced and are assumed to be unity.
<code>lda</code>	leading dimension of A . When <code>side == 'L' OR 'l'</code> , it must be at least $\max(1, m)$ and at least $\max(1, n)$ otherwise.
<code>B</code>	single-precision array of dimensions (ldb, n) . On entry, the leading $m \times n$ part of the array contains the matrix B . It is overwritten with the transformed matrix on exit.
<code>ldb</code>	leading dimension of B ; <code>ldb</code> must be at least $\max(1, m)$.

Output

<code>B</code>	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

<code>CUBLAS_STATUS_NOT_INITIALIZED</code>	if CUBLAS library was not initialized
<code>CUBLAS_STATUS_INVALID_VALUE</code>	if $m < 0$ OR $n < 0$
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	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

$$\text{op}(A) * X = \text{alpha} * B \text{ or } X * \text{op}(A) = \text{alpha} * B,$$

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

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

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

Input

side	specifies whether $\text{op}(A)$ appears on the left or right of x: side == 'L' or 'l' indicates solve $\text{op}(A) * X = \text{alpha} * B$; side == 'R' or 'r' indicates solve $X * \text{op}(A) = \text{alpha} * B$.
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 $\text{op}(A)$ to be used in matrix multiplication. If transa == 'N' OR 'n', $\text{op}(A) = A$. If transa == 'T', 't', 'C', OR 'c', $\text{op}(A) = A^T$.
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)

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

Output

<code>B</code>	contains the solution matrix <code>X</code> 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

<code>CUBLAS_STATUS_NOT_INITIALIZED</code>	if CUBLAS library was not initialized
<code>CUBLAS_STATUS_INVALID_VALUE</code>	if $m < 0$ or $n < 0$
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	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

$$C = \text{alpha} * \text{op}(A) * \text{op}(B) + \text{beta} * C,$$

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

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 $m \times k$ matrix, `op(B)` a $k \times n$ matrix and `C` an $m \times n$ matrix.

Input

-
- `transa` specifies `op(A)`. If `transa == 'N' OR 'n'`, `op(A) = A`.
If `transa == 'T' OR 't'`, `op(A) = AT`.
If `transa == 'C' OR 'c'`, `op(A) = AH`.
 - `transb` specifies `op(B)`. If `transb == 'N' OR 'n'`, `op(B) = B`.
If `transb == 'T' OR 't'`, `op(B) = BT`.
If `transb == 'C' OR 'c'`, `op(B) = BH`.
 - `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)

<code>ldb</code>	leading dimension of <code>B</code> . When <code>transb == 'N'</code> or <code>'n'</code> , it must be at least $\max(1, k)$ and at least $\max(1, n)$ otherwise.
<code>beta</code>	single-precision complex scalar multiplier applied to <code>c</code> . If <code>beta</code> is zero, <code>c</code> does not have to be a valid input.
<code>C</code>	single-precision array of dimensions (ldc, n) .
<code>ldc</code>	leading dimension of <code>c</code> ; <code>ldc</code> must be at least $\max(1, m)$.

Output

<code>C</code>	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

<code>CUBLAS_STATUS_NOT_INITIALIZED</code>	if CUBLAS library was not initialized
<code>CUBLAS_STATUS_INVALID_VALUE</code>	if $m < 0$, $n < 0$, or $k < 0$
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	if function failed to launch on GPU

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 and 64-bit Microsoft Windows XP. 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 thinking 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-thinking wrappers are used for production code. To enable the thinking wrappers, symbol `CUBLAS_USE_THINKING` must be defined for the compilation of `fortran.c`.

The non-thinking 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-thinking 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-thinking 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
enddo

```

Example A.2. Same Application Using Non-thinking 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
```
