



CUDA **CUBLAS Library**

PG-00000-002_V3.1
May, 2010

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

© 2005–2010 by NVIDIA Corporation. All rights reserved.

Portions of the SGEMM, DGEMM and ZGEMM library routines were written by Vasily Volkov and are subject to the Modified Berkeley Software Distribution License as follows:

Copyright (c) 2007-2009, Regents of the University of California

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

Neither the name of the University of California, Berkeley nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Portions of the SGEMM, DGEMM and ZGEMM library routines were written by Davide Barbieri and are subject to the Modified Berkeley Software Distribution License as follows:

Copyright (c) 2008-2009 Davide Barbieri @ University of Rome Tor Vergata.

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- 1- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- 2- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- 3- The name of the author may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Table of Contents

1. The CUBLAS Library	1
CUBLAS Types	8
Type cublasStatus	8
CUBLAS Helper Functions	9
Function cublasInit()	9
Function cublasShutdown()	10
Function cublasGetError()	10
Function cublasAlloc()	10
Function cublasFree()	11
Function cublasSetVector()	11
Function cublasGetVector()	12
Function cublasSetMatrix()	13
Function cublasGetMatrix()	13
Function cublasSetKernelStream()	14
Function cublasSetVectorAsync()	14
Function cublasGetVectorAsync()	15
Function cublasSetMatrixAsync()	15
Function cublasGetMatrixAsync()	16
2. BLAS1 Functions	17
Single-Precision BLAS1 Functions	18
Function cublasIsamax()	19
Function cublasIsamin()	19
Function cublasSasum()	20
Function cublasSaxpy()	21
Function cublasScopy()	22
Function cublasSdot()	23
Function cublasSnrm2()	24
Function cublasSrot()	24
Function cublasSrotg()	25
Function cublasSrotm()	26
Function cublasSrotmg()	28
Function cublasSscal()	29
Function cublasSswap()	30
Single-Precision Complex BLAS1 Functions	31

Function cublasCaxpy()	.32
Function cublasCcopy()	.33
Function cublasCdotc()	.34
Function cublasCdotu()	.35
Function cublasCrot()	.36
Function cublasCrotg()	.37
Function cublasCscal()	.38
Function cublasCsrot()	.38
Function cublasCsscal()	.39
Function cublasCswap()	.40
Function cublasIcamax()	.41
Function cublasIcamin()	.42
Function cublasScasum()	.42
Function cublasScnrm2()	.43
Double-Precision BLAS1 Functions	.45
Function cublasIdamax()	.46
Function cublasIdamin()	.46
Function cublasDasum()	.47
Function cublasDaxpy()	.48
Function cublasDcopy()	.49
Function cublasDdot()	.50
Function cublasDnrm2()	.51
Function cublasDrot()	.52
Function cublasDrotg()	.53
Function cublasDrotm()	.54
Function cublasDrotmg()	.55
Function cublasDscal()	.56
Function cublasDswap()	.57
Double-Precision Complex BLAS1 functions	.59
Function cublasDzasum()	.60
Function cublasDznrm2()	.61
Function cublasIzamax()	.61
Function cublasIzamin()	.62
Function cublasZaxpy()	.63
Function cublasZcopy()	.64
Function cublasZdotc()	.65
Function cublasZdotu()	.66
Function cublasZdrot()	.67
Function cublasZdscal()	.68
Function cublasZrot()	.69
Function cublasZrotg()	.70
Function cublasZscal()	.70
Function cublasZswap()	.71

3. Single-Precision BLAS2 Functions 73

Single-Precision BLAS2 Functions	74
Function cublasSgbmv()	75
Function cublasSgemv()	76
Function cublasSger()	77
Function cublasSsbmv()	78
Function cublasSspmv()	80
Function cublasSspr()	81
Function cublasSspr2()	82
Function cublasSsymv()	84
Function cublasSsyr()	85
Function cublasSsyr2()	86
Function cublasStbmv()	88
Function cublasStbsv()	89
Function cublasStpmv()	91
Function cublasStpsv()	92
Function cublasStrmv()	94
Function cublasStrsv()	95
Single-Precision Complex BLAS2 Functions	97
Function cublasCgbmv()	98
Function cublasCgemv()	99
Function cublasCgerc()	101
Function cublasCgeru()	102
Function cublasChbmv()	103
Function cublasChemv()	105
Function cublasCher()	106
Function cublasCher2()	107
Function cublasChpmv()	109
Function cublasChpr()	110
Function cublasChpr2()	111
Function cublasCtbmv()	113
Function cublasCtbsv()	115
Function cublasCtpmv()	116
Function cublasCtpsv()	118
Function cublasCtrmv()	119
Function cublasCtrsv()	121

4. Double-Precision BLAS2 Functions 123

Double-Precision BLAS2 Functions	124
Function cublasDgbmv()	125
Function cublasDgemv()	126
Function cublasDger()	128
Function cublasDsbmv()	129
Function cublasDspmv()	131

Function cublasDspr()	132
Function cublasDspr2()	133
Function cublasDsymv()	134
Function cublasDsyr()	136
Function cublasDsyr2()	137
Function cublasDtbmv()	138
Function cublasDtbsv()	140
Function cublasDtpmv()	142
Function cublasDtpsv()	143
Function cublasDtrmv()	144
Function cublasDtrsv()	146
Double-Precision Complex BLAS2 functions	148
Function cublasZgbmv()	149
Function cublasZgemv()	151
Function cublasZgerc()	152
Function cublasZgeru()	153
Function cublasZhbmvm()	155
Function cublasZhemv()	157
Function cublasZher()	158
Function cublasZher2()	160
Function cublasZhpmv()	161
Function cublasZhpr()	163
Function cublasZhpr2()	164
Function cublasZtbmv()	165
Function cublasZtbsv()	167
Function cublasZtpmv()	169
Function cublasZtpsv()	170
Function cublasZtrmv()	172
Function cublasZtrsv()	173
5. BLAS3 Functions	175
Single-Precision BLAS3 Functions	176
Function cublasSgemm()	177
Function cublasSsymm()	178
Function cublasSsyrk()	180
Function cublasSsyr2k()	182
Function cublasStrmm()	184
Function cublasStrsm()	186
Single-Precision Complex BLAS3 Functions	189
Function cublasCgemm()	190
Function cublasChemmm()	191
Function cublasCherk()	193
Function cublasCher2k()	195
Function cublasCsymm()	197

Function cublasCsyrk()	199
Function cublasCsyr2k()	201
Function cublasCtrmm()	203
Function cublasCtrsm()	205
Double-Precision BLAS3 Functions	208
Function cublasDgemm()	209
Function cublasDsymb()	210
Function cublasDsyrk()	212
Function cublasDsyr2k()	214
Function cublasDtrmm()	216
Function cublasDtrsm()	218
Double-Precision Complex BLAS3 Functions	221
Function cublasZgemm()	222
Function cublasZhemm()	223
Function cublasZherk()	225
Function cublasZher2k()	228
Function cublasZsymb()	230
Function cublasZsyrk()	232
Function cublasZsyr2k()	234
Function cublasZtrmm()	236
Function cublasZtrsm()	238
A. CUBLAS Fortran Bindings	241

The CUBLAS Library

CUBLAS is an implementation of BLAS (Basic Linear Algebra Subprograms) on top of the NVIDIA® CUDA™ runtime. 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. CUBLAS attaches to a single GPU and does not auto-parallelize across multiple GPUs.

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 choose 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.

The interface to the CUBLAS library is the header file `cublas.h`. Applications using CUBLAS need to link against the DSO `cublas.so` (Linux), the DLL `cublas.dll` (Windows), or the dynamic library `cublas.dylib` (Mac OS X) when building for the device, and against the DSO `cublasemu.so` (Linux), the DLL `cublasemu.dll` (Windows), or the dynamic library `cublasemu.dylib` (Mac OS X) when building for device emulation.

Following these three examples, the remainder of this chapter discusses [“CUBLAS Types” on page 8](#) and [“CUBLAS Helper Functions” on page 9](#).

Example 1. Fortran 77 Application Executing on the Host

```

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

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

```

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

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

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

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

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

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

```

    if (stat != CUBLAS_STATUS_SUCCESS) {
        printf ("device memory allocation failed");
        cublasShutdown();
        return EXIT_FAILURE;
    }
    stat = cublasSetMatrix (M, N, sizeof(*a), a, M, devPtrA, M);
    if (stat != CUBLAS_STATUS_SUCCESS) {
        printf ("data download failed");
        cublasFree (devPtrA);
        cublasShutdown();
        return EXIT_FAILURE;
    }
    modify (devPtrA, M, N, 2, 3, 16.0f, 12.0f);
    stat = cublasGetMatrix (M, N, sizeof(*a), devPtrA, M, a, M);
    if (stat != CUBLAS_STATUS_SUCCESS) {
        printf ("data upload failed");
        cublasFree (devPtrA);
        cublasShutdown();
        return EXIT_FAILURE;
    }
    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
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) {
```


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

```

    printf ("device memory allocation failed");
    cublasShutdown();
    return EXIT_FAILURE;
}
stat = cublasSetMatrix (M, N, sizeof(*a), a, M, devPtrA, M);
if (stat != CUBLAS_STATUS_SUCCESS) {
    printf ("data download failed");
    cublasFree (devPtrA);
    cublasShutdown();
    return EXIT_FAILURE;
}
modify (devPtrA, M, N, 1, 2, 16.0f, 12.0f);
stat = cublasGetMatrix (M, N, sizeof(*a), devPtrA, M, a, M);
if (stat != CUBLAS_STATUS_SUCCESS) {
    printf ("data upload failed");
    cublasFree (devPtrA);
    cublasShutdown();
    return EXIT_FAILURE;
}
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_ARCH_MISMATCH</code>	function requires an architectural feature absent from the architecture of the device
<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 9
- ❑ “Function `cublasShutdown()`” on page 10
- ❑ “Function `cublasGetError()`” on page 10
- ❑ “Function `cublasAlloc()`” on page 10
- ❑ “Function `cublasFree()`” on page 11
- ❑ “Function `cublasSetVector()`” on page 11
- ❑ “Function `cublasGetVector()`” on page 12
- ❑ “Function `cublasSetMatrix()`” on page 13
- ❑ “Function `cublasGetMatrix()`” on page 13
- ❑ “Function `cublasSetKernelStream()`” on page 14
- ❑ “Function `cublasSetVectorAsync()`” on page 14
- ❑ “Function `cublasGetVectorAsync()`” on page 15
- ❑ “Function `cublasSetMatrixAsync()`” on page 15
- ❑ “Function `cublasGetMatrixAsync()`” on page 16

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. It attaches CUBLAS to whatever GPU is currently bound to the host thread from which it was invoked.

Return Values

<code>CUBLAS_STATUS_ALLOC_FAILED</code>	if resources could not be allocated
<code>CUBLAS_STATUS_SUCCESS</code>	if CUBLAS library initialized successfully

Function `cublasShutdown()`

cublasStatus
cublasShutdown (void)

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

Return Values

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

Function `cublasGetError()`

cublasStatus
cublasGetError (void)

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

Function `cublasAlloc()`

cublasStatus
cublasAlloc (int n, int elemSize, void **devicePtr)

creates an object in GPU memory space capable of holding an array of `n` elements, where each element requires `elemSize` bytes of storage. If the function call is successful, a pointer to the object in GPU memory space is placed in `devicePtr`. Note that this is a device pointer that cannot be dereferenced in host code. Function **cublasAlloc()** is a wrapper around **cudaMalloc()**. Device pointers returned by

cublasAlloc() can therefore be passed to any CUDA device kernels, not just CUBLAS functions.

Return Values

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

Function cublasFree()

cublasStatus

cublasFree (const void *devicePtr)

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

Return Values

CUBLAS_STATUS_NOT_INITIALIZED	if CUBLAS library was not initialized
CUBLAS_STATUS_INTERNAL_ERROR	if the object could not be deallocated
CUBLAS_STATUS_SUCCESS	if object was deallocated successfully

Function cublasSetVector()

cublasStatus

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

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

Similarly, using an increment equal to the leading dimension of the matrix accesses a (partial) row.

Return Values

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

Function `cublasGetVector()`

cublasStatus

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

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

Return Values

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

Function `cublasSetMatrix()`

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

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

Return Values

<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

Function `cublasSetKernelStream()`

`cublasStatus`

`cublasSetKernelStream (cudaStream_t stream)`

sets the CUBLAS stream in which all subsequent CUBLAS kernel launches will run.

By default, if the CUBLAS stream is not set, all kernels use the `NULL` stream. This routine can be used to change the stream between kernel launches and can be used also to set the CUBLAS stream back to `NULL`.

Return Values

<code>CUBLAS_STATUS_NOT_INITIALIZED</code>	if CUBLAS library was not initialized
<code>CUBLAS_STATUS_SUCCESS</code>	if stream set successfully

Function `cublasSetVectorAsync()`

`cublasStatus`

`cublasSetVectorAsync (int n, int elemSize, const void *x,
int incx, void *y, int incy,
cudaStream_t stream);`

has the same functionality as `cublasSetVector()`, but the transfer is done asynchronously within the CUDA stream passed in via parameter.

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> <= 0

Return Values (continued)

<code>CUBLAS_STATUS_MAPPING_ERROR</code>	if error accessing GPU memory
<code>CUBLAS_STATUS_SUCCESS</code>	if operation completed successfully

Function `cublasGetVectorAsync()`

```

cublasStatus
cublasGetVectorAsync (int n, int elemSize, const void *x,
                     int incx, void *y, int incy,
                     cudaStream_t stream)

```

has the same functionality as `cublasGetVector()`, but the transfer is done asynchronously within the CUDA stream passed in via parameter.

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> ≤ 0
<code>CUBLAS_STATUS_MAPPING_ERROR</code>	if error accessing GPU memory
<code>CUBLAS_STATUS_SUCCESS</code>	if operation completed successfully

Function `cublasSetMatrixAsync()`

```

cublasStatus
cublasSetMatrixAsync (int rows, int cols, int elemSize,
                     const void *A, int lda, void *B,
                     int ldb, cudaStream_t stream)

```

has the same functionality as `cublasSetMatrix()`, but the transfer is done asynchronously within the CUDA stream passed in via parameter.

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 cublasGetMatrixAsync()

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

has the same functionality as `cublasGetMatrix()`, but the transfer is done asynchronously within the CUDA stream passed in via parameter.

Return Values

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

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 18
- ❑ “Single-Precision Complex BLAS1 Functions” on page 31
- ❑ “Double-Precision BLAS1 Functions” on page 45
- ❑ “Double-Precision Complex BLAS1 functions” on page 59

Single-Precision BLAS1 Functions

The single-precision BLAS1 functions are as follows:

- ❑ “Function `cublasIsamax()`” on page 19
- ❑ “Function `cublasIsamin()`” on page 19
- ❑ “Function `cublasSasum()`” on page 20
- ❑ “Function `cublasSaxpy()`” on page 21
- ❑ “Function `cublasScopy()`” on page 22
- ❑ “Function `cublasSdot()`” on page 23
- ❑ “Function `cublasSnrm2()`” on page 24
- ❑ “Function `cublasSrot()`” on page 24
- ❑ “Function `cublasSrotg()`” on page 25
- ❑ “Function `cublasSrotm()`” on page 26
- ❑ “Function `cublasSrotmg()`” on page 28
- ❑ “Function `cublasSscal()`” on page 29
- ❑ “Function `cublasSswap()`” on page 30

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 $\text{abs}(x[1 + i * \text{incx}])$. The result reflects 1-based indexing for compatibility with Fortran.

Input

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

Output

returns the smallest index (returns zero if $n \leq 0$ or $\text{incx} \leq 0$)

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 $\text{abs}(x[1 + i * \text{incx}])$. The result reflects 1-based indexing for compatibility with Fortran.

Input

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

Output

returns the smallest index (returns zero if $n \leq 0$ or $incx \leq 0$)

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

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

Error Status

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

Function cublasSasum()

float

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

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

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 $incx \leq 0$, 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

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

Function cublasSaxpy()

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

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

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

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

where

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

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

Input

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

Output

<i>y</i>	single-precision result (unchanged if <i>n</i> <= 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

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

Output

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

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

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

Output

returns single-precision dot product (returns zero if <i>n</i> <= 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 cublasSnorm2()

```
float
cublasSnorm2 (int n, const float *x, int incx)
```

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

Input

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

Output

returns the Euclidian norm
(returns zero if $n \leq 0$, $incx \leq 0$, or if an error occurred)

Reference: <http://www.netlib.org/blas/snorm2.f>

Reference: <http://www.netlib.org/slatec/lin/snorm2.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 $x[lx + i * incx]$, $i = 0$ to $n-1$, where

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

y is treated similarly using ly and $incy$.

Input

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

Output

x	rotated vector x (unchanged if $n \leq 0$)
y	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 *host_sa, float *host_sb,
             float *host_sc, float *host_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. Note that this function is provided for completeness and is run exclusively on the host.

Input

sa	single-precision scalar
sb	single-precision scalar

Output

sa	single-precision r
sb	single-precision z
sc	single-precision result
ss	single-precision result

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

This function does not set any error status.

Function cublasSrotm()

```

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

```

applies the modified Givens transformation, h , to the $2 \times n$ 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:

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

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

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

Function cublasSrotmg()

```
void
cublasSrotmg (float *host_sd1, float *host_sd2,
              float *host_sx1, const float *host_sy1,
              float *host_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:

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

$$sflag = 0.0f \qquad h = \begin{bmatrix} 1.0f & sh01 \\ sh10 & 1.0f \end{bmatrix}$$

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

$$sflag = -2.0f \qquad h = \begin{bmatrix} 1.0f & 0.0f \\ 0.0f & 1.0f \end{bmatrix}$$

$sparam[1]$ 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 $sparam$. Note that this function is provided for completeness and is run exclusively on the host.

Input

sd1	single-precision scalar.
sd2	single-precision scalar.
sx1	single-precision scalar.
sy1	single-precision scalar.

Output

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

Reference: <http://www.netlib.org/blas/srotmg.f>
This function does not set any error status.

Function cublasSscal()

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

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

where

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

Input

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

Output

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

Reference: <http://www.netlib.org/blas/sscal.f>
Error status for this function can be retrieved via **cublasGetError()**.
Error Status

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

Function cublasSswap()

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

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

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

where

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

ly is defined in a similar 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	single-precision vector y (unchanged from input if n <= 0)
y	single-precision vector x (unchanged from input if n <= 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 32
- ❑ “Function `cublasCcopy()`” on page 33
- ❑ “Function `cublasCdotc()`” on page 34
- ❑ “Function `cublasCdotu()`” on page 35
- ❑ “Function `cublasCrot()`” on page 36
- ❑ “Function `cublasCrotg()`” on page 37
- ❑ “Function `cublasCscal()`” on page 38
- ❑ “Function `cublasCsrot()`” on page 38
- ❑ “Function `cublasCsscal()`” on page 39
- ❑ “Function `cublasCswap()`” on page 40
- ❑ “Function `cublasIcamax()`” on page 41
- ❑ “Function `cublasIcamin()`” on page 42
- ❑ “Function `cublasScasum()`” on page 42
- ❑ “Function `cublasScnrm2()`” on page 43

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 $incx \geq 0$, else
 $lx = 1 + (1 - n) * incx$;

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

Input

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

Output

<i>y</i>	single-precision complex result (unchanged if <i>n</i> <= 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

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

where

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

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

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

Input

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

Output

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 complex conjugate of single-precision complex vector x and the single-precision complex vector y if successful, and complex zero otherwise. For $i = 0$ to $n-1$, it sums the products

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

where

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

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

Input

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

Output

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

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

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

Error Status

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

Function cublasCdotu()

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

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

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

where

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

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

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

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

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

y is treated similarly using ly and incy.

Input

n	number of elements in input vectors
x	single-precision 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 <= 0)
y	rotated vector y (unchanged if n <= 0)

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

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

Error Status

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

Function cublasCrotg()

```
void
cublasCrotg (cuComplex *host_ca, cuComplex cb,
             float *host_sc, float *host_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 $\begin{bmatrix} ca & cb \end{bmatrix}^T$.

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. Note that this function is provided for completeness and is run exclusively on the host.

Input

ca	single-precision complex scalar
cb	single-precision complex scalar

Output

ca	single-precision complex $ca/ ca * ca, cb $
sc	single-precision cosine component of rotation matrix
cs	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

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

where

$lx = 1$ if $incx \geq 0$, 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 $x[lx + i * incx]$, $i = 0$ 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
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 `cublasCsscal()`

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

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

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

where

$lx = 1$ if $incx \geq 0$, 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 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</code> <= 0 or <code>incx</code> <= 0)
----------------	----------------------------------------------------------------------------------------------

Reference: <http://www.netlib.org/blas/csscal.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[lx + i * incx]` with `y[ly + i * incy]`,

where

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

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

Input

<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>	single-precision complex vector <code>y</code> (unchanged from input if <code>n <= 0</code>)
<code>y</code>	single-precision complex vector <code>x</code> (unchanged from input if <code>n <= 0</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

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

Output

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

Reference: Analogous to <http://www.netlib.org/blas/icamax.f>

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

Error Status

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

Function `cublasScasum()`

float

cublasScasum (int n, const cuDouble *x, int incx)

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

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

where

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

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

Input

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

Output

returns the single-precision sum of absolute values of real and imaginary parts (returns zero if **n** <= 0, **incx** <= 0, or if an error occurred)

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

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

Error Status

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

Function `cublasScnrm2()`

float

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

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

Input

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

Output

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

Reference: <http://www.netlib.org/blas/scnrm2.f>

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

Error Status

<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

Double-Precision BLAS1 Functions

Note: Double-precision functions are only supported on GPUs with double-precision hardware.

The double-precision BLAS1 functions are as follows:

- ❑ “Function `cublasIdamax()`” on page 46
- ❑ “Function `cublasIdamin()`” on page 46
- ❑ “Function `cublasDasum()`” on page 47
- ❑ “Function `cublasDaxpy()`” on page 48
- ❑ “Function `cublasDcopy()`” on page 49
- ❑ “Function `cublasDdot()`” on page 50
- ❑ “Function `cublasDnrm2()`” on page 51
- ❑ “Function `cublasDrot()`” on page 52
- ❑ “Function `cublasDrotg()`” on page 53
- ❑ “Function `cublasDrotm()`” on page 54
- ❑ “Function `cublasDrotmg()`” on page 55
- ❑ “Function `cublasDscal()`” on page 56
- ❑ “Function `cublasDswap()`” on page 57

Function `cublasIdamax()`

```
int
cublasIdamax (int n, const double *x, int incx)
```

finds the smallest index of the maximum magnitude element of double-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>	double-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/idamax.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_ARCH_MISMATCH</code>	if function invoked on device that does not support double precision
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	if function failed to launch on GPU

Function `cublasIdamin()`

```
int
cublasIdamin (int n, const double *x, int incx)
```

finds the smallest index of the minimum magnitude element of double-precision vector `x`; that is, the result is the first `i`, `i = 0` to `n-1`, that minimizes `abs(x[1 + i * incx])`. The result reflects 1-based indexing for compatibility with Fortran.

Input

n	number of elements in input vector
x	double-precision vector with n elements
incx	storage spacing between elements of x

Output

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

Analogous to <http://www.netlib.org/blas/idamax.f>

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

Error Status

CUBLAS_STATUS_NOT_INITIALIZED	if CUBLAS library was not initialized
CUBLAS_STATUS_ALLOC_FAILED	if function could not allocate reduction buffer
CUBLAS_STATUS_ARCH_MISMATCH	if function invoked on device that does not support double precision
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function cublasDasum()

double

cublasDasum (int n, const double *x, int incx)

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

Input

n	number of elements in input vector
x	double-precision vector with n elements
incx	storage spacing between elements of x

Output

returns the double-precision sum of absolute values
(returns zero if **n** <= 0 or **incx** <= 0, or if an error occurred)

Reference: <http://www.netlib.org/blas/dasum.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_ARCH_MISMATCH</code>	if function invoked on device that does not support double precision
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	if function failed to launch on GPU

Function `cublasDaxpy()`

```
void
cublasDaxpy (int n, double alpha, const double *x,
             int incx, double *y, int incy)
```

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

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

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

where

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

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

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

Input

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

Output

<code>y</code>	double-precision result (unchanged if <code>n <= 0</code>)
----------------	----------------------------------------------------------------

Reference: <http://www.netlib.org/blas/daxpy.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_ARCH_MISMATCH</code>	if function invoked on device that does not support double precision
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	if function failed to launch on GPU

Function `cublasDcopy()`

```
void
cublasDcopy (int n, const double *x, int incx, double *y,
             int incy)
```

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

`x[1x + i * incx]` to `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>	double-precision vector with <code>n</code> elements
<code>incx</code>	storage spacing between elements of <code>x</code>
<code>y</code>	double-precision vector with <code>n</code> elements
<code>incy</code>	storage spacing between elements of <code>y</code>

Output

<code>y</code>	contains double-precision vector <code>x</code>
----------------	-------------------------------------------------

Reference: <http://www.netlib.org/blas/dcopy.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_ARCH_MISMATCH</code>	if function invoked on device that does not support double precision
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	if function failed to launch on GPU

Function cublasDdot()

```
double
cublasDdot (int n, const double *x, int incx,
            const double *y, int incy)
```

computes the dot product of two double-precision vectors. It returns the dot product of the double-precision vectors `x` and `y` if successful, and `0.0` 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

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

Output

returns double-precision dot product (returns zero if <code>n <= 0</code>)

Reference: <http://www.netlib.org/blas/ddot.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_ARCH_MISMATCH</code>	if function invoked on device that does not support double precision
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	if function failed to launch on GPU

Function `cublasDnrm2()`

`double`

`cublasDnrm2 (int n, const double *x, int incx)`

computes the Euclidean norm of the double-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>	double-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/dnrm2.f>

Reference: <http://www.netlib.org/slatec/lin/dnrm2.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_ARCH_MISMATCH</code>	if function invoked on device that does not support double precision
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	if function failed to launch on GPU

Function cublasDrot()

```
void
cublasDrot (int n, double *x, int incx, double *y,
            int incy, double dc, double ds)
```

multiplies a 2x2 matrix $\begin{bmatrix} dc & ds \\ -ds & dc \end{bmatrix}$ with the 2xn 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 >= 0, else
1x = 1 + (1 - n) * incx;
```

y is treated similarly using 1y and incy.

Input

n	number of elements in input vectors
x	double-precision vector with n elements
incx	storage spacing between elements of x
y	double-precision vector with n elements
incy	storage spacing between elements of y
dc	element of rotation matrix
ds	element of rotation matrix

Output

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

Reference: <http://www.netlib.org/blas/drot.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_ARCH_MISMATCH	if function invoked on device that does not support double precision
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function cublasDrotg()

```
void
cublasDrotg (double *host_da, double *host_db,
             double *host_dc, double *host_ds)
```

constructs the Givens transformation

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

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

The quantity $r = \pm\sqrt{da^2 + db^2}$ overwrites da in storage. The value of db is overwritten by a value z which allows dc and ds to be recovered by the following algorithm:

```
if z = 1           set dc = 0.0 and ds = 1.0.
if abs(z) < 1      set dc =  $\sqrt{1 - z^2}$  and ds = z.
if abs(z) > 1      set dc = 1/z and ds =  $\sqrt{1 - dc^2}$ .
```

The function **cublasDrot**($n, x, incx, y, incy, dc, ds$) normally is called next to apply the transformation to a $2 \times n$ matrix. Note that this function is provided for completeness and is run exclusively on the host.

Input

da	double-precision scalar
db	double-precision scalar

Output

da	double-precision r
db	double-precision z
dc	double-precision result
ds	double-precision result

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

This function does not set any error status.

Function cublasDrotm()

```
void
cublasDrotm (int n, double *x, int incx, double *y,
             int incy, const double *dparam)
```

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 $dparam[0] = dflag$, h has one of the following forms:

$$dflag = -1.0$$
$$h = \begin{bmatrix} dh00 & dh01 \\ dh10 & dh11 \end{bmatrix}$$

$$dflag = 0.0$$
$$h = \begin{bmatrix} 1.0 & dh01 \\ dh10 & 1.0 \end{bmatrix}$$

$$dflag = 1.0$$
$$h = \begin{bmatrix} dh00 & 1.0 \\ -1.0 & dh11 \end{bmatrix}$$

$$dflag = -2.0$$
$$h = \begin{bmatrix} 1.0 & 0.0 \\ 0.0 & 1.0 \end{bmatrix}$$

Input

n	number of elements in input vectors.
x	double-precision vector with n elements.
incx	storage spacing between elements of x.
y	double-precision vector with n elements.
incy	storage spacing between elements of y.
dparam	5-element vector. dparam[0] is dflag described above. dparam[1] through dparam[4] contain the 2x2 rotation matrix h: dparam[1] contains dh00, dparam[2] contains dh10, dparam[3] contains dh01, and dparam[4] contains dh11.

Output

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

Reference: <http://www.netlib.org/blas/drotm.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_ARCH_MISMATCH</code>	if function invoked on device that does not support double precision
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	if function failed to launch on GPU

Function cublasDrotmg()

```
void
cublasDrotmg (double *host_dd1, double *host_dd2,
              double *host_dx1, const double *host_dy1,
              double *host_dparam)
```

constructs the modified Givens transformation matrix h which zeros the second component of the 2-vector $(\sqrt{dd1} * dx1, \sqrt{dd2} * dy1)^T$.

With `dparam[0] = dflag`, h has one of the following forms:

$$\begin{aligned} dflag &= -1.0 \\ h &= \begin{bmatrix} dh00 & dh01 \\ dh10 & dh11 \end{bmatrix} \end{aligned}$$

$$\begin{aligned} dflag &= 0.0 \\ h &= \begin{bmatrix} 1.0 & dh01 \\ dh10 & 1.0 \end{bmatrix} \end{aligned}$$

$$\begin{aligned} dflag &= 1.0 \\ h &= \begin{bmatrix} dh00 & 1.0 \\ -1.0 & dh11 \end{bmatrix} \end{aligned}$$

$$\begin{aligned} dflag &= -2.0 \\ h &= \begin{bmatrix} 1.0 & 0.0 \\ 0.0 & 1.0 \end{bmatrix} \end{aligned}$$

`dparam[1]` through `dparam[4]` contain `dh00`, `dh10`, `dh01`, and `dh11`, respectively. Values of 1.0, -1.0, or 0.0 implied by the value of `dflag` are not stored in `dparam`. Note that this function is provided for completeness and is run exclusively on the host.

Input

<code>dd1</code>	double-precision scalar
<code>dd2</code>	double-precision scalar
<code>dx1</code>	double-precision scalar
<code>dy1</code>	double-precision scalar

Output

dd1	changed to represent the effect of the transformation
dd2	changed to represent the effect of the transformation
dx1	changed to represent the effect of the transformation
dparam	5-element vector. dparam[0] is dflag described above. dparam[1] through dparam[4] contain the 2×2 rotation matrix h: dparam[1] contains dh00, dparam[2] contains dh10, dparam[3] contains dh01, and dparam[4] contains dh11.

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

This function does not set any error status.

Function cublasDscal()

void

cublasDscal (int n, double alpha, double *x, int incx)

replaces double-precision vector x with double-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 $incx \geq 0$, else

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

Input

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

Output

x	double-precision result (unchanged if $n \leq 0$ or $incx \leq 0$)
---	---------------------------------------------------------------------

Reference: <http://www.netlib.org/blas/dscal.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_ARCH_MISMATCH	if function invoked on device that does not support double precision
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function `cublasDswap()`

```
void
cublasDswap (int n, double *x, int incx, double *y,
             int incy)
```

interchanges double-precision vector x with double-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	double-precision vector with n elements
$incx$	storage spacing between elements of x
y	double-precision vector with n elements
$incy$	storage spacing between elements of y

Output

x	double-precision vector y (unchanged from input if $n \leq 0$)
y	double-precision vector x (unchanged from input if $n \leq 0$)

Reference: <http://www.netlib.org/blas/dswap.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_ARCH_MISMATCH</code>	if function invoked on device that does not support double precision
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	if function failed to launch on GPU

Double-Precision Complex BLAS1 functions

Note: Double-precision functions are only supported on GPUs with double-precision hardware.

The double-precision complex BLAS1 functions are listed below:

- ❑ “Function `cublasDzasum()`” on page 60
- ❑ “Function `cublasDznrm2()`” on page 61
- ❑ “Function `cublasIzamax()`” on page 61
- ❑ “Function `cublasIzamin()`” on page 62
- ❑ “Function `cublasZaxpy()`” on page 63
- ❑ “Function `cublasZcopy()`” on page 64
- ❑ “Function `cublasZdotc()`” on page 65
- ❑ “Function `cublasZdotu()`” on page 66
- ❑ “Function `cublasZdrot()`” on page 67
- ❑ “Function `cublasZdscal()`” on page 68
- ❑ “Function `cublasZrot()`” on page 69
- ❑ “Function `cublasZrotg()`” on page 70
- ❑ “Function `cublasZscal()`” on page 70
- ❑ “Function `cublasZswap()`” on page 71

Function `cublasDzasum()`

double

`cublasDzasum (int n, const cuDoubleComplex *x, int incx)`

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

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

where

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

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

Input

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

Output

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

Reference <http://www.netlib.org/blas/dzasum.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_ARCH_MISMATCH</code>	if function invoked on device that does not support double precision
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	if function failed to launch on GPU

Function `cublasDznrm2()`

double

cublasDznrm2 (int n, const cuDoubleComplex *x, int incx)

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

Input

n	number of elements in input vector
x	double-precision complex vector with n elements
$incx$	storage spacing between elements of x

Output

returns the Euclidian norm
(returns zero if $n \leq 0$, $incx \leq 0$, or if an error occurred)

Reference: <http://www.netlib.org/blas/dznrm2.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_ARCH_MISMATCH	if function invoked on device that does not support double precision
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function `cublasIzamax()`

int

cublasIzamax (int n, const cuDoubleComplex *x, int incx)

finds the smallest index of the maximum magnitude element of double-precision complex vector x ; that is, the result is the first i , $i = 0$ to $n-1$, that maximizes

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

Input

n	number of elements in input vector
x	double-precision complex vector with n elements
$incx$	storage spacing between elements of x

Output

returns the smallest index (returns zero if $n \leq 0$ or $incx \leq 0$)

Reference: <http://www.netlib.org/blas/izamax.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_ARCH_MISMATCH	if function invoked on device that does not support double precision
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function `cublasIzamin()`

int

`cublasIzamin (int n, const cuDoubleComplex *x, int incx)`

finds the smallest index of the minimum magnitude element of double-precision complex vector x ; that is, the result is the first i , $i = 0$ to $n-1$, that minimizes

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

Input

n	number of elements in input vector
x	double-precision complex vector with n elements
$incx$	storage spacing between elements of x

Output

returns the smallest index (returns zero if $n \leq 0$ or $incx \leq 0$)

Reference: analogous to “[Function `cublasIzamax\(\)`](#)” on page 61.

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

Error Status

CUBLAS_STATUS_NOT_INITIALIZED	if CUBLAS library was not initialized
CUBLAS_STATUS_ARCH_MISMATCH	if function invoked on device that does not support double precision
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function cublasZaxpy()

```
void
cublasZaxpy (int n, cuDoubleComplex alpha,
             const cuDoubleComplex *x, int incx,
             cuDoubleComplex *y, int incy)
```

multiplies double-precision complex vector *x* by double-precision complex scalar *alpha* and adds the result to double-precision complex vector *y*; that is, it overwrites double-precision complex *y* with double-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 $incx \geq 0$, else
 $lx = 1 + (1 - n) * incx$;

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

Input

<i>n</i>	number of elements in input vectors
<i>alpha</i>	double-precision complex scalar multiplier
<i>x</i>	double-precision complex vector with <i>n</i> elements
<i>incx</i>	storage spacing between elements of <i>x</i>
<i>y</i>	double-precision complex vector with <i>n</i> elements
<i>incy</i>	storage spacing between elements of <i>y</i>

Output

<i>y</i>	double-precision complex result (unchanged if <i>n</i> <= 0)
----------	--------------------------------------------------------------

Reference: <http://www.netlib.org/blas/zaxpy.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
<code>CUBLAS_STATUS_ARCH_MISMATCH</code>	if function invoked on device that does not support double precision

Function cublasZcopy()

```
void
cublasZcopy (int n, const cuDoubleComplex *x, int incx,
             cuDoubleComplex *y, int incy)
```

copies the double-precision complex vector *x* to the double-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

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

Output

<i>y</i>	contains double-precision complex vector <i>x</i>
----------	---------------------------------------------------

Reference: <http://www.netlib.org/blas/zcopy.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
<code>CUBLAS_STATUS_ARCH_MISMATCH</code>	if function invoked on device that does not support double precision

Function cublasZdotc()

```
cuDoubleComplex
cublasZdotc (int n, const cuDoubleComplex *x, int incx,
             const cuDoubleComplex *y, int incy)
```

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

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

where

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

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

Input

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

Output

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

Reference: <http://www.netlib.org/blas/zdotc.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_ARCH_MISMATCH</code>	if function invoked on device that does not support double precision
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	if function failed to launch on GPU

Function cublasZdotu()

```
cuDoubleComplex
cublasZdotu (int n, const cuDoubleComplex *x, int incx,
             const cuDoubleComplex *y, int incy)
```

computes the dot product of two double-precision complex vectors. It returns the dot product of the double-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

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

Output

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

Reference: <http://www.netlib.org/blas/zdotu.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_ARCH_MISMATCH</code>	if function invoked on device that does not support double precision
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	if function failed to launch on GPU

Function cublasZdrot()

```
void
cublasZdrot (int n, cuDoubleComplex *x, int incx,
             cuDoubleComplex *y, int incy,
             double c, double s)
```

multiplies a 2x2 matrix $\begin{bmatrix} c & s \\ -s & c \end{bmatrix}$ with the 2xn matrix $\begin{bmatrix} x^T \\ y^T \end{bmatrix}$.

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

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

y is treated similarly using ly and incy.

Input

n	number of elements in input vectors
x	double-precision complex vector with n elements
incx	storage spacing between elements of x
y	double-precision complex vector with n elements
incy	storage spacing between elements of y
c	double-precision cosine component of rotation matrix
s	double-precision sine component of rotation matrix

Output

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

Reference: <http://www.netlib.org/blas/zdrot.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_ARCH_MISMATCH	if function invoked on device that does not support double precision
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function cublasZdscal()

```
void
cublasZdscal (int n, double alpha, cuDoubleComplex *x,
              int incx)
```

replaces double-precision complex vector x with double-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
alpha	double-precision scalar multiplier
x	double-precision complex vector with n elements
incx	storage spacing between elements of x

Output

x	double-precision complex result (unchanged if $n \leq 0$ or $incx \leq 0$)
---	-----------------------------------------------------------------------------

Reference: <http://www.netlib.org/blas/zdscal.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_ARCH_MISMATCH	if function invoked on device that does not support double precision
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function cublasZrot()

```
void
cublasZrot (int n, cuDoubleComplex *x, int incx,
            cuDoubleComplex *y, int incy, double sc,
            cuDoubleComplex cs)
```

multiplies a 2×2 matrix $\begin{bmatrix} sc & cs \\ -\overline{cs} & sc \end{bmatrix}$ with the 2×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 >= 0, else
1x = 1 + (1 - n) * incx;
```

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

Input

n	number of elements in input vectors
x	double-precision complex vector with n elements
$incx$	storage spacing between elements of x
y	double-precision complex vector with n elements
$incy$	storage spacing between elements of y
sc	double-precision cosine component of rotation matrix
cs	double-precision complex sine component of rotation matrix

Output

x	rotated double-precision complex vector x (unchanged if $n \leq 0$)
y	rotated double-precision complex vector y (unchanged if $n \leq 0$)

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

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

Error Status

CUBLAS_STATUS_NOT_INITIALIZED	if CUBLAS library was not initialized
CUBLAS_STATUS_ARCH_MISMATCH	if function invoked on device that does not support double precision
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function cublasZrotg()

```
void
cublasZrotg (cuDoubleComplex *host_ca,
             cuDoubleComplex *host_cb, double *host_sc,
             double *host_cs)
```

constructs the complex Givens transformation

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

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

The quantity $ca/|ca|*||ca, cb||$ overwrites ca in storage. The function **cublasCrot** ($n, x, incx, y, incy, sc, cs$) normally is called next to apply the transformation to a $2 \times n$ matrix. Note that this function is provided for completeness and is run exclusively on the host.

Input

ca	double-precision complex scalar
cb	double-precision complex scalar

Output

ca	double-precision complex $ca/ ca * ca, cb $
sc	double-precision cosine component of rotation matrix
cs	double-precision complex sine component of rotation matrix

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

This function does not set any error status.

Function cublasZscal()

```
void
cublasZscal (int n, cuDoubleComplex alpha,
             cuDoubleComplex *x, int incx)
```

replaces double-precision complex vector x with double-precision complex $\alpha * x$.

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

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

where

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

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

Input

n	number of elements in input vector
α	double-precision complex scalar multiplier
x	double-precision complex vector with n elements
$incx$	storage spacing between elements of x

Output

x	double-precision complex result (unchanged if $n \leq 0$ or $incx \leq 0$)
-----	-----------------------------------------------------------------------------

Reference: <http://www.netlib.org/blas/zscal.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
<code>CUBLAS_STATUS_ARCH_MISMATCH</code>	if function invoked on device that does not support double precision

Function `cublasZswap()`

```
void
cublasZswap (int n, cuDoubleComplex *x, int incx,
             cuDoubleComplex *y, int incy)
```

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

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

where

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

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

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

Input

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

Output

<code>x</code>	double-precision complex vector <code>y</code> (unchanged from input if <code>n <= 0</code>)
<code>y</code>	double-precision complex vector <code>x</code> (unchanged from input if <code>n <= 0</code>)

Reference: <http://www.netlib.org/blas/zswap.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
<code>CUBLAS_STATUS_ARCH_MISMATCH</code>	if function invoked on device that does not support double precision

Single-Precision BLAS2 Functions

The Level 2 Basic Linear Algebra Subprograms (BLAS2) are functions that perform matrix-vector operations. The CUBLAS implementations of single-precision BLAS2 functions are described in these sections:

- ❑ [“Single-Precision BLAS2 Functions” on page 74](#)
- ❑ [“Single-Precision Complex BLAS2 Functions” on page 97](#)

Single-Precision BLAS2 Functions

The single-precision BLAS2 functions are as follows:

- ❑ “Function `cublasSgbmv()`” on page 75
- ❑ “Function `cublasSgemv()`” on page 76
- ❑ “Function `cublasSger()`” on page 77
- ❑ “Function `cublasSsbmv()`” on page 78
- ❑ “Function `cublasSspmv()`” on page 80
- ❑ “Function `cublasSspr()`” on page 81
- ❑ “Function `cublasSspr2()`” on page 82
- ❑ “Function `cublasSsymv()`” on page 84
- ❑ “Function `cublasSsyr()`” on page 85
- ❑ “Function `cublasSsyr2()`” on page 86
- ❑ “Function `cublasStbmv()`” on page 88
- ❑ “Function `cublasStbsv()`” on page 89
- ❑ “Function `cublasStpmv()`” on page 91
- ❑ “Function `cublasStpsv()`” on page 92
- ❑ “Function `cublasStrmv()`” on page 94
- ❑ “Function `cublasStrsv()`” on page 95

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

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

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

α and β are single-precision scalars, and x and y are single-precision vectors. A is an $m \times n$ band matrix consisting of single-precision elements with kl subdiagonals and ku superdiagonals.

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>	the number of rows of matrix A ; m must be at least zero.
<code>n</code>	the number of columns of matrix A ; n must be at least zero.
<code>kl</code>	the number of subdiagonals of matrix A ; kl must be at least zero.
<code>ku</code>	the number of superdiagonals of matrix A ; ku 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) . The leading $(kl + ku + 1) \times n$ part of array A must contain the band matrix A , supplied column by column, with the leading diagonal of the matrix in row $ku+1$ of the array, the first superdiagonal starting at position 2 in row ku , the first subdiagonal starting at position 1 in row $ku+2$, and so on. Elements in the array A that do not correspond to elements in the band matrix (such as the top left $ku \times ku$ triangle) are not referenced.
<code>lda</code>	leading dimension of A ; <code>lda</code> must be at least $kl + ku + 1$.
<code>x</code>	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.
<code>incx</code>	storage spacing between 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 $(1 + (m - 1) * \text{abs}(\text{incy}))$ when <code>trans == 'N'</code> or <code>'n'</code> and at least $(1 + (n - 1) * \text{abs}(\text{incy}))$ otherwise. 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 $y = \alpha * \text{op}(A) * x + \beta * y$.
----------------	--------------------------------------------------------------------

Reference: <http://www.netlib.org/blas/sghbm.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 $m < 0$, $n < 0$, $kl < 0$, $ku < 0$, $incx == 0$, 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

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

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

`alpha` and `beta` are single-precision scalars, and `x` and `y` are single-precision vectors. `A` is an $m \times n$ 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'</code> or <code>'n'</code> , $\text{op}(A) = A$. If <code>trans == 'T'</code> , <code>'t'</code> , <code>'C'</code> , or <code>'c'</code> , $\text{op}(A) = A^T$.
<code>m</code>	specifies the number of rows of 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.

Input (continued)

<code>alpha</code>	single-precision scalar multiplier applied to <code>op(A)</code> .
<code>A</code>	single-precision array of dimensions <code>(lda, n)</code> if <code>trans == 'N'</code> or <code>'n'</code> , of dimensions <code>(lda, m)</code> otherwise; <code>lda</code> must be at least <code>max(1, m)</code> if <code>trans == 'N'</code> or <code>'n'</code> and at least <code>max(1, n)</code> otherwise.
<code>lda</code>	leading dimension of two-dimensional array used to store matrix <code>A</code> .
<code>x</code>	single-precision array of length at least <code>(1 + (n - 1) * abs(incx))</code> if <code>trans == 'N'</code> or <code>'n'</code> , else at least <code>(1 + (m - 1) * abs(incx))</code> .
<code>incx</code>	specifies the storage spacing for 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 <code>(1 + (m - 1) * abs(incy))</code> if <code>trans == 'N'</code> or <code>'n'</code> , else at least <code>(1 + (n - 1) * abs(incy))</code> .
<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 <code>y = alpha * op(A) * x + beta * y</code> .
----------------	----------------------------------------------------------------------

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 <code>m < 0</code> , <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 `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 α is a single-precision scalar, x is an m -element single-precision vector, y is an n -element single-precision vector, and A is an $m \times n$ matrix consisting of single-precision elements. Matrix A is stored in column-major format, and lda is the leading dimension of the two-dimensional array used to store A .

Input

m	specifies the number of rows of the matrix A ; m must be at least zero.
n	specifies the number of columns of matrix A ; n must be at least zero.
α	single-precision scalar multiplier applied to $x * y^T$.
x	single-precision array of length at least $(1 + (m - 1) * \text{abs}(\text{incx}))$.
incx	the storage spacing between elements of x ; incx must not be zero.
y	single-precision array of length at least $(1 + (n - 1) * \text{abs}(\text{incy}))$.
incy	the storage spacing between elements of y ; incy must not be zero.
A	single-precision array of dimensions (lda, n) .
lda	leading dimension of two-dimensional array used to store matrix A .

Output

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

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

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

Error Status

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

Input

<code>uplo</code>	specifies whether the upper or lower triangular part of the symmetric band matrix <code>A</code> 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 <code>A</code> ; <code>n</code> must be at least zero.
<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 <code>A * x</code> .
<code>A</code>	single-precision array of dimensions <code>(lda, n)</code> . When <code>uplo == 'U'</code> or <code>'u'</code> , the leading <code>(k+1)×n</code> 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 <code>k×k</code> triangle of the array <code>A</code> is not referenced. When <code>uplo == 'L'</code> or <code>'l'</code> , the leading <code>(k+1)×n</code> 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 <code>k×k</code> 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 <code>(1 + (n - 1) * abs(incx))</code> .
<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 <code>(1 + (n - 1) * abs(incy))</code> . 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 <code>y = alpha * A * x + beta * y</code> .
----------------	------------------------------------------------------------------

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

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

Error Status

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

Function `cublasSspmv()`

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

performs the matrix-vector operation

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

where α and β 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 A is supplied in <code>AP</code> . If <code>uplo == 'L'</code> or <code>'l'</code> , the lower triangular part of A is supplied in <code>AP</code> .
<code>n</code>	the number of rows and columns of matrix A ; n must be at least zero.
<code>alpha</code>	single-precision scalar multiplier applied to $A * x$.
<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 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'</code> or <code>'l'</code> , the 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 $(1 + (n - 1) * \text{abs}(incx))$.
<code>incx</code>	storage spacing between elements of x ; <code>incx</code> must not be zero.

Input (continued)

<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 $n < 0$, <code>incx</code> == 0, or <code>incy</code> == 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</code> == 'U' or 'u', the upper triangular part of <code>A</code> is supplied in <code>AP</code> . If <code>uplo</code> == 'L' or 'l', 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 $(1 + (n - 1) * \text{abs}(\text{incx}))$.

Input (continued)

<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 $i \leq j$, <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 $i \geq j$, <code>A[i, j]</code> is stored in <code>AP[i + ((2 * n - j + 1) * j) / 2]</code> .

Output

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

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

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

Error Status

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

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

where `alpha` is a single-precision scalar, and `x` and `y` are `n`-element single-precision vectors. `A` is a symmetric $n \times n$ 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' or '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' or '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 $x * y^T + \text{alpha} * y * x^T$.
<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>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 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' or '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 $i \geq j$, <code>A[i, j]</code> is stored in <code>AP[i + ((2 * n - j + 1) * j) / 2]</code> .

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/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 $n < 0$, <code>incx == 0</code> , or <code>incy == 0</code>
<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 α and β 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 A is referenced. If <code>uplo == 'U' or 'u'</code> , the symmetric matrix A is stored in upper storage mode; that is, only the upper triangular part of A is referenced while the lower triangular part of A is inferred. If <code>uplo == 'L' or 'l'</code> , the symmetric matrix A is stored in lower storage mode; that is, only the lower triangular part of A is referenced while the upper triangular part of A is inferred.
<code>n</code>	specifies the number of rows and the number of columns of the symmetric matrix A ; n 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) . If <code>uplo == 'U' or 'u'</code> , the leading $n \times n$ 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 <code>uplo == 'L' or 'l'</code> , the leading $n \times n$ 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.
<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 $(1 + (n - 1) * \text{abs}(\text{incx}))$.
<code>incx</code>	storage spacing between elements of x ; <code>incx</code> must not be zero.
<code>beta</code>	single-precision scalar multiplier applied to vector y .

Input (continued)

<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/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 $n \times n$ 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

<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. If <code>uplo == 'L' or 'l'</code> , only the lower triangular part of <code>A</code> is referenced.
<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 $(1 + (n - 1) * \text{abs}(\text{incx}))$.

Input (continued)

<code>incx</code>	the storage spacing between elements of <code>x</code> ; <code>incx</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 the two-dimensional array containing <code>A</code> ; <code>lda</code> must be at least $\max(1, n)$.

Output

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

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

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

Error Status

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

Function `cublasSsyr2()`

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

performs the symmetric rank 2 operation

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

where `alpha` is a single-precision scalar, `x` and `y` are n -element single-precision vectors, and `A` is an $n \times n$ 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 $n < 0$, <code>incx == 0</code> , or <code>incy == 0</code>
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	if function failed to launch on GPU

Function 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.
<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.

Input (continued)

<code>lda</code>	is the leading dimension of <code>A</code> ; <code>lda</code> must be at least $k+1$.
<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 $n < 0$, $k < 0$, or <code>incx</code> == 0
<code>CUBLAS_STATUS_ALLOC_FAILED</code>	if function cannot allocate enough internal scratch vector memory
<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 $\text{op}(A)$. 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 <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 $\text{op}(A) * x = b$.
----------------	--------------------------------------------------------------------------------------------

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> or <code>n < 0</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.
<code>AP</code>	single-precision array with at least $(n * (n + 1)) / 2$ elements. If <code>uplo == 'U' or 'u'</code> , the array AP contains the upper triangular part of the symmetric matrix A , packed sequentially, column by column; that is, if $i \leq j$, $A[i, j]$ is stored in $AP[i + (j * (j + 1)) / 2]$. If <code>uplo == 'L' or 'l'</code> , array AP contains the lower triangular part of the symmetric matrix A , packed sequentially, column by column; that is, if $i \geq j$, $A[i, j]$ is stored in $AP[i + ((2 * n - j + 1) * j) / 2]$.

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/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> or <code>n < 0</code>
<code>CUBLAS_STATUS_ALLOC_FAILED</code>	if function cannot allocate enough internal scratch vector memory
<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.
<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 <code>i <= j</code> , <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 <code>i >= j</code> , <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 <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 <code>op(A) * x = b</code> .
----------------	------------------------------------------------------------------------------------------------

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> or <code>n < 0</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

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

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

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

Input

<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 $\text{op}(A)$. 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.
<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

x updated according to $x = \text{op}(A) * x$.

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

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

Error Status

CUBLAS_STATUS_NOT_INITIALIZED	if CUBLAS library was not initialized
CUBLAS_STATUS_INVALID_VALUE	if <code>incx == 0</code> or <code>n < 0</code>
CUBLAS_STATUS_ALLOC_FAILED	if function cannot allocate enough internal scratch vector memory
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function cublasStrsv()

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

solves a system of equations

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

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

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

uplo	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.
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> , $\text{op}(A) = A^T$.

Input (continued)

<code>diag</code>	specifies whether or not <code>A</code> is a unit triangular matrix. If <code>diag == 'U' or 'u'</code> , <code>A</code> is assumed to be unit triangular. 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.
<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 the two-dimensional array containing <code>A</code> ; <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, <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>	the 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/strsv.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> or <code>n < 0</code>
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	if function failed to launch on GPU

Single-Precision Complex BLAS2 Functions

The two single-precision complex BLAS2 functions are as follows:

- ❑ “Function `cublasCgbmv()`” on page 98
- ❑ “Function `cublasCgemv()`” on page 99
- ❑ “Function `cublasCgerc()`” on page 101
- ❑ “Function `cublasCgeru()`” on page 102
- ❑ “Function `cublasChbmv()`” on page 103
- ❑ “Function `cublasChemv()`” on page 105
- ❑ “Function `cublasCher()`” on page 106
- ❑ “Function `cublasCher2()`” on page 107
- ❑ “Function `cublasChpmv()`” on page 109
- ❑ “Function `cublasChpr()`” on page 110
- ❑ “Function `cublasChpr2()`” on page 111
- ❑ “Function `cublasCtbmv()`” on page 113
- ❑ “Function `cublasCtbsv()`” on page 115
- ❑ “Function `cublasCtpmv()`” on page 116
- ❑ “Function `cublasCtpsv()`” on page 118
- ❑ “Function `cublasCtrmv()`” on page 119
- ❑ “Function `cublasCtrsv()`” on page 121

Function cublasCgbmv()

```
void
cublasCgbmv (char trans, int m, int n, int kl, int ku,
             cuComplex alpha, const cuComplex *A,
             int lda, const cuComplex *x, int incx,
             cuComplex beta, cuComplex *y, int incy)
```

performs one of the matrix-vector operations

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

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

α and β are single-precision complex scalars, and x and y are single-precision complex vectors. A is an $m \times n$ band matrix consisting of single-precision complex elements with kl subdiagonals and ku superdiagonals.

Input

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

Input (continued)

<code>x</code>	single-precision complex array of length at least (1 + (n - 1) * abs(<code>incx</code>)) if <code>trans == 'N'</code> or <code>'n'</code> , else at least (1 + (m - 1) * abs(<code>incx</code>)).
<code>incx</code>	specifies the increment for the elements of <code>x</code> ; <code>incx</code> must not be zero.
<code>beta</code>	single-precision complex 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 complex array of length at least (1 + (m - 1) * abs(<code>incy</code>)) if <code>trans == 'N'</code> or <code>'n'</code> , else at least (1 + (n - 1) * abs(<code>incy</code>)). If <code>beta</code> is zero, <code>y</code> is not read.
<code>incy</code>	on entry, <code>incy</code> specifies the increment for the elements of <code>y</code> ; <code>incy</code> must not be zero.

Output

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

Reference: <http://www.netlib.org/blas/cgbmv.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>m < 0</code> , <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 `cublasCgemv()`

```
void
cublasCgemv (char trans, int m, int n,
             cuComplex alpha, const cuComplex *A,
             int lda, const cuComplex *x, int incx,
             cuComplex beta, cuComplex *y, int incy)
```

performs one of the matrix-vector operations

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

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

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

Input

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

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

Error Status

CUBLAS_STATUS_NOT_INITIALIZED	if CUBLAS library was not initialized
CUBLAS_STATUS_INVALID_VALUE	if $m < 0$, $n < 0$, $incx == 0$, or $incy == 0$
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function cublasCgerc()

```
void
cublasCgerc (int m, int n, cuComplex alpha,
             const cuComplex *x, int incx,
             const cuComplex *y, int incy,
             cuComplex *A, int lda)
```

performs the symmetric rank 1 operation

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

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

Input

m	specifies the number of rows of the matrix A ; m must be at least zero.
n	specifies the number of columns of matrix A ; n must be at least zero.
α	single-precision complex scalar multiplier applied to $x * y^H$.
x	single-precision complex array of length at least $(1 + (m - 1) * \text{abs}(incx))$.
$incx$	the storage spacing between elements of x ; $incx$ must not be zero.
y	single-precision complex array of length at least $(1 + (n - 1) * \text{abs}(incy))$.
$incy$	the storage spacing between elements of y ; $incy$ must not be zero.
A	single-precision complex array of dimensions (lda, n) .
lda	leading dimension of two-dimensional array used to store matrix A .

Output

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

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

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

Error Status

CUBLAS_STATUS_NOT_INITIALIZED	if CUBLAS library was not initialized
CUBLAS_STATUS_INVALID_VALUE	if $m < 0$, $n < 0$, $incx == 0$, or $incy == 0$
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function cublasCgeru()

void

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

performs the symmetric rank 1 operation

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

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

Input

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

Input (continued)

<code>incy</code>	the storage spacing between elements of y ; <code>incy</code> must not be zero.
<code>A</code>	single-precision complex 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/cgeru.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 `cublasChbmV()`

```
void
cublasChbmV (char uplo, int n, int k, cuComplex alpha,
             const cuComplex *A, int lda,
             const cuComplex *x, int incx,
             cuComplex beta, cuComplex *y, int incy)
```

performs the matrix-vector operation

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

where α and β are single-precision complex scalars, and x and y are n -element single-precision complex vectors. A is a Hermitian $n \times n$ band matrix that consists of single-precision complex elements, with k superdiagonals and the same number of subdiagonals.

Input

<code>uplo</code>	specifies whether the upper or lower triangular part of the Hermitian 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 Hermitian, this is also the number of subdiagonals; <code>k</code> must be at least zero.
<code>alpha</code>	single-precision complex scalar multiplier applied to <code>A * x</code> .
<code>A</code>	single-precision complex array of dimensions <code>(lda, n)</code> . If <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 Hermitian 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 array <code>A</code> is not referenced. When <code>uplo == 'L'</code> or <code>'l'</code> , the leading $(k + 1) \times n$ part of array <code>A</code> must contain the lower triangular band part of the Hermitian 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 array <code>A</code> is not referenced. The imaginary parts of the diagonal elements need not be set; they are assumed to be zero.
<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 complex 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 complex scalar multiplier applied to vector <code>y</code> .
<code>y</code>	single-precision complex 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/chbmv.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>k < 0</code> , <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 cublasChemv()

```
void
cublasChemv (char uplo, int n, cuComplex alpha,
             const cuComplex *A, int lda,
             const cuComplex *x, int incx,
             cuComplex beta, cuComplex *y, int incy)
```

performs the matrix-vector operation

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

where α and β are single-precision complex scalars, and x and y are n -element single-precision complex vectors. A is a Hermitian $n \times n$ matrix that consists of single-precision complex 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 A is referenced. If <code>uplo == 'U' or 'u'</code> , the Hermitian matrix A is stored in upper storage mode; that is, only the upper triangular part of A is referenced while the lower triangular part of A is inferred. If <code>uplo == 'L' or 'l'</code> , the Hermitian matrix A is stored in lower storage mode; that is, only the lower triangular part of A is referenced while the upper triangular part of A is inferred.
<code>n</code>	specifies the number of rows and the number of columns of the symmetric matrix A ; n must be at least zero.
<code>alpha</code>	single-precision complex scalar multiplier applied to $A * x$.
<code>A</code>	single-precision complex 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 part of the Hermitian 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 part of the Hermitian matrix, and the strictly upper triangular part of A is not referenced. The imaginary parts of the diagonal elements need not be set; they are assumed to be zero.
<code>lda</code>	leading dimension of A ; <code>lda</code> must be at least $\max(1, n)$.
<code>x</code>	single-precision complex array of length at least $(1 + (n - 1) * \text{abs}(\text{incx}))$.
<code>incx</code>	storage spacing between elements of x ; <code>incx</code> must not be zero.

Input (continued)

beta	single-precision complex scalar multiplier applied to vector <i>y</i> .
<i>y</i>	single-precision complex array of length at least $(1 + (n - 1) * \text{abs}(\text{incy}))$. If beta is zero, <i>y</i> is not read.
incy	storage spacing between elements of <i>y</i> ; incy must not be zero.

Output

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

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

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

Error Status

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

Function cublasCher()

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

performs the Hermitian rank 1 operation

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

where alpha is a single-precision scalar, x is an n-element single-precision complex vector, and A is an n×n Hermitian matrix consisting of single-precision complex 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.

Input (continued)

<code>alpha</code>	single-precision scalar multiplier applied to $x * x^H$.
<code>x</code>	single-precision complex array of length at least $(1 + (n - 1) * \text{abs}(\text{incx}))$.
<code>incx</code>	the storage spacing between elements of <code>x</code> ; <code>incx</code> must not be zero.
<code>A</code>	single-precision complex array of dimensions (lda, n) . If <code>uplo == 'U'</code> or <code>'u'</code> , <code>A</code> contains the upper triangular part of the Hermitian matrix, and the strictly lower triangular part is not referenced. If <code>uplo == 'L'</code> or <code>'l'</code> , <code>A</code> contains the lower triangular part of the Hermitian matrix, and the strictly upper triangular part is not referenced. The imaginary parts of the diagonal elements need not be set, they are assumed to be zero, and on exit they are set to zero.
<code>lda</code>	leading dimension of the two-dimensional array containing <code>A</code> ; <code>lda</code> must be at least $\max(1, n)$.

Output

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

Reference: <http://www.netlib.org/blas/cher.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 $\text{incx} == 0$
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	if function failed to launch on GPU

Function `cublasCher2()`

```
void
cublasCher2 (char uplo, int n, cuComplex alpha,
             const cuComplex *x, int incx,
             const cuComplex *y, int incy,
             cuComplex *A, int lda)
```

performs the Hermitian rank 2 operation

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

where α is a single-precision complex scalar, x and y are n -element single-precision complex vectors, and A is an $n \times n$ Hermitian matrix consisting of single-precision complex elements.

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'</code> or <code>'u'</code> , only the upper triangular part of A may be referenced and the lower triangular part of A is inferred. If <code>uplo == 'L'</code> or <code>'l'</code> , only the lower triangular part of A may be referenced and the upper triangular part of A is inferred.
<code>n</code>	the number of rows and columns of matrix A ; n must be at least zero.
<code>alpha</code>	single-precision complex scalar multiplier applied to $x * y^H$ and whose conjugate is applied to $y * x^H$.
<code>x</code>	single-precision array of length at least $(1 + (n - 1) * \text{abs}(\text{incx}))$.
<code>incx</code>	the storage spacing between elements of x ; <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>	the storage spacing between elements of y ; <code>incy</code> must not be zero.
<code>A</code>	single-precision complex array of dimensions (lda, n) . If <code>uplo == 'U'</code> or <code>'u'</code> , A contains the upper triangular part of the Hermitian matrix, and the strictly lower triangular part is not referenced. If <code>uplo == 'L'</code> or <code>'l'</code> , A contains the lower triangular part of the Hermitian matrix, and the strictly upper triangular part is not referenced. The imaginary parts of the diagonal elements need not be set, they are assumed to be zero, and on exit they are set to zero.
<code>lda</code>	leading dimension of A ; <code>lda</code> must be at least $\max(1, n)$.

Output

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

Reference: <http://www.netlib.org/blas/cher2.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 $n < 0$, <code>incx == 0</code> , or <code>incy == 0</code>
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function cublasChpmv()

```
void
cublasChpmv (char uplo, int n, cuComplex alpha,
             const cuComplex *AP, const cuComplex *x,
             int incx, cuComplex beta, cuComplex *y,
             int incy)
```

performs the matrix-vector operation

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

where α and β are single-precision complex scalars, and x and y are n -element single-precision complex vectors. A is a Hermitian $n \times n$ matrix that consists of single-precision complex 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' or 'u'</code> , the upper triangular part of A is supplied in <code>AP</code> . If <code>uplo == 'L' or 'l'</code> , the lower triangular part of A is supplied in <code>AP</code> .
<code>n</code>	the number of rows and columns of matrix A ; n must be at least zero.
<code>alpha</code>	single-precision complex scalar multiplier applied to $A * x$.
<code>AP</code>	single-precision complex array with at least $(n * (n + 1)) / 2$ elements. If <code>uplo == 'U' or 'u'</code> , array <code>AP</code> contains the upper triangular part of the Hermitian matrix A , packed sequentially, column by column; that is, if $i \leq j$, $A[i, j]$ is stored in <code>AP[i + (j * (j + 1)) / 2]</code> . If <code>uplo == 'L' or 'l'</code> , the array <code>AP</code> contains the lower triangular part of the Hermitian matrix A , packed sequentially, column by column; that is, if $i > j$, $A[i, j]$ is stored in <code>AP[i + ((2 * n - j + 1) * j) / 2]</code> . The imaginary parts of the diagonal elements need not be set; they are assumed to be zero.
<code>x</code>	single-precision complex array of length at least $(1 + (n - 1) * \text{abs}(\text{incx}))$.
<code>incx</code>	storage spacing between elements of x ; <code>incx</code> must not be zero.
<code>beta</code>	single-precision scalar multiplier applied to vector y .
<code>y</code>	single-precision array of length at least $(1 + (n - 1) * \text{abs}(\text{incy}))$. If β is zero, y is not read.
<code>incy</code>	storage spacing between elements of y ; <code>incy</code> must not be zero.

Output

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

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

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

Error Status

CUBLAS_STATUS_NOT_INITIALIZED	if CUBLAS library was not initialized
CUBLAS_STATUS_INVALID_VALUE	if $n < 0$, $incx == 0$, or $incy == 0$
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function `cublasChpr()`

```
void
cublasChpr (char uplo, int n, float alpha,
            const cuComplex *x, int incx, cuComplex *AP)
```

performs the Hermitian rank 1 operation

$$A = \alpha * x * x^H + A,$$

where α is a single-precision scalar, x is an n -element single-precision complex vector, and A is an $n \times n$ Hermitian matrix consisting of single-precision complex elements that 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' or 'u'</code> , the upper triangular part of <code>A</code> is supplied in <code>AP</code> . If <code>uplo == 'L' or '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 $x * x^H$.
<code>x</code>	single-precision complex array of length at least $(1 + (n - 1) * \text{abs}(incx))$.

Input (continued)

<code>incx</code>	the storage spacing between elements of <code>x</code> ; <code>incx</code> must not be zero.
<code>AP</code>	single-precision complex array with at least $(n * (n + 1)) / 2$ elements. If <code>uplo == 'U' or 'u'</code> , array <code>AP</code> contains the upper triangular part of the Hermitian 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> , the array <code>AP</code> contains the lower triangular part of the Hermitian 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> . The imaginary parts of the diagonal elements need not be set; they are assumed to be zero, and on exit they are set to zero.

Output

<code>A</code>	updated according to $A = \alpha * x * x^H + A$.
----------------	---------------------------------------------------

Reference: <http://www.netlib.org/blas/chpr.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 <code>incx == 0</code>
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	if function failed to launch on GPU

Function `cublasChpr2()`

```
void
cublasChpr2 (char uplo, int n, cuComplex alpha,
             const cuComplex *x, int incx,
             const cuComplex *y, int incy, cuComplex *AP)
```

performs the Hermitian rank 2 operation

$$A = \alpha * x * y^H + \overline{\alpha} * y * x^H + A,$$

where `alpha` is a single-precision complex scalar, `x` and `y` are n -element single-precision complex vectors, and `A` is an $n \times n$ Hermitian

matrix consisting of single-precision complex elements that 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' or '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' or '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 complex scalar multiplier applied to $x * y^H$ and whose conjugate is applied to $y * x^H$.
<code>x</code>	single-precision complex array of length at least $(1 + (n - 1) * \text{abs}(\text{incx}))$.
<code>incx</code>	the storage spacing between elements of <code>x</code> ; <code>incx</code> must not be zero.
<code>y</code>	single-precision complex array of length 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.
<code>AP</code>	single-precision complex array with at least $(n * (n + 1)) / 2$ elements. If <code>uplo == 'U' or 'u'</code> , array <code>AP</code> contains the upper triangular part of the Hermitian 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> , the array <code>AP</code> contains the lower triangular part of the Hermitian 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> . The imaginary parts of the diagonal elements need not be set; they are assumed to be zero, and on exit they are set to zero.

Output

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

Reference: <http://www.netlib.org/blas/chpr2.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 $n < 0$, <code>incx == 0</code> , or <code>incy == 0</code>
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	if function failed to launch on GPU

Function cublasCtbbmv()

```
void
cublasCtbbmv (char uplo, char trans, char diag, int n,
              int k, const cuComplex *A, int lda,
              cuComplex *x, int incx)
```

performs one of the matrix-vector operations

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

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

x is an n -element single-precision complex vector, and A is an $n \times n$, unit or non-unit, upper or lower, triangular band matrix consisting of single-precision complex 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' or 't'</code> , $\text{op}(A) = A^T$. If <code>trans == 'C' or 'c'</code> , $\text{op}(A) = A^H$.
<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.
<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.

Input (continued)

A	single-precision complex array of dimension (lda, n) . If <code>uplo == 'U'</code> or <code>'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'</code> or <code>'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.
lda	is the leading dimension of A; lda must be at least $k+1$.
x	single-precision complex 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.
incx	specifies the storage spacing for elements of x; incx must not be zero.

Output

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

Reference: <http://www.netlib.org/blas/ctbmvt.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>k < 0</code> , or <code>n < 0</code>
CUBLAS_STATUS_ALLOC_FAILED	if function cannot allocate enough internal scratch vector memory
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function cublasCtbsv()

```
void
cublasCtbsv (char uplo, char trans, char diag, int n,
             int k, const cuComplex *A, int lda,
             cuComplex *X, int incx)
```

solves one of the systems of equations

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

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

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> , 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' or 't'</code> , $\text{op}(A) = A^T$. If <code>trans == 'C', or 'c'</code> , $\text{op}(A) = A^H$.
<code>diag</code>	specifies whether A is unit triangular. If <code>diag == 'U' or 'u'</code> , A 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> , A is not assumed to be unit triangular.
<code>n</code>	the number of rows and columns of matrix A ; n must be at least zero.
<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 be at least zero.

Input (continued)

A	single-precision complex array of dimension (lda, n) . If <code>uplo == 'U'</code> or <code>'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'</code> or <code>'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 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.
x	single-precision complex array of length at least $(1 + (n - 1) * \text{abs}(\text{incx}))$.
incx	storage spacing between elements of x ; incx must not be zero.

Output

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

Reference: <http://www.netlib.org/blas/ctbsv.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> or <code>n < 0</code>
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function `cublasCtpmv()`

```
void
cublasCtpmv (char uplo, char trans, char diag, int n,
             const cuComplex *AP, cuComplex *x, int incx)
```

performs one of the matrix-vector operations

$$x = op(A) * x,$$

where $op(A) = A$, $op(A) = A^T$, or $op(A) = A^H$;

x is an n -element single-precision complex vector, and A is an $n \times n$, unit or non-unit, upper or lower, triangular matrix consisting of single-precision complex 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 $op(A)$. If <code>trans == 'N' or 'n'</code> , $op(A) = A$. If <code>trans == 'T' or 't'</code> , $op(A) = A^T$. If <code>trans == 'C' or 'c'</code> , $op(A) = A^H$.
<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.
<code>AP</code>	single-precision complex 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 complex array of length at least $(1 + (n - 1) * abs(incx))$. On entry, x contains the source vector. On exit, x is overwritten with the result vector.
<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 $x = op(A) * x$.
----------------	----------------------------------------

Reference: <http://www.netlib.org/blas/ctpmv.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> or <code>n < 0</code>

Error Status (continued)

CUBLAS_STATUS_ALLOC_FAILED	if function cannot allocate enough internal scratch vector memory
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function `cublasCtpsv()`

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

solves one of the systems of equations

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

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

b and x are n -element complex vectors, and A is an $n \times n$, unit or non-unit, upper or lower, triangular matrix.

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

Input

<code>uplo</code>	specifies whether the matrix 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' or 't'</code> , $\text{op}(A) = A^T$. If <code>trans == 'C' or 'c'</code> , $\text{op}(A) = A^H$.
<code>diag</code>	specifies whether A is unit triangular. If <code>diag == 'U' or 'u'</code> , A 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> , 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.

Input (continued)

<code>AP</code>	single-precision complex 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 complex 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.

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/ctpsv.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> or <code>n < 0</code>
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	if function failed to launch on GPU

Function `cublasCttrmv()`

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

performs one of the matrix-vector operations

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

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

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

Input

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

Output

<code>x</code>	updated according to <code>x = op(A) * x</code> .
----------------	---------------------------------------------------

Reference: <http://www.netlib.org/blas/ctrmv.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> or <code>n < 0</code>

Error Status (continued)

CUBLAS_STATUS_ALLOC_FAILED	if function cannot allocate enough internal scratch vector memory
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function cublasCtrsv()

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

solves a system of equations

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

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

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

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

Input

uplo	specifies whether the matrix data is stored in the upper or the lower triangular part of array A . If <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.
trans	specifies $\text{op}(A)$. If <code>trans == 'N' or 'n'</code> , $\text{op}(A) = A$. If <code>trans == 'T' or 't'</code> , $\text{op}(A) = A^T$. If <code>trans == 'C' or 'c'</code> , $\text{op}(A) = A^H$.
diag	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.
n	specifies the number of rows and columns of the matrix A ; n must be at least zero.

Input (continued)

A	single-precision complex array of dimensions (lda, n). If uplo == 'U' or 'u', A contains the upper triangular part of the symmetric matrix, and the strictly lower triangular part is not referenced. If uplo == 'L' or 'l', A contains the lower triangular part of the symmetric matrix, and the strictly upper triangular part is not referenced.
lda	leading dimension of the two-dimensional array containing A; lda must be at least max(1, n).
x	single-precision complex array of length at least (1 + (n - 1) * abs(incx)). On entry, x contains the n-element, right-hand-side vector b. On exit, it is overwritten with solution vector x.
incx	the storage spacing between elements of x; incx must not be zero.

Output

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

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

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

Error Status

CUBLAS_STATUS_NOT_INITIALIZED	if CUBLAS library was not initialized
CUBLAS_STATUS_INVALID_VALUE	if incx == 0 Or n < 0
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Double-Precision BLAS2 Functions

The Level 2 Basic Linear Algebra Subprograms (BLAS2) are functions that perform matrix-vector operations. The CUBLAS implementations of double-precision BLAS2 functions are described in these sections:

- ❑ “Double-Precision BLAS2 Functions” on page 124
- ❑ “Double-Precision Complex BLAS2 functions” on page 148

Double-Precision BLAS2 Functions

Note: Double-precision functions are only supported on GPUs with double-precision hardware.

The double-precision BLAS2 functions are as follows:

- ❑ “Function `cublasDgbmv()`” on page 125
- ❑ “Function `cublasDgemv()`” on page 126
- ❑ “Function `cublasDger()`” on page 128
- ❑ “Function `cublasDsbmv()`” on page 129
- ❑ “Function `cublasDspmv()`” on page 131
- ❑ “Function `cublasDspr()`” on page 132
- ❑ “Function `cublasDspr2()`” on page 133
- ❑ “Function `cublasDsymv()`” on page 134
- ❑ “Function `cublasDsyr()`” on page 136
- ❑ “Function `cublasDsyr2()`” on page 137
- ❑ “Function `cublasDtbbmv()`” on page 138
- ❑ “Function `cublasDtbsv()`” on page 140
- ❑ “Function `cublasDtpmv()`” on page 142
- ❑ “Function `cublasDtpsv()`” on page 143
- ❑ “Function `cublasDtrmv()`” on page 144
- ❑ “Function `cublasDtrsv()`” on page 146

Function cublasDgbmv()

```
void
cublasDgbmv (char trans, int m, int n, int kl, int ku,
             double alpha, const double *A, int lda,
             const double *x, int incx, double beta,
             double *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 double-precision scalars, and x and y are double-precision vectors. A is an $m \times n$ band matrix consisting of double-precision elements with kl subdiagonals and ku superdiagonals.

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>kl</code>	specifies the number of subdiagonals of matrix A ; kl must be at least zero.
<code>ku</code>	specifies the number of superdiagonals of matrix A ; ku must be at least zero.
<code>alpha</code>	double-precision scalar multiplier applied to $\text{op}(A)$.
<code>A</code>	double-precision array of dimensions (lda, n) . The leading $(kl+ku+1) \times n$ part of the array A must contain the band matrix A , supplied column by column, with the leading diagonal of the matrix in row $(ku+1)$ of the array, the first superdiagonal starting at position 2 in row ku , the first subdiagonal starting at position 1 in row $(ku+2)$, and so on. Elements in the array A that do not correspond to elements in the band matrix (such as the top left $ku \times ku$ triangle) are not referenced.
<code>lda</code>	leading dimension A ; lda must be at least $(kl+ku+1)$.
<code>x</code>	double-precision array of length at least $(1 + (n-1) * \text{abs}(\text{incx}))$ if <code>trans == 'N' or 'n'</code> , else at least $(1 + (m-1) * \text{abs}(\text{incx}))$.
<code>incx</code>	specifies the increment for the elements of x ; <code>incx</code> must not be zero.

Input (continued)

<code>beta</code>	double-precision scalar multiplier applied to vector y . If <code>beta</code> is zero, y is not read.
y	double-precision array of length at least $(1 + (m - 1) * \text{abs}(\text{incy}))$ if <code>trans == 'N' or 'n'</code> , else at least $(1 + (n - 1) * \text{abs}(\text{incy}))$. If <code>beta</code> is zero, y is not read.
<code>incy</code>	on entry, <code>incy</code> specifies the increment for the elements of y ; <code>incy</code> 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/dgbmv.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_ARCH_MISMATCH</code>	if function invoked on device that does not support double precision
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	if function failed to launch on GPU

Function `cublasDgemv()`

```
void
cublasDgemv (char trans, int m, int n, double alpha,
             const double *A, int lda, const double *x,
             int incx, double beta, double *y, int incy)
```

performs one of the matrix-vector operations

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

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

`alpha` and `beta` are double-precision scalars, and x and y are double-precision vectors. A is an $m \times n$ matrix consisting of double-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 $op(A)$. If <code>trans == 'N' OR 'n'</code> , $op(A) = A$. If <code>trans == 'T', 't', 'C', OR 'c'</code> , $op(A) = A^T$.
<code>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>	double-precision scalar multiplier applied to $op(A)$.
<code>A</code>	double-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>	double-precision array of length at least $(1 + (n - 1) * \text{abs}(\text{incx}))$ if <code>trans == 'N' OR 'n'</code> , else at least $(1 + (m - 1) * \text{abs}(\text{incx}))$.
<code>incx</code>	specifies the storage spacing for elements of x ; <code>incx</code> must not be zero.
<code>beta</code>	double-precision scalar multiplier applied to vector y . If <code>beta</code> is zero, y is not read.
<code>y</code>	double-precision array of length at least $(1 + (m - 1) * \text{abs}(\text{incy}))$ if <code>trans == 'N' OR 'n'</code> , else at least $(1 + (n - 1) * \text{abs}(\text{incy}))$.
<code>incy</code>	the storage spacing between elements of y ; <code>incy</code> must not be zero.

Output

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

Reference: <http://www.netlib.org/blas/dgemv.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 $m < 0$, $n < 0$, <code>incx == 0</code> , OR <code>incy == 0</code>
CUBLAS_STATUS_ARCH_MISMATCH	if function invoked on device that does not support double precision
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function cublasDger()

```
void
cublasDger (int m, int n, double alpha, const double *x,
            int incx, const double *y, int incy,
            double *A, int lda)
```

performs the symmetric rank 1 operation

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

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

Input

m	specifies the number of rows of the matrix A ; m must be at least zero.
n	specifies the number of columns of matrix A ; n must be at least zero.
α	double-precision scalar multiplier applied to $x * y^T$.
x	double-precision array of length at least $(1 + (m - 1) * \text{abs}(\text{incx}))$.
incx	the storage spacing between elements of x ; incx must not be zero.
y	double-precision array of length at least $(1 + (n - 1) * \text{abs}(\text{incy}))$.
incy	the storage spacing between elements of y ; incy must not be zero.
A	double-precision array of dimensions (lda, n) .
lda	leading dimension of two-dimensional array used to store matrix A .

Output

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

Reference: <http://www.netlib.org/blas/dger.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 $m < 0$, $n < 0$, $\text{incx} == 0$, or $\text{incy} == 0$

Error Status (continued)

CUBLAS_STATUS_ARCH_MISMATCH	if function invoked on device that does not support double precision
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function `cublasDsbmv()`

```
void
cublasDsbmv (char uplo, int n, int k, double alpha,
             const double *A, int lda, const double *x,
             int incx, double beta, double *y, int incy)
```

performs the matrix-vector operation

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

where `alpha` and `beta` are double-precision scalars, and `x` and `y` are `n`-element double-precision vectors. `A` is an `n`×`n` symmetric band matrix consisting of double-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 <code>A</code> 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 <code>A</code> ; <code>n</code> must be at least zero.
<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>	double-precision scalar multiplier applied to <code>A * x</code> .

Input (continued)

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

Output

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

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

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

Error Status

CUBLAS_STATUS_NOT_INITIALIZED	if CUBLAS library was not initialized
CUBLAS_STATUS_INVALID_VALUE	if $k < 0$, $n < 0$, $\text{incx} == 0$, OR $\text{incy} == 0$
CUBLAS_STATUS_ARCH_MISMATCH	if function invoked on device that does not support double precision
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function cublasDspmv()

```
void
cublasDspmv (char uplo, int n, double alpha,
             const double *AP, const double *x, int incx,
             double beta, double *y, int incy)
```

performs the matrix-vector operation

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

where `alpha` and `beta` are double-precision scalars, and `x` and `y` are `n`-element double-precision vectors. `A` is a symmetric `n`×`n` matrix that consists of double-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>	double-precision scalar multiplier applied to <code>A * x</code> .
<code>AP</code>	double-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 <code>i <= j</code> , <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 <code>i >= j</code> , <code>A[i, j]</code> is stored in <code>AP[i + ((2 * n - j + 1) * j) / 2]</code> .
<code>x</code>	double-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>	double-precision scalar multiplier applied to vector <code>y</code> .
<code>y</code>	double-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 <code>y = alpha * A * x + beta * y</code> .
----------------	------------------------------------------------------------------

Reference: <http://www.netlib.org/blas/dspmv.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_ARCH_MISMATCH</code>	if function invoked on device that does not support double precision
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	if function failed to launch on GPU

Function `cublasDspr()`

```
void
cublasDspr (char uplo, int n, double alpha,
            const double *x, int incx, double *AP)
```

performs the symmetric rank 1 operation

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

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

Output

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

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

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

Error Status

CUBLAS_STATUS_NOT_INITIALIZED	if CUBLAS library was not initialized
CUBLAS_STATUS_INVALID_VALUE	if $n < 0$ or $incx == 0$
CUBLAS_STATUS_ARCH_MISMATCH	if function invoked on device that does not support double precision
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function cublasDspr2()

```
void
cublasDspr2 (char uplo, int n, double alpha,
             const double *x, int incx, const double *y,
             int incy, double *AP)
```

performs the symmetric rank 2 operation

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

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

Input

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

Input (continued)

<code>incy</code>	storage spacing between elements of y ; <code>incy</code> must not be zero.
<code>AP</code>	double-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 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'</code> or <code>'l'</code> , the 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 $AP[i + ((2 * n - j + 1) * j) / 2]$.

Output

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

Reference: <http://www.netlib.org/blas/dspr2.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 $n < 0$, <code>incx == 0</code> , or <code>incy == 0</code>
<code>CUBLAS_STATUS_ARCH_MISMATCH</code>	if function invoked on device that does not support double precision
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	if function failed to launch on GPU

Function `cublasDsymv()`

```
void
cublasDsymv (char uplo, int n, double alpha,
             const double *A, int lda, const double *x,
             int incx, double beta, double *y, int incy)
```

performs the matrix-vector operation

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

where `alpha` and `beta` are double-precision scalars, and `x` and `y` are n -element double-precision vectors. `A` is a symmetric $n \times n$ matrix that

consists of double-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' or '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' or '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>	double-precision scalar multiplier applied to $A * x$.
<code>A</code>	double-precision array of dimensions (lda, n) . If <code>uplo == 'U' or 'u'</code> , the leading $n \times n$ upper triangular part of the array <code>A</code> must contain the upper triangular part of the symmetric matrix, and the strictly lower triangular part of <code>A</code> is not referenced. If <code>uplo == 'L' or 'l'</code> , the leading $n \times n$ lower triangular part of the array <code>A</code> must contain the lower triangular part of the symmetric matrix, and the strictly upper triangular part of <code>A</code> is not referenced.
<code>lda</code>	leading dimension of <code>A</code> ; <code>lda</code> must be at least $\max(1, n)$.
<code>x</code>	double-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>	double-precision scalar multiplier applied to vector <code>y</code> .
<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/dsymv.f>

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

Error Status

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

Error Status (continued)

<code>CUBLAS_STATUS_ARCH_MISMATCH</code>	if function invoked on device that does not support double precision
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	if function failed to launch on GPU

Function `cublasDsyrr()`

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

performs the symmetric rank 1 operation

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

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

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. If <code>uplo == 'L' or 'l'</code> , only the lower triangular part of <code>A</code> is referenced.
<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>	double-precision scalar multiplier applied to <code>x * x^T</code> .
<code>x</code>	double-precision array of length at least <code>(1 + (n - 1) * abs(incx))</code> .
<code>incx</code>	the storage spacing between elements of <code>x</code> ; <code>incx</code> must not be zero.
<code>A</code>	double-precision array of dimensions <code>(lda, n)</code> . 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 the two-dimensional array containing <code>A</code> ; <code>lda</code> must be at least <code>max(1, n)</code> .

Output

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

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

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

Error Status

CUBLAS_STATUS_NOT_INITIALIZED	if CUBLAS library was not initialized
CUBLAS_STATUS_INVALID_VALUE	if $n < 0$ or $incx == 0$
CUBLAS_STATUS_ARCH_MISMATCH	if function invoked on device that does not support double precision
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function cublasDsyr2()

```
void
cublasDsyr2 (char uplo, int n, double alpha,
             const double *x, int incx, const double *y,
             int incy, double *A, int lda)
```

performs the symmetric rank 2 operation

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

where α is a double-precision scalar, x and y are n -element double-precision vectors, and A is an $n \times n$ symmetric matrix consisting of double-precision elements.

Input

uplo	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 is referenced and the lower triangular part of A is inferred. If <code>uplo == 'L' or 'l'</code> , only the lower triangular part of A is referenced and the upper triangular part of A is inferred.
n	the number of rows and columns of matrix A ; n must be at least zero.
alpha	double-precision scalar multiplier applied to $x * y^T$ and $y * x^T$.
x	double-precision array of length at least $(1 + (n - 1) * \text{abs}(incx))$.
incx	storage spacing between elements of x ; $incx$ must not be zero.
y	double-precision array of length at least $(1 + (n - 1) * \text{abs}(incy))$.

Input (continued)

<code>incy</code>	storage spacing between elements of y ; <code>incy</code> must not be zero.
<code>A</code>	double-precision array of dimensions (lda, n) . If <code>uplo == 'U'</code> or <code>'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'</code> or <code>'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 A ; <code>lda</code> must be at least $\max(1, n)$.

Output

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

Reference: <http://www.netlib.org/blas/dsyr2.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 $n < 0$, <code>incx == 0</code> , or <code>incy == 0</code>
<code>CUBLAS_STATUS_ARCH_MISMATCH</code>	if function invoked on device that does not support double precision
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	if function failed to launch on GPU

Function `cublasDtbmv()`

```
void
cublasDtbmv (char uplo, char trans, char diag, int n,
             int k, const double *A, int lda, double *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 double-precision vector, and A is an $n \times n$, unit or non-unit, upper or lower, triangular band matrix consisting of double-precision elements.

Input

<code>uplo</code>	specifies whether the matrix A is an upper or lower triangular band matrix. If <code>uplo</code> == 'U' or 'u', A is an upper triangular band matrix. If <code>uplo</code> == 'L' or 'l', A is a lower triangular band matrix.
<code>trans</code>	specifies $\text{op}(A)$. If <code>trans</code> == 'N' or 'n', $\text{op}(A) = A$. If <code>trans</code> == 'T', 't', 'C', or 'c', $\text{op}(A) = A^T$.
<code>diag</code>	specifies whether or not matrix A is unit triangular. If <code>diag</code> == 'U' or 'u', A is assumed to be unit triangular. If <code>diag</code> == 'N' or 'n', 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.
<code>k</code>	specifies the number of superdiagonals or subdiagonals. If <code>uplo</code> == 'U' or 'u', k specifies the number of superdiagonals. If <code>uplo</code> == 'L' or 'l', k specifies the number of subdiagonals; k must at least be zero.
<code>A</code>	double-precision array of dimension (lda, n) . If <code>uplo</code> == 'U' or 'u', the leading $(k+1) \times n$ part of the array A must contain the upper triangular band matrix, supplied column by column, with the leading diagonal of the matrix in row $k+1$ of the array, the first superdiagonal starting at position 2 in row k , and so on. The top left $k \times k$ triangle of the array A is not referenced. If <code>uplo</code> == 'L' or 'l', the leading $(k+1) \times n$ part of the array A must contain the lower triangular band matrix, supplied column by column, with the leading diagonal of the matrix in row 1 of the array, the first subdiagonal starting at position 1 in row 2, and so on. The bottom right $k \times k$ triangle of the array is not referenced.
<code>lda</code>	is the leading dimension of A ; <code>lda</code> must be at least $k+1$.
<code>x</code>	double-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/dtbmv.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>k < 0</code> , or <code>n < 0</code>
CUBLAS_STATUS_ARCH_MISMATCH	if function invoked on device that does not support double precision
CUBLAS_STATUS_ALLOC_FAILED	if function cannot allocate enough internal scratch vector memory
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function `cublasDtbsv()`

```
void
cublasDtbsv (char uplo, char trans, char diag, int n,
             int k, const double *A, int lda, double *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`×`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> .
<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.

Input (continued)

<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>	double-precision array of dimension <code>(lda, n)</code> . If <code>uplo == 'U' or 'u'</code> , the leading <code>(k+1) × n</code> 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 <code>k × k</code> triangle of the array <code>A</code> is not referenced. If <code>uplo == 'L' or 'l'</code> , the leading <code>(k+1) × n</code> 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 <code>k × k</code> triangle of the array is not referenced.
<code>x</code>	double-precision array of length at least <code>(1 + (n - 1) * abs(incx))</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 <code>op(A) * x = b</code> .
----------------	------------------------------------------------------------------------------------------------

Reference: <http://www.netlib.org/blas/dtbsv.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> or <code>n < 0</code>
<code>CUBLAS_STATUS_ARCH_MISMATCH</code>	if function invoked on device that does not support double precision
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	if function failed to launch on GPU

Function cublasDtpmv()

```
void
cublasDtpmv (char uplo, char trans, char diag, int n,
             const double *AP, double *x, int incx)
```

performs one of the matrix-vector operations

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

x is an n -element double-precision vector, and A is an $n \times n$, unit or non-unit, upper or lower, triangular matrix consisting of double-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.
<code>AP</code>	double-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>	double-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/dtpmv.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> or <code>n < 0</code>
CUBLAS_STATUS_ALLOC_FAILED	if function cannot allocate enough internal scratch vector memory
CUBLAS_STATUS_ARCH_MISMATCH	if function invoked on device that does not support double precision
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function `cublasDtpsv()`

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

solves one of the systems of equations

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

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

b and x are n -element vectors, and A is an $n \times n$, unit or non-unit, upper or lower, triangular 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'</code> or <code>'u'</code> , A is an upper triangular matrix. If <code>uplo == 'L'</code> or <code>'l'</code> , A is a lower triangular matrix.
<code>trans</code>	specifies $\text{op}(A)$. If <code>trans == 'N'</code> or <code>'n'</code> , $\text{op}(A) = A$. If <code>trans == 'T'</code> , <code>'t'</code> , <code>'C'</code> , or <code>'c'</code> , $\text{op}(A) = A^T$.
<code>diag</code>	specifies whether A is unit triangular. If <code>diag == 'U'</code> or <code>'u'</code> , A is assumed to be unit triangular; that is, diagonal elements are not read and are assumed to be unity. If <code>diag == 'N'</code> or <code>'n'</code> , A is not assumed to be unit triangular.

Input (continued)

<code>n</code>	specifies the number of rows and columns of the matrix <code>A</code> ; <code>n</code> must be at least zero.
<code>AP</code>	double-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>	double-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.

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/dtpsv.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> or <code>n < 0</code>
<code>CUBLAS_STATUS_ARCH_MISMATCH</code>	if function invoked on device that does not support double precision
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	if function failed to launch on GPU

Function `cublasDtrmv()`

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

performs one of the matrix-vector operations

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

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

x is an n -element double-precision vector, and A is an $n \times n$, unit or non-unit, upper or lower, triangular matrix consisting of double-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.
<code>A</code>	double-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>	double-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/dtrmv.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> or <code>n < 0</code>

Error Status (continued)

CUBLAS_STATUS_ALLOC_FAILED	if function cannot allocate enough internal scratch vector memory
CUBLAS_STATUS_ARCH_MISMATCH	if function invoked on device that does not support double precision
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function cublasDtrsv()

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

solves a system of equations

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

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

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

Input

uplo	specifies whether the matrix data is stored in the upper or the lower triangular part of array A . If uplo == 'U' or 'u', only the upper triangular part of A may be referenced. If uplo == 'L' or 'l', only the lower triangular part of A may be referenced.
trans	specifies $\text{op}(A)$. If trans == 'N' or 'n', $\text{op}(A) = A$. If trans == 'T', 't', 'C', or 'c', $\text{op}(A) = A^T$.
diag	specifies whether or not A is a unit triangular matrix. If diag == 'U' or 'u', A is assumed to be unit triangular. If diag == 'N' or 'n', A is not assumed to be unit triangular.
n	specifies the number of rows and columns of the matrix A ; n must be at least zero.

Input (continued)

A	double-precision array of dimensions (lda, n). If uplo == 'U' or 'u', A contains the upper triangular part of the symmetric matrix, and the strictly lower triangular part is not referenced. If uplo == 'L' or 'l', A contains the lower triangular part of the symmetric matrix, and the strictly upper triangular part is not referenced.
lda	leading dimension of the two-dimensional array containing A; lda must be at least max(1, n).
x	double-precision array of length at least (1 + (n - 1) * abs(incx)). On entry, x contains the n-element, right-hand-side vector b. On exit, it is overwritten with the solution vector x.
incx	the storage spacing between elements of x; incx must not be zero.

Output

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

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

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

Error Status

CUBLAS_STATUS_NOT_INITIALIZED	if CUBLAS library was not initialized
CUBLAS_STATUS_INVALID_VALUE	if incx == 0 or n < 0
CUBLAS_STATUS_ARCH_MISMATCH	if function invoked on device that does not support double precision
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Double-Precision Complex BLAS2 functions

Note: Double-precision functions are only supported on GPUs with double-precision hardware.

Two double-precision complex BLAS2 functions are implemented:

- ❑ “Function `cublasZgbmv()`” on page 149
- ❑ “Function `cublasZgemv()`” on page 151
- ❑ “Function `cublasZgerc()`” on page 152
- ❑ “Function `cublasZgeru()`” on page 153
- ❑ “Function `cublasZhbmV()`” on page 155
- ❑ “Function `cublasZhemv()`” on page 157
- ❑ “Function `cublasZher()`” on page 158
- ❑ “Function `cublasZher2()`” on page 160
- ❑ “Function `cublasZhpmv()`” on page 161
- ❑ “Function `cublasZhpr()`” on page 163
- ❑ “Function `cublasZhpr2()`” on page 164
- ❑ “Function `cublasZtbmv()`” on page 165
- ❑ “Function `cublasZtbsv()`” on page 167
- ❑ “Function `cublasZtpmv()`” on page 169
- ❑ “Function `cublasZtpsv()`” on page 170
- ❑ “Function `cublasZtrmv()`” on page 172
- ❑ “Function `cublasZtrsv()`” on page 173

Function cublasZgbmv()

```
void
cublasZgbmv (char trans, int m, int n, int kl, int ku,
             cuDoubleComplex alpha,
             const cuDoubleComplex *A, int lda,
             const cuDoubleComplex *x, int incx,
             cuDoubleComplex beta, cuDoubleComplex *y,
             int incy)
```

performs one of the matrix-vector operations

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

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

α and β are double-precision complex scalars, and x and y are double-precision complex vectors. A is an $m \times n$ band matrix consisting of double-precision complex elements with kl subdiagonals and ku superdiagonals.

Input

trans	specifies $\text{op}(A)$. If $\text{trans} == \text{'N'}$ or 'n' , $\text{op}(A) = A$. If $\text{trans} == \text{'T'}$ or 't' , $\text{op}(A) = A^T$. If $\text{trans} == \text{'C'}$, or 'c' , $\text{op}(A) = A^H$.
m	specifies the number of rows of matrix A ; m must be at least zero.
n	specifies the number of columns of matrix A ; n must be at least zero.
kl	specifies the number of subdiagonals of matrix A ; kl must be at least zero.
ku	specifies the number of superdiagonals of matrix A ; ku must be at least zero.
alpha	double-precision complex scalar multiplier applied to $\text{op}(A)$.
A	double-precision complex array of dimensions (lda, n) . The leading $(kl+ku+1) \times n$ part of the array A must contain the band matrix A , supplied column by column, with the leading diagonal of the matrix in row $(ku+1)$ of the array, the first superdiagonal starting at position 2 in row ku , the first subdiagonal starting at position 1 in row $(ku+2)$, and so on. Elements in the array A that do not correspond to elements in the band matrix (such as the top left $ku \times ku$ triangle) are not referenced.
lda	leading dimension A ; lda must be at least $(kl+ku+1)$.

Input (continued)

<code>x</code>	double-precision complex array of length at least (1 + (n - 1) * abs(<code>incx</code>)) if <code>trans == 'N'</code> or <code>'n'</code> , else at least (1 + (m - 1) * abs(<code>incx</code>)).
<code>incx</code>	specifies the increment for the elements of <code>x</code> ; <code>incx</code> must not be zero.
<code>beta</code>	double-precision complex scalar multiplier applied to vector <code>y</code> . If <code>beta</code> is zero, <code>y</code> is not read.
<code>y</code>	double-precision complex array of length at least (1 + (m - 1) * abs(<code>incy</code>)) if <code>trans == 'N'</code> or <code>'n'</code> , else at least (1 + (n - 1) * abs(<code>incy</code>)). If <code>beta</code> is zero, <code>y</code> is not read.
<code>incy</code>	on entry, <code>incy</code> specifies the increment for the elements of <code>y</code> ; <code>incy</code> must not be zero.

Output

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

Reference: <http://www.netlib.org/blas/zgbmv.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>m < 0</code> , <code>n < 0</code> , <code>incx == 0</code> , or <code>incy == 0</code>
CUBLAS_STATUS_ARCH_MISMATCH	if function invoked on device that does not support double precision
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function cublasZgemv()

```
void
cublasZgemv (char trans, int m, int n,
             cuDoubleComplex alpha,
             const cuDoubleComplex *A, int lda,
             const cuDoubleComplex *x, int incx,
             cuDoubleComplex beta, cuDoubleComplex *y,
             int incy)
```

performs one of the matrix-vector operations

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

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

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

Input

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

Input (continued)

<code>y</code>	double-precision complex array of length at least (1 + (m - 1) * abs(incy)) if <code>trans == 'N'</code> or <code>'n'</code> , else at least (1 + (n - 1) * abs(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 $y = \alpha * \text{op}(A) * x + \beta * y$.
----------------	--------------------------------------------------------------------

Reference: <http://www.netlib.org/blas/zgemv.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>m < 0</code> , <code>n < 0</code> , <code>incx == 0</code> , or <code>incy == 0</code>
<code>CUBLAS_STATUS_ARCH_MISMATCH</code>	if function invoked on device that does not support double precision
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	if function failed to launch on GPU

Function `cublasZgerc()`

```
void
cublasZgerc (int m, int n, cuDoubleComplex alpha,
             const cuDoubleComplex *x, int incx,
             const cuDoubleComplex *y, int incy,
             cuDoubleComplex *A, int lda)
```

performs the symmetric rank 1 operation

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

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

Input

m	specifies the number of rows of the matrix A ; m must be at least zero.
n	specifies the number of columns of matrix A ; n must be at least zero.
alpha	double-precision complex scalar multiplier applied to $x * y^H$.
x	double-precision complex array of length at least $(1 + (m - 1) * \text{abs}(\text{incx}))$.
incx	the storage spacing between elements of x ; incx must not be zero.
y	double-precision complex array of length at least $(1 + (n - 1) * \text{abs}(\text{incy}))$.
incy	the storage spacing between elements of y ; incy must not be zero.
A	double-precision complex array of dimensions (lda, n) .
lda	leading dimension of two-dimensional array used to store matrix A .

Output

A	updated according to $A = \text{alpha} * x * y^H + A$.
----------	---------------------------------------------------------

Reference: <http://www.netlib.org/blas/zgerc.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 $m < 0$, $n < 0$, $\text{incx} == 0$, or $\text{incy} == 0$
CUBLAS_STATUS_ARCH_MISMATCH	if function invoked on device that does not support double precision
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function cublasZgeru()

```
void
cublasZgeru (int m, int n, cuDoubleComplex alpha,
             const cuDoubleComplex *x, int incx,
             const cuDoubleComplex *y, int incy,
             cuDoubleComplex *A, int lda)
```

performs the symmetric rank 1 operation

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

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

Input

m	specifies the number of rows of the matrix A ; m must be at least zero.
n	specifies the number of columns of matrix A ; n must be at least zero.
α	double-precision complex scalar multiplier applied to $x * y^T$.
x	double-precision complex array of length at least $(1 + (m - 1) * \text{abs}(\text{incx}))$.
incx	the storage spacing between elements of x ; incx must not be zero.
y	double-precision complex array of length at least $(1 + (n - 1) * \text{abs}(\text{incy}))$.
incy	the storage spacing between elements of y ; incy must not be zero.
A	double-precision complex array of dimensions (lda, n) .
lda	leading dimension of two-dimensional array used to store matrix A .

Output

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

Reference: <http://www.netlib.org/blas/zgeru.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 $m < 0$, $n < 0$, $\text{incx} == 0$, or $\text{incy} == 0$
CUBLAS_STATUS_ARCH_MISMATCH	if function invoked on device that does not support double precision
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function cublasZhbmv()

```
void
cublasZhbmv (char uplo, int n, int k,
             cuDoubleComplex alpha,
             const cuDoubleComplex *A, int lda,
             const cuDoubleComplex *x, int incx,
             cuDoubleComplex beta, cuDoubleComplex *y,
             int incy)
```

performs the matrix-vector operation

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

where α and β are double-precision complex scalars, and x and y are n -element double-precision complex vectors. A is a Hermitian $n \times n$ band matrix that consists of double-precision complex elements, with k superdiagonals and the same number of subdiagonals.

Input

<code>uplo</code>	specifies whether the upper or lower triangular part of the Hermitian band matrix A is being supplied. If <code>uplo == 'U' or 'u'</code> , the upper triangular part is being supplied. If <code>uplo == 'L' or '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 ; n must be at least zero.
<code>k</code>	specifies the number of superdiagonals of matrix A . Since the matrix is Hermitian, this is also the number of subdiagonals; k must be at least zero.
<code>alpha</code>	double-precision complex scalar multiplier applied to $A * x$.

Input (continued)

A	double-precision complex array of dimensions (lda, n) . If <code>uplo == 'U'</code> or <code>'u'</code> , the leading $(k + 1) \times n$ part of array A must contain the upper triangular band of the Hermitian 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 array A is not referenced. When <code>uplo == 'L'</code> or <code>'l'</code> , the leading $(k + 1) \times n$ part of array A must contain the lower triangular band part of the Hermitian 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 array A is not referenced. The imaginary parts of the diagonal elements need not be set; they are assumed to be zero.
lda	leading dimension of A ; lda must be at least $k + 1$.
x	double-precision complex array of length at least $(1 + (n - 1) * \text{abs}(\text{incx}))$.
incx	storage spacing between elements of x ; incx must not be zero.
beta	double-precision complex scalar multiplier applied to vector y .
y	double-precision complex array of length at least $(1 + (n - 1) * \text{abs}(\text{incy}))$. If beta is zero, y is not read.
incy	storage spacing between elements of y ; incy must not be zero.

Output

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

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

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

Error Status

CUBLAS_STATUS_NOT_INITIALIZED	if CUBLAS library was not initialized
CUBLAS_STATUS_INVALID_VALUE	if $k < 0$, $n < 0$, <code>incx == 0</code> , OR <code>incy == 0</code>
CUBLAS_STATUS_ARCH_MISMATCH	if function invoked on device that does not support double precision
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function cublasZhemv()

```
void
cublasZhemv (char uplo, int n, cuDoubleComplex alpha,
             const cuDoubleComplex *A, int lda,
             const cuDoubleComplex *x, int incx,
             cuDoubleComplex beta, cuDoubleComplex *y,
             int incy)
```

performs the matrix-vector operation

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

where `alpha` and `beta` are double-precision complex scalars, and `x` and `y` are `n`-element double-precision complex vectors. `A` is a Hermitian $n \times n$ matrix that consists of double-precision complex 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 Hermitian 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 Hermitian 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>	double-precision complex scalar multiplier applied to <code>A * x</code> .
<code>A</code>	double-precision complex array of dimensions (lda, n) . If <code>uplo == 'U'</code> or <code>'u'</code> , the leading $n \times n$ upper triangular part of the array <code>A</code> must contain the upper triangular part of the Hermitian matrix, and the strictly lower triangular part of <code>A</code> is not referenced. If <code>uplo == 'L'</code> or <code>'l'</code> , the leading $n \times n$ lower triangular part of the array <code>A</code> must contain the lower triangular part of the Hermitian matrix, and the strictly upper triangular part of <code>A</code> is not referenced. The imaginary parts of the diagonal elements need not be set; they are assumed to be zero.
<code>lda</code>	leading dimension of <code>A</code> ; <code>lda</code> must be at least $\max(1, n)$.
<code>x</code>	double-precision complex 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.

Input (continued)

<code>beta</code>	double-precision complex scalar multiplier applied to vector <code>y</code> .
<code>y</code>	double-precision complex 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/zhemv.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_ARCH_MISMATCH</code>	if function invoked on device that does not support double precision
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	if function failed to launch on GPU

Function cublasZher()

```
void
cublasZher (char uplo, int n, double alpha,
            const cuDoubleComplex *x, int incx,
            cuDoubleComplex *A, int lda)
```

performs the Hermitian rank 1 operation

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

where `alpha` is a double-precision scalar, `x` is an n -element double-precision complex vector, and `A` is an $n \times n$ Hermitian matrix consisting of double-precision complex elements. `A` is stored in column-major format, and `lda` is the leading dimension of the two-dimensional array containing `A`.

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. If <code>uplo == 'L' or 'l'</code> , only the lower triangular part of <code>A</code> is referenced.
<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>	double-precision scalar multiplier applied to $x * x^H$.
<code>x</code>	double-precision complex array of length at least $(1 + (n - 1) * \text{abs}(\text{incx}))$.
<code>incx</code>	the storage spacing between elements of <code>x</code> ; <code>incx</code> must not be zero.
<code>A</code>	double-precision complex array of dimensions (lda, n) . If <code>uplo == 'U' or 'u'</code> , <code>A</code> contains the upper triangular part of the Hermitian 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 Hermitian matrix, and the strictly upper triangular part is not referenced. The imaginary parts of the diagonal elements need not be set, they are assumed to be zero, and on exit they are set to zero.
<code>lda</code>	leading dimension of the two-dimensional array containing <code>A</code> ; <code>lda</code> must be at least $\max(1, n)$.

Output

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

Reference: <http://www.netlib.org/blas/zher.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> or <code>incx == 0</code>
<code>CUBLAS_STATUS_ARCH_MISMATCH</code>	if function invoked on device that does not support double precision
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	if function failed to launch on GPU

Function cublasZher2()

```
void
cublasZher2 (char uplo, int n, cuDoubleComplex alpha,
             const cuDoubleComplex *x, int incx,
             const cuDoubleComplex *y, int incy,
             cuDoubleComplex *A, int lda)
```

performs the Hermitian rank 2 operation

$$A = \alpha * x * y^H + \overline{\alpha} * y * x^H + A,$$

where α is a double-precision complex scalar, x and y are n -element double-precision complex vectors, and A is an $n \times n$ Hermitian matrix consisting of double-precision complex elements.

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 and the lower triangular part of A is inferred. If <code>uplo == 'L' or 'l'</code> , only the lower triangular part of A may be referenced and the upper triangular part of A is inferred.
<code>n</code>	the number of rows and columns of matrix A ; n must be at least zero.
<code>alpha</code>	double-precision complex scalar multiplier applied to $x * y^H$ and whose conjugate is applied to $y * x^H$.
<code>x</code>	double-precision array of length at least $(1 + (n - 1) * \text{abs}(\text{incx}))$.
<code>incx</code>	the storage spacing between elements of x ; <code>incx</code> must not be zero.
<code>y</code>	double-precision array of length at least $(1 + (n - 1) * \text{abs}(\text{incy}))$.
<code>incy</code>	the storage spacing between elements of y ; <code>incy</code> must not be zero.
<code>A</code>	double-precision complex array of dimensions (lda, n) . If <code>uplo == 'U' or 'u'</code> , A contains the upper triangular part of the Hermitian 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 Hermitian matrix, and the strictly upper triangular part is not referenced. The imaginary parts of the diagonal elements need not be set, they are assumed to be zero, and on exit they are set to zero.
<code>lda</code>	leading dimension of A ; <code>lda</code> must be at least $\max(1, n)$.

Output

A	updated according to $A = \alpha * x * y^H + \overline{\alpha} * y * x^H + A$
---	-------------------------------------------------------------------------------

Reference: <http://www.netlib.org/blas/zher2.f>

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

Error Status

CUBLAS_STATUS_NOT_INITIALIZED	if CUBLAS library was not initialized
CUBLAS_STATUS_INVALID_VALUE	if $n < 0$, $incx == 0$, or $incy == 0$
CUBLAS_STATUS_ARCH_MISMATCH	if function invoked on device that does not support double precision
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function cublasZhpmv()

```
void
cublasZhpmv (char uplo, int n, cuDoubleComplex alpha,
             const cuDoubleComplex *AP,
             const cuDoubleComplex *x, int incx,
             cuDoubleComplex beta,
             cuDoubleComplex *y, int incy)
```

performs the matrix-vector operation

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

where alpha and beta are double-precision complex scalars, and x and y are n-element double-precision complex vectors. A is a Hermitian n×n matrix that consists of double-precision complex elements and is supplied in packed form.

Input

uplo	specifies whether the matrix data is stored in the upper or the lower triangular part of array AP. If <code>uplo == 'U' or 'u'</code> , the upper triangular part of A is supplied in AP. If <code>uplo == 'L' or 'l'</code> , the lower triangular part of A is supplied in AP.
n	the number of rows and columns of matrix A; n must be at least zero.
alpha	double-precision complex scalar multiplier applied to $A * x$.

Input (continued)

<code>AP</code>	double-precision complex array with at least $(n * (n + 1)) / 2$ elements. If <code>uplo == 'U' or 'u'</code> , array <code>AP</code> contains the upper triangular part of the Hermitian 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> , the array <code>AP</code> contains the lower triangular part of the Hermitian 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> . The imaginary parts of the diagonal elements need not be set; they are assumed to be zero.
<code>x</code>	double-precision complex 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>	double-precision scalar multiplier applied to vector <code>y</code> .
<code>y</code>	double-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/zhpmv.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 $n < 0$, <code>incx == 0</code> , or <code>incy == 0</code>
<code>CUBLAS_STATUS_ARCH_MISMATCH</code>	if function invoked on device that does not support double precision
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	if function failed to launch on GPU

Function cublasZhpr()

```
void
cublasZhpr (char uplo, int n, double alpha,
            const cuDoubleComplex *x, int incx,
            cuDoubleComplex *AP)
```

performs the Hermitian rank 1 operation

$$A = \alpha * x * x^H + A,$$

where α is a double-precision scalar, x is an n -element double-precision complex vector, and A is an $n \times n$ Hermitian matrix consisting of double-precision complex elements that 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' or 'u'</code> , the upper triangular part of A is supplied in <code>AP</code> . If <code>uplo == 'L' or 'l'</code> , the lower triangular part of A is supplied in <code>AP</code> .
<code>n</code>	the number of rows and columns of matrix A ; n must be at least zero.
<code>alpha</code>	double-precision scalar multiplier applied to $x * x^H$.
<code>x</code>	double-precision complex array of length at least $(1 + (n - 1) * \text{abs}(\text{incx}))$.
<code>incx</code>	the storage spacing between elements of x ; <code>incx</code> must not be zero.
<code>AP</code>	double-precision complex array with at least $(n * (n + 1)) / 2$ elements. If <code>uplo == 'U' or 'u'</code> , array <code>AP</code> contains the upper triangular part of the Hermitian matrix A , packed sequentially, column by column; that is, if $i \leq j$, $A[i, j]$ is stored in <code>AP[i + (j * (j + 1)) / 2]</code> . If <code>uplo == 'L' or 'l'</code> , the array <code>AP</code> contains the lower triangular part of the Hermitian matrix A , packed sequentially, column by column; that is, if $i \geq j$, $A[i, j]$ is stored in <code>AP[i + ((2 * n - j + 1) * j) / 2]</code> . The imaginary parts of the diagonal elements need not be set; they are assumed to be zero, and on exit they are set to zero.

Output

<code>A</code>	updated according to $A = \alpha * x * x^H + A$.
----------------	---------------------------------------------------

Reference: <http://www.netlib.org/blas/zhpr.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_ARCH_MISMATCH</code>	if function invoked on device that does not support double precision
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	if function failed to launch on GPU

Function `cublasZhpr2()`

```
void
cublasZhpr2 (char uplo, int n, cuDoubleComplex alpha,
             const cuDoubleComplex *x, int incx,
             const cuDoubleComplex *y, int incy,
             cuDoubleComplex *AP)
```

performs the Hermitian rank 2 operation

$$A = \alpha * x * y^H + \overline{\alpha} * y * x^H + A,$$

where α is a double-precision complex scalar, x and y are n -element double-precision complex vectors, and A is an $n \times n$ Hermitian matrix consisting of double-precision complex elements that 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 A . If <code>uplo == 'U' or 'u'</code> , only the upper triangular part of A may be referenced and the lower triangular part of A is inferred. If <code>uplo == 'L' or 'l'</code> , only the lower triangular part of A may be referenced and the upper triangular part of A is inferred.
<code>n</code>	the number of rows and columns of matrix A ; n must be at least zero.
<code>alpha</code>	double-precision complex scalar multiplier applied to $x * y^H$ and whose conjugate is applied to $y * x^H$.
<code>x</code>	double-precision complex array of length at least $(1 + (n - 1) * \text{abs}(incx))$.
<code>incx</code>	the storage spacing between elements of x ; <code>incx</code> must not be zero.
<code>y</code>	double-precision complex array of length at least $(1 + (n - 1) * \text{abs}(incy))$.

Input (continued)

<code>incy</code>	the storage spacing between elements of y ; <code>incy</code> must not be zero.
<code>AP</code>	double-precision complex array with at least $(n * (n + 1)) / 2$ elements. If <code>uplo</code> == 'U' or 'u', array <code>AP</code> contains the upper triangular part of the Hermitian 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</code> == 'L' or 'l', the array <code>AP</code> contains the lower triangular part of the Hermitian matrix A , packed sequentially, column by column; that is, if $i \geq j$, $A[i, j]$ is stored in $AP[i + ((2 * n - j + 1) * j) / 2]$. The imaginary parts of the diagonal elements need not be set; they are assumed to be zero, and on exit they are set to zero.

Output

A	updated according to $A = \alpha * x * y^H + \overline{\alpha} * y * x^H + A$
-----	-------------------------------------------------------------------------------

Reference: <http://www.netlib.org/blas/zhpr2.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 $n < 0$, <code>incx</code> == 0, or <code>incy</code> == 0
<code>CUBLAS_STATUS_ARCH_MISMATCH</code>	if function invoked on device that does not support double precision
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	if function failed to launch on GPU

Function `cublasZtbmv()`

```
void
cublasZtbmv (char uplo, char trans, char diag, int n,
             int k, const cuDoubleComplex *A, int lda,
             cuDoubleComplex *x, int incx)
```

performs one of the matrix-vector operations

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

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

x is an n -element double-precision complex vector, and A is an $n \times n$, unit or non-unit, upper or lower, triangular band matrix consisting of double-precision complex 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' or 't'</code> , $\text{op}(A) = A^T$. If <code>trans == 'C' or 'c'</code> , $\text{op}(A) = A^H$.
<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.
<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>	double-precision complex 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 $k+1$.
<code>x</code>	double-precision complex 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/ztbmv.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>k < 0</code> , or <code>n < 0</code>
<code>CUBLAS_STATUS_ALLOC_FAILED</code>	if function cannot allocate enough internal scratch vector memory
<code>CUBLAS_STATUS_ARCH_MISMATCH</code>	if function invoked on device that does not support double precision
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	if function failed to launch on GPU

Function `cublasZtbsv()`

```
void
cublasZtbsv (char uplo, char trans, char diag, int n,
             int k, const cuDoubleComplex *A, int lda,
             cuDoubleComplex *X, int incx)
```

solves one of the systems of equations

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

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

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> , 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' or 't'</code> , $\text{op}(A) = A^T$. If <code>trans == 'C', or 'c'</code> , $\text{op}(A) = A^H$.

Input (continued)

<code>diag</code>	specifies whether <code>A</code> is unit triangular. If <code>diag == 'U'</code> or <code>'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'</code> or <code>'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'</code> or <code>'u'</code> , <code>k</code> specifies the number of superdiagonals. If <code>uplo == 'L'</code> or <code>'l'</code> , <code>k</code> specifies the number of subdiagonals; <code>k</code> must be at least zero.
<code>A</code>	double-precision complex array of dimension (lda, n) . If <code>uplo == 'U'</code> or <code>'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'</code> or <code>'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>	double-precision complex array of length at least $(1 + (n - 1) * abs(incx))$.
<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/ztbsv.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> or <code>n < 0</code>
<code>CUBLAS_STATUS_ARCH_MISMATCH</code>	if function invoked on device that does not support double precision
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	if function failed to launch on GPU

Function cublasZtpmv()

```
void
cublasZtpmv (char uplo, char trans, char diag, int n,
             const cuDoubleComplex *AP,
             cuDoubleComplex *x, int incx)
```

performs one of the matrix-vector operations

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

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

x is an n -element double-precision complex vector, and A is an $n \times n$, unit or non-unit, upper or lower, triangular matrix consisting of double-precision complex 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' or 't'</code> , $\text{op}(A) = A^T$. If <code>trans == 'C' or 'c'</code> , $\text{op}(A) = A^H$.
<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.
<code>AP</code>	double-precision complex 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>	double-precision complex 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

x updated according to $x = \text{op}(A) * x$.

Reference: <http://www.netlib.org/blas/ztpmv.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> or <code>n < 0</code>
CUBLAS_STATUS_ALLOC_FAILED	if function cannot allocate enough internal scratch vector memory
CUBLAS_STATUS_ARCH_MISMATCH	if function invoked on device that does not support double precision
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function `cublasZtpsv()`

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

solves one of the systems of equations

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

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

b and x are n -element complex vectors, and A is an $n \times n$, unit or non-unit, upper or lower, triangular matrix.

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

Input

`uplo` specifies whether the matrix is an upper or lower triangular matrix. If `uplo == 'U'` or `'u'`, A is an upper triangular matrix. If `uplo == 'L'` or `'l'`, A is a lower triangular matrix.

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

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>	specifies the number of rows and columns of the matrix <code>A</code> ; <code>n</code> must be at least zero.
<code>AP</code>	double-precision complex 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>	double-precision complex 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.

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/ztpsv.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> or <code>n < 0</code>
<code>CUBLAS_STATUS_ARCH_MISMATCH</code>	if function invoked on device that does not support double precision
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	if function failed to launch on GPU

Function cublasZtrmv()

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

performs one of the matrix-vector operations

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

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

x is an n -element double-precision complex vector; and A is an $n \times n$, unit or non-unit, upper or lower, triangular matrix consisting of double-precision complex 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 $\text{op}(A)$. If <code>trans == 'N' or 'n'</code> , $\text{op}(A) = A$. If <code>trans == 'T' or 't'</code> , $\text{op}(A) = A^T$. If <code>trans == 'C' or 'c'</code> , $\text{op}(A) = A^H$.
<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.
<code>A</code>	double-precision complex 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 ; lda must be at least $\max(1, n)$.

Input (continued)

<code>x</code>	double-precision complex 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>	the storage spacing between 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/ztrmv.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> Or <code>n < 0</code>
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	if function failed to launch on GPU
<code>CUBLAS_STATUS_ALLOC_FAILED</code>	if function cannot allocate enough internal scratch vector memory

Function `cublasZtrsv()`

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

solves a system of equations

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

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

`b` and `x` are `n`-element double-precision complex 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 <code>A</code> . If <code>uplo == 'U' or 'u'</code> , only the upper triangular part of <code>A</code> may be referenced. If <code>uplo == 'L' or 'l'</code> , only the lower triangular part of <code>A</code> 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' or 't'</code> , $\text{op}(A) = A^T$. If <code>trans == 'C' or 'c'</code> , $\text{op}(A) = A^H$.
<code>diag</code>	specifies whether or not <code>A</code> is a unit triangular matrix. If <code>diag == 'U' or 'u'</code> , <code>A</code> is assumed to be unit triangular. 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.
<code>A</code>	double-precision complex 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 the two-dimensional array containing <code>A</code> ; <code>lda</code> must be at least $\max(1, n)$.
<code>x</code>	double-precision complex 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 solution vector <code>x</code> .
<code>incx</code>	the 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/ztrsv.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> or <code>n < 0</code>
<code>CUBLAS_STATUS_ARCH_MISMATCH</code>	if function invoked on device that does not support double precision
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	if function failed to launch on GPU

BLAS3 Functions

Level 3 Basic Linear Algebra Subprograms (BLAS3) perform matrix-matrix operations. The CUBLAS implementations are described in the following sections:

- ❑ “Single-Precision BLAS3 Functions” on page 176
- ❑ “Single-Precision Complex BLAS3 Functions” on page 189
- ❑ “Double-Precision BLAS3 Functions” on page 208
- ❑ “Double-Precision Complex BLAS3 Functions” on page 221

Single-Precision BLAS3 Functions

The single-precision BLAS3 functions are listed below:

- ❑ [“Function cublasSgemm\(\)” on page 177](#)
- ❑ [“Function cublasSsymm\(\)” on page 178](#)
- ❑ [“Function cublasSsyrk\(\)” on page 180](#)
- ❑ [“Function cublasSsyr2k\(\)” on page 182](#)
- ❑ [“Function cublasStrmm\(\)” on page 184](#)
- ❑ [“Function cublasStrsm\(\)” on page 186](#)

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

Input (continued)

B	single-precision array of dimensions (ldb, n) if transb == 'N' or 'n', and of dimensions (ldb, k) otherwise. If transb == 'N' or 'n', ldb must be at least max(1, k); otherwise, ldb must be at least max(1, n).
ldb	leading dimension of two-dimensional array used to store matrix B.
beta	single-precision scalar multiplier applied to c. If zero, c does not have to be a valid input.
C	single-precision array of dimensions (ldc, n); ldc must be at least max(1, m).
ldc	leading dimension of two-dimensional array used to store matrix C.

Output

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

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

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

Error Status

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

Function cublasSsymm()

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

performs one of the matrix-matrix operations

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

where alpha and beta are single-precision scalars, A is a symmetric matrix consisting of single-precision elements and is stored in either

lower or upper storage mode. B and C are $m \times n$ matrices consisting of single-precision elements.

Input

side	<p>specifies whether the symmetric matrix A appears on the left-hand side or right-hand side of matrix B.</p> <p>If side == 'L' or 'l', $C = \alpha * A * B + \beta * C$.</p> <p>If side == 'R' or 'r', $C = \alpha * B * A + \beta * C$.</p>
uplo	<p>specifies whether the symmetric matrix A is stored in upper or lower storage mode. If uplo == 'U' or 'u', only the upper triangular part of the symmetric matrix is referenced, and the elements of the strictly lower triangular part are inferred from those in the upper triangular part. If uplo == 'L' or 'l', only the lower triangular part of the symmetric matrix is referenced, and the elements of the strictly upper triangular part are inferred from those in the lower triangular part.</p>
m	<p>specifies the number of rows of matrix C, and the number of rows of matrix B. It also specifies the dimensions of symmetric matrix A when side == 'L' or 'l'; m must be at least zero.</p>
n	<p>specifies the number of columns of matrix C, and the number of columns of matrix B. It also specifies the dimensions of symmetric matrix A when side == 'R' or 'r'; n must be at least zero.</p>
alpha	<p>single-precision scalar multiplier applied to $A * B$ or $B * A$.</p>
A	<p>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.</p>
lda	<p>leading dimension of A. When side == 'L' or 'l', it must be at least $\max(1, m)$ and at least $\max(1, n)$ otherwise.</p>

Input (continued)

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; <code>ldb</code> must be at least $\max(1, m)$.
beta	single-precision scalar multiplier applied to C. If <code>beta</code> is zero, <code>c</code> does not have to be a valid input.
C	single-precision array of dimensions (ldc, n) .
ldc	leading dimension of C; <code>ldc</code> 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

<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 `cublasSsyrk()`

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

performs one of the symmetric rank k operations

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

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 $\max(1, k)$.
<code>beta</code>	single-precision scalar multiplier applied to <code>c</code> . If <code>beta</code> is zero, <code>c</code> is not read.

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 $C = \alpha * A * A^T + \beta * C$ 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 <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.
<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> must contain the matrix <code>A</code> , otherwise the leading $k \times n$ part of the array must contain 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 $\max(1, k)$.
<code>B</code>	single-precision array of dimensions (ldb, kb) , where <code>kb</code> = <code>k</code> when <code>trans == 'N' or 'n'</code> , and <code>k</code> = <code>n</code> otherwise. When <code>trans == 'N' or 'n'</code> , the leading $n \times k$ part of array <code>B</code> must contain the matrix <code>B</code> , otherwise the leading $k \times n$ part of the array must contain the matrix <code>B</code> .
<code>ldb</code>	leading dimension of <code>B</code> . 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'</code> or <code>'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'</code> or <code>'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 $C = \alpha * A * B^T + \alpha * B * A^T + \beta * C \text{ or}$ $C = \alpha * A^T * B + \alpha * B^T * A + \beta * C.$
----------------	-------------------------------------------------------------------------------------------------------------------------------------------------

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

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

Error Status

<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

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

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

α 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

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

Input (continued)

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

Output

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

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

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

Error Status

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

Function cublasStrsm()

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

solves one of the matrix equations

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

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

α 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	<p>specifies whether $\text{op}(A)$ appears on the left or right of x:</p> <p>side == 'L' or 'l' indicates solve $\text{op}(A) * X = \alpha * B$;</p> <p>side == 'R' or 'r' indicates solve $X * \text{op}(A) = \alpha * B$.</p>
uplo	<p>specifies whether the matrix A is an upper or lower triangular matrix:</p> <p>uplo == 'U' or 'u' indicates A is an upper triangular matrix;</p> <p>uplo == 'L' or 'l' indicates A is a lower triangular matrix.</p>
transa	<p>specifies the form of $\text{op}(A)$ to be used in matrix multiplication.</p> <p>If transa == 'N' or 'n', $\text{op}(A) = A$.</p> <p>If transa == 'T', 't', 'C', or 'c', $\text{op}(A) = A^T$.</p>
diag	<p>specifies whether or not A is a unit triangular matrix.</p> <p>If diag == 'U' or 'u', A is assumed to be unit triangular.</p> <p>If diag == 'N' or 'n', A is not assumed to be unit triangular.</p>
m	specifies the number of rows of B; m must be at least zero.
n	specifies the number of columns of B; n must be at least zero.
alpha	single-precision scalar multiplier applied to B. When alpha is zero, A is not referenced and B does not have to be a valid input.
A	<p>single-precision array of dimensions (lda, k), where k is m when side == 'L' or 'l' and is n when side == 'R' or 'r'. If uplo == 'U' or 'u', the leading $k \times k$ upper triangular part of the array A must contain the upper triangular matrix, and the strictly lower triangular matrix of A is not referenced. When uplo == 'L' or 'l', the leading $k \times k$ lower triangular part of the array A must contain the lower triangular matrix, and the strictly upper triangular part of A is not referenced. Note that when diag == 'U' or 'u', the diagonal elements of A are not referenced and are assumed to be unity.</p>
lda	<p>leading dimension of the two-dimensional array containing A.</p> <p>When side == 'L' or 'l', lda must be at least $\max(1, m)$.</p> <p>When side == 'R' or 'r', lda must be at least $\max(1, n)$.</p>
B	<p>single-precision array of dimensions (ldb, n); ldb must be at least $\max(1, m)$. The leading $m \times n$ part of the array B must contain the right-hand side matrix B. On exit B is overwritten by the solution matrix X.</p>
ldb	leading dimension of the two-dimensional array containing B; ldb must be at least $\max(1, m)$.

Output

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

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

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

Error Status

<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

These are the single-precision complex BLAS3 functions:

- ❑ “Function `cublasCgemm()`” on page 190
- ❑ “Function `cublasChemmm()`” on page 191
- ❑ “Function `cublasCherk()`” on page 193
- ❑ “Function `cublasCher2k()`” on page 195
- ❑ “Function `cublasCsymm()`” on page 197
- ❑ “Function `cublasCsyrk()`” on page 199
- ❑ “Function `cublasCsyr2k()`” on page 201
- ❑ “Function `cublasCtrmm()`” on page 203
- ❑ “Function `cublasCtrsm()`” on page 205

Function cublasCgemm()

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

performs one of the matrix-matrix operations

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

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

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

Input (continued)

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

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

Function `cublasChemmm()`

```
void
cublasChemmm (char side, char uplo, int m, int n,
               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

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

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

Input

<code>side</code>	<p>specifies whether the Hermitian matrix <code>A</code> appears on the left-hand side or right-hand side of matrix <code>B</code>.</p> <p>If <code>side == 'L' or 'l'</code>, $C = \alpha * A * B + \beta * C$.</p> <p>If <code>side == 'R' or 'r'</code>, $C = \alpha * B * A + \beta * C$.</p>
<code>uplo</code>	<p>specifies whether the Hermitian matrix <code>A</code> is stored in upper or lower storage mode. If <code>uplo == 'U' or 'u'</code>, only the upper triangular part of the Hermitian 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 Hermitian matrix is referenced, and the elements of the strictly upper triangular part are inferred from those in the lower triangular part.</p>
<code>m</code>	<p>specifies the number of rows of matrix <code>C</code>, and the number of rows of matrix <code>B</code>. It also specifies the dimensions of Hermitian matrix <code>A</code> when <code>side == 'L' or 'l'</code>; <code>m</code> must be at least zero.</p>
<code>n</code>	<p>specifies the number of columns of matrix <code>C</code>, and the number of columns of matrix <code>B</code>. It also specifies the dimensions of Hermitian matrix <code>A</code> when <code>side == 'R' or 'r'</code>; <code>n</code> must be at least zero.</p>
<code>alpha</code>	single-precision complex scalar multiplier applied to $A * B$ or $B * A$.
<code>A</code>	<p>single-precision complex array of dimensions (lda, ka), where <code>ka</code> is <code>m</code> when <code>side == 'L' or 'l'</code> and is <code>n</code> otherwise. If <code>side == 'L' or 'l'</code>, the leading <code>m</code>×<code>m</code> part of array <code>A</code> must contain the Hermitian matrix such that when <code>uplo == 'U' or 'u'</code>, the leading <code>m</code>×<code>m</code> part stores the upper triangular part of the Hermitian matrix, and the strictly lower triangular part of <code>A</code> is not referenced; and when <code>uplo == 'L' or 'l'</code>, the leading <code>m</code>×<code>m</code> part stores the lower triangular part of the Hermitian matrix, and the strictly upper triangular part is not referenced. If <code>side == 'R' or 'r'</code>, the leading <code>n</code>×<code>n</code> part of array <code>A</code> must contain the Hermitian matrix such that when <code>uplo == 'U' or 'u'</code>, the leading <code>n</code>×<code>n</code> part stores the upper triangular part of the Hermitian matrix, and the strictly lower triangular part of <code>A</code> is not referenced; and when <code>uplo == 'L' or 'l'</code>, the leading <code>n</code>×<code>n</code> part stores the lower triangular part of the Hermitian matrix, and the strictly upper triangular part is not referenced. The imaginary parts of the diagonal elements need not be set; they are assumed to be zero.</p>
<code>lda</code>	<p>leading dimension of <code>A</code>. When <code>side == 'L' or 'l'</code>, it must be at least $\max(1, m)$ and at least $\max(1, n)$ otherwise.</p>

Input (continued)

B	single-precision complex 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; <code>ldb</code> must be at least $\max(1, m)$.
beta	single-precision complex scalar multiplier applied to C. If <code>beta</code> is zero, C does not have to be a valid input.
C	single-precision complex array of dimensions (ldc, n) .
ldc	leading dimension of C; <code>ldc</code> 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/chemm.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 `cublasCherk()`

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

performs one of the Hermitian rank k operations

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

where `alpha` and `beta` are single-precision real scalars. C is an $n \times n$ Hermitian matrix consisting of single-precision complex elements and is stored in either lower or upper storage mode. A is a matrix consisting of single-precision complex 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 Hermitian 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 Hermitian 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 Hermitian 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> , $C = \alpha * A * A^H + \text{beta} * C$. If <code>trans == 'T', 't', 'C', or 'c'</code> , $C = \alpha * A^H * A + \text{beta} * C$.
<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 $A * A^H$ or $A^H * A$.
<code>A</code>	single-precision complex 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 $\max(1, k)$.
<code>beta</code>	single-precision real 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 complex array of dimensions (ldc, n) . If <code>uplo == 'U'</code> or <code>'u'</code> , the leading $n \times n$ triangular part of the array <code>C</code> must contain the upper triangular part of the Hermitian 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'</code> or <code>'l'</code> , the leading $n \times n$ triangular part of the array <code>C</code> must contain the lower triangular part of the Hermitian 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. The imaginary parts of the diagonal elements need not be set; they are assumed to be zero, and on exit they are set to zero.
<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 $C = \alpha * A * A^H + \beta * C$ or $C = \alpha * A^H * A + \beta * C$.
----------------	----------------------------------------------------------------------------------------------------

Reference: <http://www.netlib.org/blas/cherk.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 `cublasCher2k()`

```
void
cublasCher2k (char uplo, char trans, int n, int k,
              cuComplex alpha, const cuComplex *A,
              int lda, const cuComplex *B, int ldb,
              float beta, cuComplex *C, int ldc)
```

performs one of the Hermitian rank $2k$ operations

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

$$C = \alpha * A^H * B + \overline{\alpha} * B^H * A + \beta * C,$$

where α is a single-precision complex scalar and β is a single-precision real scalar. C is an $n \times n$ Hermitian matrix consisting of single-precision complex elements and is stored in either lower or upper storage mode. A and B are matrices consisting of single-precision complex 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 Hermitian matrix C is stored in upper or lower storage mode. If <code>uplo == 'U' or 'u'</code> , only the upper triangular part of the Hermitian 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 Hermitian 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> , $C = \alpha * A * B^H + \overline{\alpha} * B * A^H + \beta * C$. If <code>trans == 'T'</code> , <code>'t'</code> , <code>'C'</code> , or <code>'c'</code> , $C = \alpha * A^H * B + \overline{\alpha} * B^H * A + \beta * C$.
<code>n</code>	specifies the number of rows and the number columns of matrix C . If <code>trans == 'N' or 'n'</code> , <code>n</code> specifies the number of rows of matrices A and B . If <code>trans == 'T'</code> , <code>'t'</code> , <code>'C'</code> , or <code>'c'</code> , <code>n</code> specifies the number of columns of matrices A and B ; <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 matrices A and B . If <code>trans == 'T'</code> , <code>'t'</code> , <code>'C'</code> , or <code>'c'</code> , <code>k</code> specifies the number of rows of matrices A and B ; <code>k</code> must be at least zero.
<code>alpha</code>	single-precision complex scalar multiplier.
<code>A</code>	single-precision complex 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 A contains the matrix A ; otherwise, the leading $k \times n$ part of the array contains 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 complex array of dimensions (ldb, kb) , where <code>kb</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 B contains the matrix B ; otherwise, the leading $k \times n$ part of the array contains 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 $\max(1, k)$.

Input (continued)

<code>beta</code>	single-precision real 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 complex array of dimensions (ldc, n) . If <code>uplo == 'U'</code> or <code>'u'</code> , the leading $n \times n$ triangular part of the array <code>c</code> must contain the upper triangular part of the Hermitian 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'</code> or <code>'l'</code> , the leading $n \times n$ triangular part of the array <code>c</code> must contain the lower triangular part of the Hermitian 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. The imaginary parts of the diagonal elements need not be set; they are assumed to be zero, and on exit they are set to zero.
<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 $C = \alpha * A * B^H + \overline{\alpha} * B * A^H + \beta * C$ or $C = \alpha * A^H * B + \overline{\alpha} * B^H * A + \beta * C$.
----------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------

Reference: <http://www.netlib.org/blas/cher2k.f>

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

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 `cublasCsymm()`

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

performs one of the matrix-matrix operations

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

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

Input

<code>side</code>	<p>specifies whether the symmetric matrix <code>A</code> appears on the left-hand side or right-hand side of matrix <code>B</code>.</p> <p>If <code>side == 'L' or 'l'</code>, $C = \alpha * A * B + \beta * C$.</p> <p>If <code>side == 'R' or 'r'</code>, $C = \alpha * B * A + \beta * C$.</p>
<code>uplo</code>	<p>specifies whether the symmetric matrix <code>A</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.</p>
<code>m</code>	<p>specifies the number of rows of matrix <code>C</code>, and the number of rows of matrix <code>B</code>. It also specifies the dimensions of symmetric matrix <code>A</code> when <code>side == 'L' or 'l'</code>; <code>m</code> must be at least zero.</p>
<code>n</code>	<p>specifies the number of columns of matrix <code>C</code>, and the number of columns of matrix <code>B</code>. It also specifies the dimensions of symmetric matrix <code>A</code> when <code>side == 'R' or 'r'</code>; <code>n</code> must be at least zero.</p>
<code>alpha</code>	single-precision complex scalar multiplier applied to $A * B$ or $B * A$.
<code>A</code>	<p>single-precision complex array of dimensions (lda, ka), where <code>ka</code> is <code>m</code> when <code>side == 'L' or 'l'</code> and is <code>n</code> otherwise. If <code>side == 'L' or 'l'</code>, the leading $m \times m$ part of array <code>A</code> must contain the symmetric matrix such that when <code>uplo == 'U' or 'u'</code>, the leading $m \times m$ part stores the upper triangular part of the symmetric matrix, and the strictly lower triangular part of <code>A</code> is not referenced; and when <code>uplo == 'L' or 'l'</code>, 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 <code>side == 'R' or 'r'</code>, the leading $n \times n$ part of array <code>A</code> must contain the symmetric matrix such that when <code>uplo == 'U' or 'u'</code>, the leading $n \times n$ part stores the upper triangular part of the symmetric matrix, and the strictly lower triangular part of <code>A</code> is not referenced; and when <code>uplo == 'L' or 'l'</code>, the leading $n \times n$ part stores the lower triangular part of the symmetric matrix, and the strictly upper triangular part is not referenced.</p>

Input (continued)

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

```
void
cublasCsyrrk (char uplo, char trans, int n, int k,
              cuComplex alpha, const cuComplex *A,
              int lda, cuComplex beta, cuComplex *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 complex scalars. C is an $n \times n$ symmetric matrix consisting of single-precision complex elements and is stored in either lower or upper storage mode. A is a matrix consisting of single-precision complex 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 complex scalar multiplier applied to $A * A^T$ or $A^T * A$.
<code>A</code>	single-precision complex 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 $\max(1, k)$.
<code>beta</code>	single-precision complex scalar multiplier applied to <code>c</code> . If <code>beta</code> is zero, <code>c</code> is not read.

Input (continued)

<code>C</code>	single-precision complex array of dimensions (ldc, n) . If <code>uplo == 'U'</code> or <code>'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'</code> or <code>'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 $C = \alpha * A * A^T + \beta * C$ or $C = \alpha * A^T * A + \beta * C$.
----------------	----------------------------------------------------------------------------------------------------

Reference: <http://www.netlib.org/blas/csyrrk.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 `cublasCsyrrk2k()`

```
void
cublasCsyrrk2k (char uplo, char trans, 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 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 complex scalars. `C` is an $n \times n$ symmetric matrix consisting of single-precision complex elements and is stored in either lower or upper storage mode. `A` and `B` are

matrices consisting of single-precision complex 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> , $C = \alpha * A * B^T + \alpha * B * A^T + \beta * C$. 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 <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 complex scalar multiplier.
<code>A</code>	single-precision complex 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 $\max(1, k)$.
<code>B</code>	single-precision complex array of dimensions (ldb, kb) , where <code>kb</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>B</code> contains the matrix <code>B</code> ; otherwise, the leading $k \times n$ part of the array contains the matrix <code>B</code> .
<code>ldb</code>	leading dimension of <code>B</code> . 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 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.

Input (continued)

<code>C</code>	single-precision complex array of dimensions (ldc, n) . If <code>uplo == 'U'</code> or <code>'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'</code> or <code>'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 $C = \alpha * A * B^T + \alpha * B * A^T + \beta * C$ or $C = \alpha * A^T * B + \alpha * B^T * A + \beta * C$.
----------------	------------------------------------------------------------------------------------------------------------------------------------------

Reference: <http://www.netlib.org/blas/csyrr2k.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 `cublasCtrmm()`

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

performs one of the matrix-matrix operations

$B = \alpha * \text{op}(A) * B$ or $B = \alpha * B * \text{op}(A)$,
 where $\text{op}(A) = A$, $\text{op}(A) = A^T$, or $\text{op}(A) = A^H$;

`alpha` is a single-precision complex scalar; `B` is an $m \times n$ matrix consisting of single-precision complex elements; and `A` is a unit or non-

unit, upper or lower triangular matrix consisting of single-precision complex 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 <code>op(A)</code> 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 <code>op(A)</code> . If <code>transa == 'N' or 'n'</code> , $op(A) = A$. If <code>transa == 'T' or 't'</code> , $op(A) = A^T$. If <code>transa == 'C' or 'c'</code> , $op(A) = A^H$.
<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 complex 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 complex 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.

Input (continued)

B	single-precision complex 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.
ldb	leading dimension of B; ldb must be at least $\max(1, m)$.

Output

B	updated according to $B = \alpha * \text{op}(A) * B$ or $B = \alpha * B * \text{op}(A)$.
---	----------------------------------------------------------------------------------------------

Reference: <http://www.netlib.org/blas/ctrmm.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 cublasCtrsm()

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

solves one of the matrix equations

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

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

alpha is a single-precision complex scalar, and X and B are $m \times n$ matrices that consist of single-precision complex 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

<code>side</code>	specifies whether $\text{op}(A)$ appears on the left or right of x : <code>side == 'L' or 'l'</code> indicates solve $\text{op}(A) * X = \alpha * B$; <code>side == 'R' or 'r'</code> indicates solve $X * \text{op}(A) = \alpha * B$.
<code>uplo</code>	specifies whether the matrix <code>A</code> is an upper or lower triangular matrix: <code>uplo == 'U' or 'u'</code> indicates <code>A</code> is an upper triangular matrix; <code>uplo == 'L' or 'l'</code> indicates <code>A</code> is a lower triangular matrix.
<code>transa</code>	specifies $\text{op}(A)$. If <code>transa == 'N' or 'n'</code> , $\text{op}(A) = A$. If <code>transa == 'T' or 't'</code> , $\text{op}(A) = A^T$. If <code>transa == 'C' or 'c'</code> , $\text{op}(A) = A^H$.
<code>diag</code>	specifies whether or not <code>A</code> is a unit triangular matrix. If <code>diag == 'U' or 'u'</code> , <code>A</code> is assumed to be unit triangular. If <code>diag == 'N' or 'n'</code> , <code>A</code> is not assumed to be unit triangular.
<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 complex 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 complex array of dimensions (lda, k) , where <code>k</code> is <code>m</code> when <code>side == 'L' or 'l'</code> and is <code>n</code> when <code>side == 'R' or 'r'</code> . If <code>uplo == 'U' or 'u'</code> , 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 == 'L' or 'l'</code> , 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 == 'U' or 'u'</code> , 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 == 'L' or 'l'</code> , <code>lda</code> must be at least $\max(1, m)$. When <code>side == 'R' or 'r'</code> , <code>lda</code> must be at least $\max(1, n)$.
<code>B</code>	single-precision complex 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

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

Reference: <http://www.netlib.org/blas/ctrsm.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

Double-Precision BLAS3 Functions

Note: Double-precision functions are only supported on GPUs with double-precision hardware.

The double-precision BLAS3 functions are listed below:

- ❑ “Function `cublasDgemm()`” on page 209
- ❑ “Function `cublasDsymm()`” on page 210
- ❑ “Function `cublasDsyrk()`” on page 212
- ❑ “Function `cublasDsyr2k()`” on page 214
- ❑ “Function `cublasDtrmm()`” on page 216
- ❑ “Function `cublasDtrsm()`” on page 218

Function cublasDgemm()

```
void
cublasDgemm (char transa, char transb, int m, int n,
             int k, double alpha, const double *A,
             int lda, const double *B, int ldb,
             double beta, double *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 double-precision scalars. A, B, and C are matrices consisting of double-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.
n	number of columns of matrix $\text{op}(B)$ and number of columns of C; n must be at least zero.
k	number of columns of matrix $\text{op}(A)$ and number of rows of $\text{op}(B)$; k must be at least zero.
alpha	double-precision scalar multiplier applied to $\text{op}(A) * \text{op}(B)$.
A	double-precision array of dimensions (lda, k) if transa == 'N' OR 'n', and of dimensions (lda, m) otherwise. If transa == 'N' OR 'n', lda must be at least $\max(1, m)$; otherwise, lda must be at least $\max(1, k)$.
lda	leading dimension of two-dimensional array used to store matrix A.

Input (continued)

B	double-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> , ldb must be at least <code>max(1, k)</code> ; otherwise, ldb must be at least <code>max(1, n)</code> .
ldb	leading dimension of two-dimensional array used to store matrix B.
beta	double-precision scalar multiplier applied to c. If zero, c does not have to be a valid input.
C	double-precision array of dimensions (ldc, n); ldc must be at least <code>max(1, m)</code> .
ldc	leading dimension of two-dimensional array used to store matrix c.

Output

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

Reference: <http://www.netlib.org/blas/dgemm.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>m < 0</code> , <code>n < 0</code> , or <code>k < 0</code>
CUBLAS_STATUS_ARCH_MISMATCH	if function invoked on device that does not support double precision
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function `cublasDsymm()`

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

performs one of the matrix-matrix operations

`C = alpha * A * B + beta * C` or `C = alpha * B * A + beta * C`,

where alpha and beta are double-precision scalars, A is a symmetric matrix consisting of double-precision elements and is stored in either

lower or upper storage mode. B and C are $m \times n$ matrices consisting of double-precision elements.

Input

side	<p>specifies whether the symmetric matrix A appears on the left-hand side or right-hand side of matrix B.</p> <p>If side == 'L' or 'l', $C = \alpha * A * B + \beta * C$.</p> <p>If side == 'R' or 'r', $C = \alpha * B * A + \beta * C$.</p>
uplo	<p>specifies whether the symmetric matrix A is stored in upper or lower storage mode. If uplo == 'U' or 'u', only the upper triangular part of the symmetric matrix is referenced, and the elements of the strictly lower triangular part are inferred from those in the upper triangular part. If uplo == 'L' or 'l', only the lower triangular part of the symmetric matrix is referenced, and the elements of the strictly upper triangular part are inferred from those in the lower triangular part.</p>
m	<p>specifies the number of rows of matrix C, and the number of rows of matrix B. It also specifies the dimensions of symmetric matrix A when side == 'L' or 'l'; m must be at least zero.</p>
n	<p>specifies the number of columns of matrix C, and the number of columns of matrix B. It also specifies the dimensions of symmetric matrix A when side == 'R' or 'r'; n must be at least zero.</p>
alpha	<p>double-precision scalar multiplier applied to $A * B$ or $B * A$.</p>
A	<p>double-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.</p>
lda	<p>leading dimension of A. When side == 'L' or 'l', it must be at least $\max(1, m)$ and at least $\max(1, n)$ otherwise.</p>

Input (continued)

B	double-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	double-precision scalar multiplier applied to C. If beta is zero, C does not have to be a valid input.
C	double-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/dsymm.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_ARCH_MISMATCH	if function invoked on device that does not support double precision
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function `cublasDsyrrk()`

```
void
cublasDsyrrk (char uplo, char trans, int n, int k,
              double alpha, const double *A, int lda,
              double beta, double *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 α and β are double-precision scalars. C is an $n \times n$ symmetric matrix consisting of double-precision elements and is stored in either lower or upper storage mode. A is a matrix consisting of double-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>	double-precision scalar multiplier applied to $A * A^T$ or $A^T * A$.
<code>A</code>	double-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 $\max(1, k)$.
<code>beta</code>	double-precision scalar multiplier applied to <code>c</code> . If <code>beta</code> is zero, <code>c</code> is not read.

Input (continued)

<code>C</code>	double-precision array of dimensions (ldc, n) . If <code>uplo == 'U'</code> or <code>'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'</code> or <code>'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 $C = \alpha * A * A^T + \beta * C$ or $C = \alpha * A^T * A + \beta * C$.
----------------	----------------------------------------------------------------------------------------------------

Reference: <http://www.netlib.org/blas/dsyrc.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_ARCH_MISMATCH</code>	if function invoked on device that does not support double precision
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	if function failed to launch on GPU

Function `cublasDsyrc2k()`

```
void
cublasDsyrc2k (char uplo, char trans, int n, int k,
               double alpha, const double *A, int lda,
               const double *B, int ldb, double beta,
               double *C, int ldc)
```

performs one of the symmetric rank $2k$ operations

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

where `alpha` and `beta` are double-precision scalars. `C` is an $n \times n$ symmetric matrix consisting of double-precision elements and is stored in either lower or upper storage mode. `A` and `B` are matrices consisting of double-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 <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>	double-precision scalar multiplier.
<code>A</code>	double-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> must contain the matrix <code>A</code> , otherwise the leading $k \times n$ part of the array must contain 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 $\max(1, k)$.
<code>B</code>	double-precision array of dimensions (ldb, kb) , where <code>kb</code> = <code>k</code> when <code>trans == 'N' or 'n'</code> , and <code>k</code> = <code>n</code> otherwise. When <code>trans == 'N' or 'n'</code> , the leading $n \times k$ part of array <code>B</code> must contain the matrix <code>B</code> , otherwise the leading $k \times n$ part of the array must contain the matrix <code>B</code> .
<code>ldb</code>	leading dimension of <code>B</code> . 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>	double-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>	double-precision array of dimensions (ldc, n) . If <code>uplo == 'U'</code> or <code>'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'</code> or <code>'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 $C = \alpha * A * B^T + \alpha * B * A^T + \beta * C \text{ or}$ $C = \alpha * A^T * B + \alpha * B^T * A + \beta * C.$
----------------	-------------------------------------------------------------------------------------------------------------------------------------------------

Reference: <http://www.netlib.org/blas/dsyr2k.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_ARCH_MISMATCH	if function invoked on device that does not support double precision
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function `cublasDtrmm()`

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

performs one of the matrix-matrix operations

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

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

alpha is a double-precision scalar, B is an $m \times n$ matrix consisting of double-precision elements, and A is a unit or non-unit, upper or lower triangular matrix consisting of double-precision elements.

Matrices A and B are stored in column-major format, and lda and ldb are the leading dimensions of the two-dimensional arrays that contain A and B, respectively.

Input

side	specifies whether $\text{op}(A)$ multiplies B from the left or right. If side == 'L' or 'l', $B = \text{alpha} * \text{op}(A) * B$. If side == 'R' or 'r', $B = \text{alpha} * B * \text{op}(A)$.
uplo	specifies whether the matrix A is an upper or lower triangular matrix. If uplo == 'U' or 'u', A is an upper triangular matrix. If uplo == 'L' or 'l', A is a lower triangular matrix.
transa	specifies the form of $\text{op}(A)$ to be used in the 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.
m	the number of rows of matrix B; m must be at least zero.
n	the number of columns of matrix B; n must be at least zero.
alpha	double-precision scalar multiplier applied to $\text{op}(A) * B$ or $B * \text{op}(A)$, respectively. If alpha is zero, no accesses are made to matrix A, and no read accesses are made to matrix B.
A	double-precision array of dimensions (lda, k). If side == 'L' or 'l', $k = m$. If side == 'R' or 'r', $k = n$. If uplo == 'U' or 'u', the leading $k \times k$ 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 uplo == 'L' or 'l', the leading $k \times k$ lower triangular part of the array A must contain the lower triangular matrix, and the strictly upper triangular part of A is not referenced. When diag == 'U' or 'u', the diagonal elements of A are not referenced and are assumed to be unity.
lda	leading dimension of A. When side == 'L' or 'l', it must be at least $\max(1, m)$ and at least $\max(1, n)$ otherwise.

Input (continued)

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

Output

B	updated according to $B = \alpha * \text{op}(A) * B$ or $B = \alpha * B * \text{op}(A)$.
---	----------------------------------------------------------------------------------------------

Reference: <http://www.netlib.org/blas/dtrmm.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_ARCH_MISMATCH	if function invoked on device that does not support double precision
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function cublasDtrsm()

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

solves one of the matrix equations

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

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

α is a double-precision scalar, and X and B are $m \times n$ matrices that consist of double-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	<p>specifies whether $\text{op}(A)$ appears on the left or right of x:</p> <p>side == 'L' or 'l' indicates solve $\text{op}(A) * X = \alpha * B$;</p> <p>side == 'R' or 'r' indicates solve $X * \text{op}(A) = \alpha * B$.</p>
uplo	<p>specifies whether the matrix A is an upper or lower triangular matrix:</p> <p>uplo == 'U' or 'u' indicates A is an upper triangular matrix;</p> <p>uplo == 'L' or 'l' indicates A is a lower triangular matrix.</p>
transa	<p>specifies the form of $\text{op}(A)$ to be used in matrix multiplication.</p> <p>If transa == 'N' or 'n', $\text{op}(A) = A$.</p> <p>If transa == 'T', 't', 'C', or 'c', $\text{op}(A) = A^T$.</p>
diag	<p>specifies whether or not A is a unit triangular matrix.</p> <p>If diag == 'U' or 'u', A is assumed to be unit triangular.</p> <p>If diag == 'N' or 'n', A is not assumed to be unit triangular.</p>
m	specifies the number of rows of B ; m must be at least zero.
n	specifies the number of columns of B ; n must be at least zero.
alpha	double-precision scalar multiplier applied to B . When α is zero, A is not referenced and B does not have to be a valid input.
A	<p>double-precision array of dimensions (lda, k), where k is m when side == 'L' or 'l' and is n when side == 'R' or 'r'. If uplo == 'U' or 'u', the leading $k \times k$ upper triangular part of the array A must contain the upper triangular matrix, and the strictly lower triangular matrix of A is not referenced. When uplo == 'L' or 'l', the leading $k \times k$ lower triangular part of the array A must contain the lower triangular matrix, and the strictly upper triangular part of A is not referenced. Note that when diag == 'U' or 'u', the diagonal elements of A are not referenced and are assumed to be unity.</p>
lda	<p>leading dimension of the two-dimensional array containing A.</p> <p>When side == 'L' or 'l', lda must be at least $\max(1, m)$.</p> <p>When side == 'R' or 'r', lda must be at least $\max(1, n)$.</p>
B	<p>double-precision array of dimensions (ldb, n); ldb must be at least $\max(1, m)$. The leading $m \times n$ part of the array B must contain the right-hand side matrix B. On exit, B is overwritten by the solution matrix x.</p>
ldb	<p>leading dimension of the two-dimensional array containing B; ldb must be at least $\max(1, m)$.</p>

Output

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

Reference: <http://www.netlib.org/blas/dtrsm.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_ARCH_MISMATCH</code>	if function invoked on device that does not support double precision
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	if function failed to launch on GPU

Double-Precision Complex BLAS3 Functions

Note: Double-precision functions are only supported on GPUs with double-precision hardware.

Five double-precision complex BLAS3 functions are implemented:

- ❑ “Function `cublasZgemm()`” on page 222
- ❑ “Function `cublasZhemm()`” on page 223
- ❑ “Function `cublasZherk()`” on page 225
- ❑ “Function `cublasZher2k()`” on page 228
- ❑ “Function `cublasZsymm()`” on page 230
- ❑ “Function `cublasZsyrk()`” on page 232
- ❑ “Function `cublasZsyr2k()`” on page 234
- ❑ “Function `cublasZtrmm()`” on page 236
- ❑ “Function `cublasZtrsm()`” on page 238

Function cublasZgemm()

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

performs one of the matrix-matrix operations

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

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

and alpha and beta are double-precision complex scalars. A, B, and C are matrices consisting of double-precision complex 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.

Input

transa	specifies $\text{op}(A)$. If transa == 'N' OR 'n', $\text{op}(A) = A$. If transa == 'T' OR 't', $\text{op}(A) = A^T$. If transa == 'C' OR 'c', $\text{op}(A) = A^H$.
transb	specifies $\text{op}(B)$. If transb == 'N' OR 'n', $\text{op}(B) = B$. If transb == 'T' OR 't', $\text{op}(B) = B^T$. If transb == 'C' OR 'c', $\text{op}(B) = B^H$.
m	number of rows of matrix $\text{op}(A)$ and rows of matrix C; m must be at least zero.
n	number of columns of matrix $\text{op}(B)$ and number of columns of C; n must be at least zero.
k	number of columns of matrix $\text{op}(A)$ and number of rows of $\text{op}(B)$; k must be at least zero.
alpha	double-precision complex scalar multiplier applied to $\text{op}(A) * \text{op}(B)$.
A	double-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	double-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 B. When <code>transb == 'N'</code> or <code>'n'</code> , it must be at least <code>max(1, k)</code> and at least <code>max(1, n)</code> otherwise.
<code>beta</code>	double-precision complex scalar multiplier applied to C. If <code>beta</code> is zero, C does not have to be a valid input.
<code>C</code>	double-precision array of dimensions <code>(ldc, n)</code> .
<code>ldc</code>	leading dimension of C; <code>ldc</code> must be at least <code>max(1, m)</code> .

Output

<code>C</code>	updated according to <code>C = alpha * op(A) * op(B) + beta * C</code> .
----------------	--------------------------------------------------------------------------

Reference: <http://www.netlib.org/blas/zgemm.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>m < 0</code> , <code>n < 0</code> , or <code>k < 0</code>
<code>CUBLAS_STATUS_ARCH_MISMATCH</code>	if function invoked on device that does not support double precision
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	if function failed to launch on GPU

Function `cublasZhemm()`

```
void
cublasZhemm (char side, char uplo, int m, int n,
             cuDoubleComplex alpha,
             const cuDoubleComplex *A, int lda,
             const cuDoubleComplex *B, int ldb,
             cuDoubleComplex beta, cuDoubleComplex *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 `alpha` and `beta` are double-precision complex scalars, A is a Hermitian matrix consisting of double-precision complex elements

and is stored in either lower or upper storage mode. B and C are $m \times n$ matrices consisting of double-precision complex elements.

Input

side	<p>specifies whether the Hermitian matrix A appears on the left-hand side or right-hand side of matrix B.</p> <p>If <code>side == 'L' or 'l'</code>, $C = \alpha * A * B + \beta * C$.</p> <p>If <code>side == 'R' or 'r'</code>, $C = \alpha * B * A + \beta * C$.</p>
uplo	<p>specifies whether the Hermitian matrix A is stored in upper or lower storage mode. If <code>uplo == 'U' or 'u'</code>, only the upper triangular part of the Hermitian 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 Hermitian matrix is referenced, and the elements of the strictly upper triangular part are inferred from those in the lower triangular part.</p>
m	<p>specifies the number of rows of matrix C, and the number of rows of matrix B. It also specifies the dimensions of Hermitian matrix A when <code>side == 'L' or 'l'</code>; m must be at least zero.</p>
n	<p>specifies the number of columns of matrix C, and the number of columns of matrix B. It also specifies the dimensions of Hermitian matrix A when <code>side == 'R' or 'r'</code>; n must be at least zero.</p>
alpha	double-precision complex scalar multiplier applied to $A * B$ or $B * A$.
A	<p>double-precision complex array of dimensions (lda, ka), where ka is m when <code>side == 'L' or 'l'</code> and is n otherwise. If <code>side == 'L' or 'l'</code>, the leading $m \times m$ part of array A must contain the Hermitian matrix such that when <code>uplo == 'U' or 'u'</code>, the leading $m \times m$ part stores the upper triangular part of the Hermitian matrix, and the strictly lower triangular part of A is not referenced; and when <code>uplo == 'L' or 'l'</code>, the leading $m \times m$ part stores the lower triangular part of the Hermitian matrix, and the strictly upper triangular part is not referenced. If <code>side == 'R' or 'r'</code>, the leading $n \times n$ part of array A must contain the Hermitian matrix such that when <code>uplo == 'U' or 'u'</code>, the leading $n \times n$ part stores the upper triangular part of the Hermitian matrix, and the strictly lower triangular part of A is not referenced; and when <code>uplo == 'L' or 'l'</code>, the leading $n \times n$ part stores the lower triangular part of the Hermitian matrix, and the strictly upper triangular part is not referenced. The imaginary parts of the diagonal elements need not be set; they are assumed to be zero.</p>
lda	<p>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.</p>

Input (continued)

B	double-precision complex 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	double-precision complex scalar multiplier applied to C. If beta is zero, C does not have to be a valid input.
C	double-precision complex 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/zhemm.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_ARCH_MISMATCH	if function invoked on device that does not support double precision
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function cublasZherk()

```
void
cublasZherk (char uplo, char trans, int n, int k,
             double alpha, const cuDoubleComplex *A,
             int lda, double beta, cuDoubleComplex *C,
             int ldc)
```

performs one of the Hermitian rank k operations

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

where alpha and beta are double-precision scalars. C is an $n \times n$ Hermitian matrix consisting of double-precision complex elements and is stored in either lower or upper storage mode. A is a matrix

consisting of double-precision complex 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 Hermitian 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 Hermitian 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 Hermitian 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> , $C = \alpha * A * A^H + \beta * C$. If <code>trans == 'T', 't', 'C', or 'c'</code> , $C = \alpha * A^H * A + \beta * C$.
<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>	double-precision scalar multiplier applied to $A * A^H$ or $A^H * A$.
<code>A</code>	double-precision complex 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 $\max(1, k)$.
<code>beta</code>	double-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>	double-precision complex array of dimensions (ldc, n) . If <code>uplo == 'U'</code> or <code>'u'</code> , the leading $n \times n$ triangular part of the array <code>C</code> must contain the upper triangular part of the Hermitian 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'</code> or <code>'l'</code> , the leading $n \times n$ triangular part of the array <code>C</code> must contain the lower triangular part of the Hermitian 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. The imaginary parts of the diagonal elements need not be set; they are assumed to be zero, and on exit they are set to zero.
<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 $C = \alpha * A * A^H + \beta * C$ or $C = \alpha * A^H * A + \beta * C$.
----------------	-------------------------------------------------------------------------------------------------

Reference: <http://www.netlib.org/blas/zherk.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_ARCH_MISMATCH</code>	if function invoked on device that does not support double precision
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	if function failed to launch on GPU

Function cublasZher2k()

```
void
cublasZher2k (char uplo, char trans, int n, int k,
              cuDoubleComplex alpha,
              const cuDoubleComplex *A, int lda,
              const cuDoubleComplex *B, int ldb,
              double beta, cuDoubleComplex *C, int ldc)
```

performs one of the Hermitian rank 2k operations

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

$$C = \alpha * A^H * B + \overline{\alpha} * B^H * A + \beta * C,$$

where alpha is a double-precision complex scalar and beta is a double-precision real scalar. C is an $n \times n$ Hermitian matrix consisting of double-precision complex elements and is stored in either lower or upper storage mode. A and B are matrices consisting of double-precision complex elements with dimensions of $n \times k$ in the first case and $k \times n$ in the second case.

Input

uplo	specifies whether the Hermitian matrix C is stored in upper or lower storage mode. If uplo == 'U' or 'u', only the upper triangular part of the Hermitian matrix is referenced, and the elements of the strictly lower triangular part are inferred from those in the upper triangular part. If uplo == 'L' or 'l', only the lower triangular part of the Hermitian matrix is referenced, and the elements of the strictly upper triangular part are inferred from those in the lower triangular part.
trans	specifies the operation to be performed. If trans == 'N' or 'n', $C = \alpha * A * B^H + \overline{\alpha} * B * A^H + \beta * C$. If trans == 'T', 't', 'C', or 'c', $C = \alpha * A^H * B + \overline{\alpha} * B^H * A + \beta * C$.
n	specifies the number of rows and the number columns of matrix C. If trans == 'N' or 'n', n specifies the number of rows of matrix A. If trans == 'T', 't', 'C', or 'c', n specifies the number of columns of matrix A; n must be at least zero.
k	If trans == 'N' or 'n', k specifies the number of columns of matrix A. If trans == 'T', 't', 'C', or 'c', k specifies the number of rows of matrix A; k must be at least zero.
alpha	double-precision complex multiplier.

Input (continued)

A	double-precision array of dimensions (lda, ka), where ka is k when trans == 'N' or 'n' and is n otherwise. When trans == 'N' or 'n', the leading n×k part of array A contains the matrix A; otherwise, the leading k×n part of the array contains the matrix A.
lda	leading dimension of A. When trans == 'N' or 'n', lda must be at least max(1, n). Otherwise lda must be at least max(1, k).
B	double-precision array of dimensions (ldb, kb), where kb is k when trans == 'N' or 'n' and is n otherwise. When trans == 'N' or 'n', the leading n×k part of array B contains the matrix B; otherwise, the leading k×n part of the array contains the matrix B.
ldb	leading dimension of B. When trans == 'N' or 'n', ldb must be at least max(1, n). Otherwise ldb must be at least max(1, k).
beta	double-precision real scalar multiplier applied to C. If beta is zero, C does not have to be a valid input.
C	double-precision array of dimensions (ldc, n). If uplo == 'U' or 'u', the leading n×n triangular part of the array C must contain the upper triangular part of the Hermitian matrix C, and the strictly lower triangular part of C is not referenced. On exit, the upper triangular part of C is overwritten by the upper triangular part of the updated matrix. If uplo == 'L' or 'l', the leading n×n triangular part of the array C must contain the lower triangular part of the Hermitian 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. The imaginary parts of the diagonal elements need not be set; they are assumed to be zero, and on exit they are set to zero.
ldc	leading dimension of C; ldc must be at least max(1, n).

Output

C	updated according to $C = \alpha * A * B^H + \overline{\alpha} * B * A^H + \beta * C$ or $C = \alpha * A^H * B + \overline{\alpha} * B^H * A + \beta * C$.
---	----------------------------------------------------------------------------------------------------------------------------------------------------------------

Reference: <http://www.netlib.org/blas/zher2k.f>

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

Error Status

CUBLAS_STATUS_NOT_INITIALIZED	if CUBLAS library was not initialized
CUBLAS_STATUS_INVALID_VALUE	if n < 0 or k < 0

Error Status (continued)

<code>CUBLAS_STATUS_ARCH_MISMATCH</code>	if function invoked on device that does not support double precision
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	if function failed to launch on GPU

Function `cublasZsymm()`

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

performs one of the matrix-matrix operations

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

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

Input (continued)

<code>n</code>	specifies the number of columns of matrix <code>c</code> , and the number of columns of matrix <code>B</code> . It also specifies the dimensions of symmetric matrix <code>A</code> when <code>side == 'R' or 'r'</code> ; <code>n</code> must be at least zero.
<code>alpha</code>	double-precision complex scalar multiplier applied to $A * B$ or $B * A$.
<code>A</code>	double-precision complex array of dimensions (lda, ka) , where <code>ka</code> is <code>m</code> when <code>side == 'L' or 'l'</code> and is <code>n</code> otherwise. If <code>side == 'L' or 'l'</code> , the leading $m \times m$ part of array <code>A</code> must contain the symmetric matrix such that when <code>uplo == 'U' or 'u'</code> , the leading $m \times m$ part stores the upper triangular part of the symmetric matrix, and the strictly lower triangular part of <code>A</code> is not referenced; and when <code>uplo == 'L' or 'l'</code> , 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 <code>side == 'R' or 'r'</code> , the leading $n \times n$ part of array <code>A</code> must contain the symmetric matrix such that when <code>uplo == 'U' or 'u'</code> , the leading $n \times n$ part stores the upper triangular part of the symmetric matrix, and the strictly lower triangular part of <code>A</code> is not referenced; and when <code>uplo == 'L' or 'l'</code> , the leading $n \times n$ part stores 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> . 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>	double-precision complex array of dimensions (ldb, n) . On entry, the leading $m \times n$ part of the array contains the matrix <code>B</code> .
<code>ldb</code>	leading dimension of <code>B</code> ; <code>ldb</code> must be at least $\max(1, m)$.
<code>beta</code>	double-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>	double-precision complex 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 $C = \alpha * A * B + \beta * C$ or $C = \alpha * B * A + \beta * C$.
----------------	------------------------------------------------------------------------------------------------

Reference: <http://www.netlib.org/blas/zsymm.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_ARCH_MISMATCH</code>	if function invoked on device that does not support double precision
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	if function failed to launch on GPU

Function `cublasZsyrk()`

```
void
cublasZsyrk (char uplo, char trans, int n, int k,
             cuDoubleComplex alpha,
             const cuDoubleComplex *A, int lda,
             cuDoubleComplex beta,
             cuDoubleComplex *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 α and β are double-precision complex scalars. C is an $n \times n$ symmetric matrix consisting of double-precision complex elements and is stored in either lower or upper storage mode. A is a matrix consisting of double-precision complex 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 C 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> , $C = \alpha * A^T * A + \beta * C$.

Input (continued)

<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>	double-precision complex scalar multiplier applied to $A * A^T$ or $A^T * A$.
<code>A</code>	double-precision complex 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 $\max(1, k)$.
<code>beta</code>	double-precision complex scalar multiplier applied to <code>c</code> . If <code>beta</code> is zero, <code>c</code> is not read.
<code>C</code>	double-precision complex 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 $C = \alpha * A * A^T + \beta * C$ or $C = \alpha * A^T * A + \beta * C$.
----------------	----------------------------------------------------------------------------------------------------

Reference: <http://www.netlib.org/blas/zsyrrk.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_ARCH_MISMATCH</code>	if function invoked on device that does not support double precision
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	if function failed to launch on GPU

Function `cublasZsyr2k()`

```
void
cublasZsyr2k (char uplo, char trans, int n, int k,
              cuDoubleComplex alpha,
              const cuDoubleComplex *A, int lda,
              const cuDoubleComplex *B, int ldb,
              cuDoubleComplex beta,
              cuDoubleComplex *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 α and β are double-precision complex scalars. C is an $n \times n$ symmetric matrix consisting of double-precision complex elements and is stored in either lower or upper storage mode. A and B are matrices consisting of double-precision complex 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 C 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> , $C = \alpha * A^T * B + \alpha * B^T * A + \beta * C$.

Input (continued)

<code>n</code>	specifies the number of rows and the number columns of matrix <code>C</code> . If <code>trans == 'N'</code> or <code>'n'</code> , <code>n</code> specifies the number of rows of matrices <code>A</code> and <code>B</code> . If <code>trans == 'T'</code> , <code>'t'</code> , <code>'C'</code> , or <code>'c'</code> , <code>n</code> specifies the number of columns of matrices <code>A</code> and <code>B</code> ; <code>n</code> must be at least zero.
<code>k</code>	If <code>trans == 'N'</code> or <code>'n'</code> , <code>k</code> specifies the number of columns of matrices <code>A</code> and <code>B</code> . If <code>trans == 'T'</code> , <code>'t'</code> , <code>'C'</code> , or <code>'c'</code> , <code>k</code> specifies the number of rows of matrices <code>A</code> and <code>B</code> ; <code>k</code> must be at least zero.
<code>alpha</code>	double-precision scalar multiplier.
<code>A</code>	double-precision array of dimensions (lda, ka) , where ka is k when <code>trans == 'N'</code> or <code>'n'</code> and is n otherwise. When <code>trans == 'N'</code> or <code>'n'</code> , the leading $n \times k$ part of array <code>A</code> must contain the matrix <code>A</code> , otherwise the leading $k \times n$ part of the array must contain the matrix <code>A</code> .
<code>lda</code>	leading dimension of <code>A</code> . When <code>trans == 'N'</code> or <code>'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>	double-precision array of dimensions (ldb, kb) , where $kb = k$ when <code>trans == 'N'</code> or <code>'n'</code> , and $k = n$ otherwise. When <code>trans == 'N'</code> or <code>'n'</code> , the leading $n \times k$ part of array <code>B</code> must contain the matrix <code>B</code> , otherwise the leading $k \times n$ part of the array must contain the matrix <code>B</code> .
<code>ldb</code>	leading dimension of <code>B</code> . When <code>trans == 'N'</code> or <code>'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>	double-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.
<code>C</code>	double-precision array of dimensions (ldc, n) . If <code>uplo == 'U'</code> or <code>'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'</code> or <code>'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

C updated according to
 $C = \alpha * A * B^T + \alpha * B * A^T + \beta * C$ *or*
 $C = \alpha * A^T * B + \alpha * B^T * A + \beta * C.$

Reference: <http://www.netlib.org/blas/zsyr2k.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_ARCH_MISMATCH	if function invoked on device that does not support double precision
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

Function `cublasZtrmm()`

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

performs one of the matrix-matrix operations

$B = \alpha * \text{op}(A) * B$ *or* $B = \alpha * B * \text{op}(A)$,
 where $\text{op}(A) = A$, $\text{op}(A) = A^T$, *or* $\text{op}(A) = A^H$;

α is a double-precision complex scalar; B is an $m \times n$ matrix consisting of double-precision complex elements; and A is a unit or non-unit, upper or lower triangular matrix consisting of double-precision complex elements.

Matrices A and B are stored in column-major format, and lda and ldb are the leading dimensions of the two-dimensional arrays that contain A and B , respectively.

Input

side	<p>specifies whether $\text{op}(A)$ multiplies B from the left or right.</p> <p>If <code>side == 'L' OR 'l'</code>, $B = \alpha * \text{op}(A) * B$.</p> <p>If <code>side == 'R' OR 'r'</code>, $B = \alpha * B * \text{op}(A)$.</p>
uplo	<p>specifies whether the matrix A is an upper or lower triangular matrix.</p> <p>If <code>uplo == 'U' OR 'u'</code>, A is an upper triangular matrix.</p> <p>If <code>uplo == 'L' OR 'l'</code>, A is a lower triangular matrix.</p>
transa	<p>specifies $\text{op}(A)$. If <code>transa == 'N' OR 'n'</code>, $\text{op}(A) = A$.</p> <p>If <code>transa == 'T' OR 't'</code>, $\text{op}(A) = A^T$.</p> <p>If <code>transa == 'C' OR 'c'</code>, $\text{op}(A) = A^H$.</p>
diag	<p>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.</p>
m	the number of rows of matrix B ; m must be at least zero.
n	the number of columns of matrix B ; n must be at least zero.
alpha	double-precision complex scalar multiplier applied to $\text{op}(A) * B$ or $B * \text{op}(A)$, respectively. If α is zero, no accesses are made to matrix A , and no read accesses are made to matrix B .
A	double-precision complex 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.
lda	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.
B	double-precision complex 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.
ldb	leading dimension of B ; ldb must be at least $\max(1, m)$.

Output

B updated according to $B = \alpha * \text{op}(A) * B$ or
 $B = \alpha * B * \text{op}(A)$.

Reference: <http://www.netlib.org/blas/ztrmm.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_ARCH_MISMATCH</code>	if function invoked on device that does not support double precision
<code>CUBLAS_STATUS_EXECUTION_FAILED</code>	if function failed to launch on GPU

Function `cublasZtrsm()`

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

solves one of the matrix equations

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

α is a double-precision complex scalar, and X and B are $m \times n$ matrices that consist of double-precision complex 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	<p>specifies whether $\text{op}(A)$ appears on the left or right of x:</p> <p>side == 'L' or 'l' indicates solve $\text{op}(A) * X = \alpha * B$;</p> <p>side == 'R' or 'r' indicates solve $X * \text{op}(A) = \alpha * B$.</p>
uplo	<p>specifies whether the matrix A is an upper or lower triangular matrix:</p> <p>uplo == 'U' or 'u' indicates A is an upper triangular matrix;</p> <p>uplo == 'L' or 'l' indicates A is a lower triangular matrix.</p>
transa	<p>specifies $\text{op}(A)$. If transa == 'N' or 'n', $\text{op}(A) = A$.</p> <p>If transa == 'T' or 't', $\text{op}(A) = A^T$.</p> <p>If transa == 'C' or 'c', $\text{op}(A) = A^H$.</p>
diag	<p>specifies whether or not A is a unit triangular matrix.</p> <p>If diag == 'U' or 'u', A is assumed to be unit triangular.</p> <p>If diag == 'N' or 'n', A is not assumed to be unit triangular.</p>
m	specifies the number of rows of B ; m must be at least zero.
n	specifies the number of columns of B ; n must be at least zero.
alpha	double-precision complex scalar multiplier applied to B . When alpha is zero, A is not referenced and B does not have to be a valid input.
A	double-precision complex array of dimensions (lda, k) , where k is m when side == 'L' or 'l' and is n when side == 'R' or 'r'. If uplo == 'U' or 'u', the leading $k \times k$ upper triangular part of the array A must contain the upper triangular matrix, and the strictly lower triangular matrix of A is not referenced. When uplo == 'L' or 'l', the leading $k \times k$ lower triangular part of the array A must contain the lower triangular matrix, and the strictly upper triangular part of A is not referenced. Note that when diag == 'U' or 'u', the diagonal elements of A are not referenced and are assumed to be unity.
lda	<p>leading dimension of the two-dimensional array containing A.</p> <p>When side == 'L' or 'l', lda must be at least $\max(1, m)$.</p> <p>When side == 'R' or 'r', lda must be at least $\max(1, n)$.</p>
B	double-precision complex array of dimensions (ldb, n) ; ldb must be at least $\max(1, m)$. The leading $m \times n$ part of the array B must contain the right-hand side matrix B . On exit, B is overwritten by the solution matrix x .
ldb	leading dimension of the two-dimensional array containing B ; ldb must be at least $\max(1, m)$.

Output

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

Reference: <http://www.netlib.org/blas/ztrsm.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_ARCH_MISMATCH	if function invoked on device that does not support double precision
CUBLAS_STATUS_EXECUTION_FAILED	if function failed to launch on GPU

CUBLAS Fortran Bindings

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

- ❑ symbol names (capitalization, name decoration)
- ❑ argument passing (by value or reference)
- ❑ passing of string arguments (length information)
- ❑ passing of pointer arguments (size of the pointer)
- ❑ returning 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, which are written in C and provided in two forms:

- ❑ the thunking wrapper interface in the file `fortran_thunking.c`
- ❑ the direct wrapper interface in the file `fortran.c`

The code of one of those two files must 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 those two C files has been used to demonstrate interoperability with the compilers g77 3.2.3 and g95 0.91 on 32-bit Linux, g77 3.4.5 and g95 0.91 on 64-bit Linux, Intel Fortran 9.0 and Intel Fortran 10.0 on 32-bit and 64-bit Microsoft Windows XP, and g77 3.4.0 and g95 0.92 on Mac OS X.

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.

The thunking wrappers allow interfacing to existing Fortran applications without any changes to the application. 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. To use the thunking wrappers, the application needs to be compiled with the file `fortran_thunking.c`.

The direct wrappers, intended for production code, substitute device pointers for vector and matrix arguments in all BLAS functions. To use these interfaces, existing applications must 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 space (with `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 and Mac OS X, 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 two examples show a small application implemented in Fortran 77 on the host ([Example A.1., “Fortran 77 Application Executing on the Host” on page 244](#)), and show the same application using the non-thunking wrappers after it has been ported to use CUBLAS ([Example A.2., “Fortran 77 Application Ported to Use CUBLAS” on page 245](#)).

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 (*,*) ""
  enddo
  stop
end
```

Example A.2. Fortran 77 Application Ported to Use CUBLAS

```

#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, cublas_set_matrix, cublas_get_matrix
      do j = 1, N
         do i = 1, M
            a(i,j) = (i-1) * M + j
         enddo
      enddo

      call cublas_init
      stat = cublas_alloc(M*N, sizeof_real, devPtrA)
      if (stat .NE. 0) then

```

Example A.2. Fortran 77 Application Ported to Use CUBLAS (continued)

```
      write(*,*) "device memory allocation failed"
      call cublas_shutdown
      stop
    endif
    stat = cublas_set_matrix (M, N, sizeof_real, a, M, devPtrA, M)
    if (stat .NE. 0) then
      call cublas_free (devPtrA)
      write(*,*) "data download failed"
      call cublas_shutdown
      stop
    endif
    call modify (devPtrA, M, N, 2, 3, 16.0, 12.0)
    stat = cublas_get_matrix (M, N, sizeof_real, devPtrA, M, a, M)
    if (stat .NE. 0) then
      call cublas_free (devPtrA)
      write(*,*) "data upload failed"
      call cublas_shutdown
      stop
    endif
    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
```
