NVIDIA® HPC Compilers

FORTRAN CUDA INTERFACES

PG-09864-001-V2021 | February 2021

TABLE OF CONTENTS

PREFACE

This document describes the NVIDIA Fortran interfaces to cuBLAS, cuFFT, cuRAND, and cuSPARSE, which are CUDA Libraries used in scientific and engineering applications built upon the CUDA computing architecture.

Intended Audience

This guide is intended for application programmers, scientists and engineers proficient in programming with the Fortran language. This guide assumes some familiarity with either CUDA Fortran or OpenACC.

Organization

The organization of this document is as follows:

Introduction
contains a general introduction to Fortran interfaces, OpenACC, CUDA Fortran, and CUDA Library functions

BLAS Runtime Library APIs
describes the Fortran interfaces to the various cuBLAS libraries

FFT Runtime Library APIs
describes the module types, definitions and Fortran interfaces to the cuFFT library

Random Number Runtime APIs
describes the Fortran interfaces to the host and device cuRAND libraries

Sparse Matrix Runtime APIs
describes the module types, definitions and Fortran interfaces to the cuSPARSE Library

Examples
provides sample code and an explanation of each of the simple examples.

Conventions

This guide uses the following conventions:

italic
is used for emphasis.

`Constant Width`
is used for filenames, directories, arguments, options, examples, and for language statements in the text, including assembly language statements.

Bold
is used for commands.

[item1]
in general, square brackets indicate optional items. In this case item1 is optional. In the context of p/t-sets, square brackets are required to specify a p/t-set.

{ item2 | item 3 }
braces indicate that a selection is required. In this case, you must select either item2 or item3.

filename ...
ellipsis indicate a repetition. Zero or more of the preceding item may occur. In this example, multiple filenames are allowed.

`FORTRAN`
Fortran language statements are shown in the text of this guide using a reduced fixed point size.

`C++ and C`
C++ and C language statements are shown in the test of this guide using a reduced fixed point size.

Terminology

If there are terms in this guide with which you are unfamiliar, see the NVIDIA HPC glossary at docs.nvidia.com/hpc-sdk/definitions.

Related Publications

The following documents contain additional information related to OpenACC and CUDA Fortran programming, CUDA, and the CUDA Libraries.

- ISO/IEC 1539-1:1997, Information Technology – Programming Languages – FORTRAN, Geneva, 1997 (Fortran 95).
- NVIDIA CUDA Programming Guide
- HPC Compilers User Guide, docs.nvidia.com/hpc-sdk/compilers/pdf/hpc212ug.pdf

Chapter 1. INTRODUCTION

This document provides a reference for calling CUDA Library functions from NVIDIA Fortran. It can be used from Fortran code using the OpenACC programming model, or from NVIDIA CUDA Fortran. Currently, the CUDA libraries which NVIDIA provides pre-built interface modules for, and which are documented here, are:

- cuBLAS, an implementation of the BLAS.
- cuFFT, a library of Fast Fourier Transform (FFT) routines.
- cuRAND, a library for random number generation.
- cuSPARSE, a library of linear algebra routines used with sparse matrices.

The OpenACC Application Program Interface is a collection of compiler directives and runtime routines that allows the programmer to specify loops and regions of code for offloading from a host CPU to an attached accelerator, such as a GPU. The OpenACC API was designed and is maintained by an industry consortium. See the OpenACC website, http://www.openacc.org for more information about the OpenACC API.

CUDA Fortran is a small set of extensions to Fortran that supports and is built upon the CUDA computing architecture. CUDA Fortran includes a Fortran 2003 compiler and tool chain for programming NVIDIA GPUs using Fortran, and is an analog to NVIDIA's CUDA C compiler. Compared to the NVIDIA Accelerator and OpenACC directives-based model and compilers, CUDA Fortran is a lower-level explicit programming model with substantial runtime library components that give expert programmers direct control of all aspects of GPGPU programming.

This document does not contain explanations or purposes of the library functions, nor does it contain details of the approach used in the CUDA implementation to target GPUs. For that information, please see the appropriate library document that comes with the NVIDIA CUDA Toolkit. This document does provide the Fortran module contents: derived types, enumerations, and interfaces, to make use of the libraries from Fortran rather than from C or C++.

Many of the examples used in this document are provided in the HPC compiler and tools distribution, along with Makefiles, and are stored in the yearly directory, such as `2016/CUDA-Libraries`.

1.1. Fortran Interfaces and Wrappers

Almost all of the function interfaces shown in this document make use of features from the Fortran 2003 `iso_c_binding` intrinsic module. This module provides a standard way for dealing with isues such as inter-language data types, capitalization, adding underscores to symbol names, or passing arguments by value.

Often, the `iso_c_binding` module enables Fortran programs containing properly written interfaces to call directly into the C library functions. In some cases, NVIDIA has written small wrappers around the C library function, to make the Fortran call site more "Fortran-like", hiding some issues exposed in the C interfaces like handle management, host vs. device pointer management, or character and complex data type issues.

In a small number of cases, the C Library may contain multiple entry points to handle different data types, perhaps an `int` in one function and a `size_t` in another, otherwise the functions are identical. In these cases, NVIDIA may provide just one generic Fortran interface, and will call the appropriate C function under the hood.

1.2. Using CUDA Libraries from OpenACC Host Code

All four of the libraries covered in this document contain functions which are callable from OpenACC host code. Most functions take some arguments which are expected to be device pointers (the address of a variable in device global memory). There are several ways to do that in OpenACC.

If the call is lexically nested within an OpenACC data directive, the NVIDIA Fortran compiler, in the presence of an explicit interface such as those provided by the NVIDIA library modules, will default to passing the device pointer when required.

```
subroutine hostcall(a, b, n)
use cublas
real a(n), b(n)
!$acc data copy(a, b)
call cublasSswap(n, a, 1, b, 1)
!$acc end data
return
end
```

A Fortran interface is made explicit when you `use` the module that contains it, as in the line `use cublas` in the example above. If you look ahead to the actual interface for cublasSswap, you will see that the arrays a and b are declared with the CUDA Fortran device attribute, so they take only device addresses as arguments.

It is more acceptable and general when using OpenACC to pass device pointers to subprograms by using the `host_data` clause as most implementations don't have a way to mark arguments as device pointers. The `host_data` construct with the `use_device` clause makes the device addresses available in host code for passing to the subprogram.

```
use cufft
```

```
use openacc
. . .
!$acc data copyin(a), copyout(b,c)
ierr = cufftPlan2D(iplan1,m,n,CUFFT_C2C)
ierr = ierr + cufftSetStream(iplan1,acc_get_cuda_stream(acc_async_sync))
!$acc host_data use_device(a,b,c)
ierr = ierr + cufftExecC2C(iplan1,a,b,CUFFT_FORWARD)
ierr = ierr + cufftExecC2C(iplan1,b,c,CUFFT_INVERSE)
!$acc end host_data

! scale c
!$acc kernels
c = c / (m*n)
!$acc end kernels
!$acc end data
```

This code snippet also shows an example of sharing the stream that OpenACC and the cuFFT library use. Every library in this document has a function for setting the CUDA stream which the library runs on. Usually, when using OpenACC, you want the OpenACC kernels to run on the same stream as the library functions. In the case above, this guarantees that the kernel **`c = c / (m*n)`** does not start until the FFT operations complete. The function `acc_get_cuda_stream` and the definition for `acc_async_sync` are in the openacc module.

1.3. Using CUDA Libraries from OpenACC Device Code

Two libraries are available from withn OpenACC compute regions, though the two can behave quite differently. Functions in both the openacc_cublas module and the openacc_curand module are marked **`acc routine seq`**. The cuBLAS interfaces from device code closely mirror what's available from the host, but the underlying implementation may launch a new kernel using CUDA dynamic parallelism. The routines should not be called by multiple threads if you expect the threads to cooperate together to compute the answer.

```
subroutine testdev( a, b, n )
use openacc_cublas
real :: a(n), b(n)
type(cublasHandle) :: h
!$acc parallel num_gangs(1) copy(a,b,h)
j = cublasCreate(h)
j = cublasSswap(h,n,a,1,b,1)
j = cublasDestroy(h)
!$acc end parallel
return
end subroutine
```

When using the openacc_cublas module, you must link with -lcublas_device (or defaultlib:cublas_device on Windows) and compile and link with -Mcuda.

The cuRAND device library is all contained within CUDA header files. In device code, it is designed to return one or a small number of random numbers per thread. The thread's random generators run independently of each other, and it is usually advised for performance reasons to give each thread a different seed, rather than a different offset.

```
program t
```

```
use openacc_curand
integer, parameter :: n = 500
real a(n,n,4)
type(curandStateXORWOW) :: h
integer(8) :: seed, seq, offset
a = 0.0
!$acc parallel num_gangs(n) vector_length(n) copy(a)
!$acc loop gang
do j = 1, n
!$acc loop vector private(h)
  do i = 1, n
    seed = 12345_8 + j*n*n + i*2
    seq = 0_8
    offset = 0_8
    call curand_init(seed, seq, offset, h)
!$acc loop seq
    do k = 1, 4
      a(i,j,k) = curand_uniform(h)
    end do
  end do
end do
!$acc end parallel
print *,maxval(a),minval(a),sum(a)/(n*n*4)
end
```

When using the openacc_curand module, since all the code is contained in CUDA header files, you do not need any additional libraries on the link line. However, since the current implementation relies on CUDA compilation, you must compile with -ta=tesla,nonvvm.

1.4. Using CUDA Libraries from CUDA Fortran Host Code

The predominant usage model for the library functions listed in this document is to call them from CUDA Host code. CUDA Fortran allows some special capabilities in that the compiler is able to recognize the device and managed attribute in resolving generic interfaces. Device actual arguments can only match the interface's device dummy arguments; managed actual arguments, by precedence, match managed dummy arguments first, then device dummies, then host.

```
program testisamax  ! link with -Mcudalib=cublas -lblas
use cublas
real*4              x(1000)
real*4, device  :: xd(1000)
real*4, managed :: xm(1000)

call random_number(x)

! Call host BLAS
j = isamax(1000,x,1)

xd = x
! Call cuBLAS
k = isamax(1000,xd,1)
print *,j.eq.k

xm = x
! Also calls cuBLAS
k = isamax(1000,xm,1)
print *,j.eq.k
```

```
end
```

Using the **cudafor** module, the full set of CUDA functionality is available to programmers for managing CUDA events, streams, synchronization, and asynchronous behaviors. CUDA Fortran can be used in OpenMP programs, and the CUDA Libraries in this document are thread safe with respect to host CPU threads. Further examples are included in chapter Examples.

1.5. Using CUDA Libraries from CUDA Fortran Device Code

The cuBLAS and cuRAND libraries have functions callable from CUDA Fortran device code, and their interfaces are accessed via the `cublas_device` and `curand_device` modules, respectively. The module interfaces are very similar to the modules used in OpenACC device code, but for CUDA Fortran, each subroutine and function is declared attributes(device), and the subroutines and functions do not need to be marked as **acc routine seq**.

```
! cuBLAS in device code requires -Mcuda=cc35 or higher
! since it potentially uses dynamic parallelism to launch kernels.
! pgfortran -Mcuda=cc35 testcu.cuf -lcublas_device
attributes(global) subroutine testcu( a, b, n )
use cublas_device
real, device :: a(*), b(*)
type(cublasHandle) :: h
integer, value :: n
i = threadIdx%x
if (i.eq.1) then
    j = cublasCreate(h)
    j = cublasSswap(h,n,a,1,b,1)
    j = cublasDestroy(h)
end if
return
end subroutine
```

Using the device cuRAND library with CUDA Fortran also requires compiling with -Mcuda=nonvvm so the CUDA in the cuRAND headers can get compiled.

```
module mrand
    use curand_device
    integer, parameter :: n = 500
    contains
 attributes(global) subroutine randsub(a)
 real, device :: a(n,n,4)
 type(curandStateXORWOW) :: h
 integer(8) :: seed, seq, offset
 j = blockIdx%x; i = threadIdx%x
 seed = 12345_8 + j*n*n + i*2
 seq = 0_8
 offset = 0_8
 call curand_init(seed, seq, offset, h)
 do k = 1, 4
     a(i,j,k) = curand_uniform(h)
 end do
 end subroutine
end module

program t   ! pgfortran -Mcuda=nonvvm t.cuf
use mrand
use cudafor ! recognize maxval, minval, sum w/managed
```

```
real, managed :: a(n,n,4)
a = 0.0
call randsub<<<n,n>>>(a)
print *,maxval(a),minval(a),sum(a)/(n*n*4)
end program
```

1.6. Pointer Modes in cuBLAS and cuSPARSE

Because the NVIDIA Fortran compiler can distinguish between host and device arguments, the NVIDIA modules for interfacing to cuBLAS and cuSPARSE handle pointer modes differently than CUDA C, which requires setting the mode explicitly for scalar arguments. Examples of scalar arguments which can reside either on the host or device are the alpha and beta scale factors to the *gemm functions.

Typically, when using the normal "non-_v2" interfaces in the cuBLAS and cuSPARSE modules, the runtime wrappers will implicitly add the setting and restoring of the library pointer modes behind the scenes. This adds some negligible but non-zero overhead to the calls.

To avoid the implicit getting and setting of the pointer mode with every invocation of a library function do the following:

- For the BLAS, use the **`cublas_v2`** module, and the v2 entry points, such as **`cublasIsamax_v2`**. It is the programmer's responsibility to properly set the pointer mode when needed. Examples of scalar arguments which do require setting the pointer mode are the alpha and beta scale factors passed to the *gemm routines, and the scalar results returned from the v2 versions of the *amax(), *amin(), *asum(), *rotg(), *rotmg(), *nrm2(), and *dot() functions. In the v2 interfaces shown in the chapter 2, these scalar arguments will have the comment **`! device or host variable`**. Examples of scalar arguments which do not require setting the pointer mode are increments, extents, and lengths such as incx, incy, n, lda, ldb, and ldc.
- For the cuSPARSE library, each function listed in chapter 5 which contains scalar arguments with the comment **`! device or host variable`** has a corresponding v2 interface, though it is not documented here. For instance, in addition to the interface named **`cusparseSaxpyi`**, there is another interface named **`cusparseSaxpyi_v2`** with the exact same argument list which calls into the cuSPARSE library directly and will not implicitly get or set the library pointer mode.

The CUDA default pointer mode is that the scalar arguments reside on the host. The NVIDIA runtime does not change that setting.

1.7. Writing Your Own CUDA Interfaces

Despite the large number of interfaces included in the modules described in this document, users will have the need from time-to-time to write their own interfaces to new libraries or their own tuned CUDA, perhaps written in C/C++. There are some standard techniques to use, and some non-standard NVIDIA extensions which can make creating working interfaces easier.

```
! cufftExecC2C
interface cufftExecC2C
```

```
    integer function cufftExecC2C( plan, idata, odata, direction )
 bind(C,name='cufftExecC2C')
        integer, value :: plan
        complex, device, dimension(*) :: idata, odata
        integer, value :: direction
    end function cufftExecC2C
end interface cufftExecC2C
```

This interface calls the C library function directly. You can deal with Fortran's capitalization issues by putting the properly capitalized C function in the **bind(C)** attribute. If the C function expects input arguments passed by value, you can add the **value** attribute to the dummy declaration as well. A nice feature of Fortran is that the interface can change, but the code at the call site may not have to. The compiler changes the details of the call to fit the interface.

Now suppose a user of this interface would like to call this function with REAL data (F77 code is notorious for mixing REAL and COMPLEX declarations). There are two ways to do this:

```
! cufftExecC2C
interface cufftExecC2C
    integer function cufftExecC2C( plan, idata, odata, direction )
 bind(C,name='cufftExecC2C')
        integer, value :: plan
        complex, device, dimension(*) :: idata, odata
        integer, value :: direction
    end function cufftExecC2C
    integer function cufftExecR2R( plan, idata, odata, direction )
 bind(C,name='cufftExecC2C')
        integer, value :: plan
        real, device, dimension(*) :: idata, odata
        integer, value :: direction
    end function cufftExecR2R
end interface cufftExecC2C
```

Here the C name hasn't changed. The compiler will now accept actual arguments corresponding to idata and odata that are declared REAL. A generic interface is created named **cufftExecC2C**. If you have problems debugging your generic interface, as a debugging aid you can try calling the specific name, **cufftExecR2R** in this case, to help diagnose the problem.

A commonly used extension which is supported by NVIDIA is **ignore_tkr**. A programmer can use it in an interface to instruct the compiler to ignore any combination of the type, kind, and rank during the interface matching process. The previous example using **ignore_tkr** looks like this:

```
! cufftExecC2C
interface cufftExecC2C
    integer function cufftExecC2C( plan, idata, odata, direction )
 bind(C,name='cufftExecC2C')
        integer, value :: plan
        !dir$ ignore_tkr(tr) idata, (tr) odata
        complex, device, dimension(*) :: idata, odata
        integer, value :: direction
    end function cufftExecC2C
end interface cufftExecC2C
```

Now the compiler will ignore both the type and rank (F77 could also be sloppy in its handling of array dimensions) of idata and odata when matching the call site to the interface. An unfortunate side-effect is that the interface will now allow integer, logical,

and character data for idata and odata. It is up to the implementor to determine if that is acceptable.

A final aid, specific to NVIDIA, worth mentioning here is **`ignore_tkr (d)`**, which ignores the device attribute of an actual argument during interface matching.

Of course, if you write a wrapper, a narrow strip of code between the Fortran call and your library function, you are not limited by the simple transormations that a compiler can do, such as those listed here. As mentioned earlier, many of the interfaces provided in the cuBLAS and cuSPARSE modules use wrappers.

A common request is a way for Fortran programmers to take advantage of the thrust library. Explaining thrust and C++ programming is outside of the scope of this document, but this simple example can show how to take advantage of the excellent sort capabilities in thrust:

```
// Filename: csort.cu
// nvcc -c -arch sm_35 csort.cu
#include <thrust/device_vector.h>
#include <thrust/copy.h>
#include <thrust/sort.h>

extern "C" {

    //Sort for integer arrays
    void thrust_int_sort_wrapper( int *data, int N)
    {
    thrust::device_ptr <int> dev_ptr(data);
    thrust::sort(dev_ptr, dev_ptr+N);
    }

    //Sort for float arrays
    void thrust_float_sort_wrapper( float *data, int N)
    {
    thrust::device_ptr <float> dev_ptr(data);
    thrust::sort(dev_ptr, dev_ptr+N);
    }

    //Sort for double arrays
    void thrust_double_sort_wrapper( double *data, int N)
    {
    thrust::device_ptr <double> dev_ptr(data);
    thrust::sort(dev_ptr, dev_ptr+N);
    }
}
```

Set up interface to the sort subroutine in Fortran and calls are simple:

```
program t
interface sort
   subroutine sort_int(array, n) &
        bind(C,name='thrust_int_sort_wrapper')
        integer(4), device, dimension(*) :: array
        integer(4), value :: n
    end subroutine
end interface
integer(4), parameter :: n = 100
integer(4), device :: a_d(n)
integer(4) :: a_h(n)
!$cuf kernel do
do i = 1, n
    a_d(i) = 1 + mod(47*i,n)
end do
call sort(a_d, n)
a_h = a_d
```

```
nres  = count(a_h .eq. (/(i,i=1,n)/))
if (nres.eq.n) then
    print *,"test PASSED"
else
    print *,"test FAILED"
endif
end
```

1.8. NVIDIA Fortran Compiler Options

The NVIDIA Fortran compiler driver is called `pgfortran`. General information on the compiler options which can be passed to pgfortran can be obtained by typing `pgfortran -help`. To enable targeting NVIDIA GPUs using OpenACC, use `pgfortran -ta=tesla`. To enable targeting NVIDIA GPUs using CUDA Fortran, use `pgfortran -Mcuda`. CUDA Fortran is also supported by the NVIDIA Fortran compilers when the filename uses the `.cuf` extension. Uppercase file extensions, `.F90` or `.CUF`, for example, may also be used, in which case the program is processed by the preprocessor before being compiled.

Other options which are pertinent to the examples in this document are:

- -Mcudalib[=cublas|cufft|curand|cusparse]: this option adds the appropriate versions of the CUDA-optimized libraries to the link line. It handles static and dynamic linking, and platform (Linux, Windows) differences unobtrusively.
- -Mcuda=[no]nvvm: this option chooses between two choices for the compiler back-end code generator. Currently, using the cuRAND library from device code requires -Mcuda=nonvvm.
- -ta=tesla:cc35: this option compiles for compute capability 3.5. Certain device functionality, such as dynamic parallelism in the cuBLAS library, requires compute capability 3.5 or higher.
- -lcublas_device: this adds the cuBLAS device library to set of linker options. On Windows, use -defaultlib:cublas_device.

Chapter 2. BLAS RUNTIME APIS

This section describes the Fortran interfaces to the CUDA BLAS libraries. There are currently four somewhat separate collections of function entry points which are commonly referred to as the cuBLAS:

- The original CUDA implementation of the BLAS routines, referred to as the legacy API, which are callable from the host and expect and operate on device data.
- The newer "v2" CUDA implementation of the BLAS routines, plus some extensions for batched operations. These are also callable from the host and operate on device data. In Fortran terms, these entry points have been changed from subroutines to functions which return status.
- Another implementation of the BLAS routines using the v2 entry points, callable from device code and which may take advantage of dynamic parallelism.
- A new cuBLAS XT library which can target multiple GPUs using only host-resident data.

NVIDIA currently ships with five Fortran modules which programmers can use to call into this cuBLAS functionality:

- cublas, which provides interfaces to into the main cublas library. Both the legacy and v2 names are supported. In this module, the cublas names (such as cublasSaxpy) use the legacy calling conventions. Interfaces to a host BLAS library (for instance libblas.a in the NVIDIA distribution) are also included in the cublas module.
- cublas_v2, which is similar to the cublas module in most ways except the cublas names (such as cublasSaxpy) use the v2 calling conventions. For instance, instead of a subroutine, cublasSaxpy is a function which takes a handle as the first argument and returns an integer containing the status of the call.
- cublasxt, which interfaces directly to the cublasXT API.
- cublas_device, which is useable from CUDA Fortran device code and interfaces into the static cuBLAS Library cublas_device.a. The legacy cuBLAS API is not supported in this library or module.
- openacc_cublas, which is useable from OpenACC device code and also provides interfaces into cublas_device.a. For convenience, this module marks each function as "!$acc routine seq". The legacy cuBLAS API is not supported in this library or module.

The v2 routines are integer functions that return an error status code; they return a value of CUBLAS_STATUS_SUCCESS if the call was successful, or other cuBLAS status return value if there was an error.

Documented interfaces to the traditional BLAS names in the subsequent sections, which contain the comment **`! device or host variable`** should not be confused with the pointer mode issue from section 1.6. The traditional BLAS names are overloaded generic names in the **`cublas`** module. For instance, in this interface

```
 subroutine scopy(n, x, incx, y, incy)
   integer :: n
   real(4), device, dimension(*) :: x, y ! device or host variable
   integer :: incx, incy
```

The arrays x and y can either both be device arrays, in which case **`cublasScopy`** is called via the generic interface, or they can both be host arrays, in which case **`scopy`** from the host BLAS library is called. Using CUDA Fortran managed data as actual arguments to **`scopy`** poses an interesting case, and calling **`cublasScopy`** is chosen by default. If you wish to call the host library version of scopy with managed data, don't expose the generic scopy interface at the call site.

Unless a specific kind is provided, in the following interfaces the plain integer type implies integer(4) and the plain real type implies real(4).

2.1. CUBLAS Definitions and Helper Functions

This section contains definitions and data types used in the cuBLAS library and interfaces to the cuBLAS Helper Functions.

The cublas module contains the following derived type definitions:

```
TYPE cublasHandle
  TYPE(C_PTR)  :: handle
END TYPE
```

The cuBLAS module contains the following enumerations:

```
enum, bind(c)
    enumerator :: CUBLAS_STATUS_SUCCESS         =0
    enumerator :: CUBLAS_STATUS_NOT_INITIALIZED =1
    enumerator :: CUBLAS_STATUS_ALLOC_FAILED    =3
    enumerator :: CUBLAS_STATUS_INVALID_VALUE   =7
    enumerator :: CUBLAS_STATUS_ARCH_MISMATCH   =8
    enumerator :: CUBLAS_STATUS_MAPPING_ERROR   =11
    enumerator :: CUBLAS_STATUS_EXECUTION_FAILED=13
    enumerator :: CUBLAS_STATUS_INTERNAL_ERROR  =14
end enum
```

```
enum, bind(c)
    enumerator :: CUBLAS_FILL_MODE_LOWER=0
    enumerator :: CUBLAS_FILL_MODE_UPPER=1
end enum
```

```
enum, bind(c)
    enumerator :: CUBLAS_DIAG_NON_UNIT=0
    enumerator :: CUBLAS_DIAG_UNIT=1
end enum
```

```
enum, bind(c)
    enumerator :: CUBLAS_SIDE_LEFT =0
    enumerator :: CUBLAS_SIDE_RIGHT=1
```

```
end enum
```

```
enum, bind(c)
    enumerator :: CUBLAS_OP_N=0
    enumerator :: CUBLAS_OP_T=1
    enumerator :: CUBLAS_OP_C=2
end enum
```

```
enum, bind(c)
    enumerator :: CUBLAS_POINTER_MODE_HOST   = 0
    enumerator :: CUBLAS_POINTER_MODE_DEVICE = 1
end enum
```

2.1.1. cublasCreate

This function initializes the CUBLAS library and creates a handle to an opaque structure holding the CUBLAS library context. It allocates hardware resources on the host and device and must be called prior to making any other CUBLAS library calls. The CUBLAS library context is tied to the current CUDA device. To use the library on multiple devices, one CUBLAS handle needs to be created for each device. Furthermore, for a given device, multiple CUBLAS handles with different configuration can be created. Because cublasCreate allocates some internal resources and the release of those resources by calling cublasDestroy will implicitly call cublasDeviceSynchronize, it is recommended to minimize the number of cublasCreate/cublasDestroy occurences. For multi-threaded applications that use the same device from different threads, the recommended programming model is to create one CUBLAS handle per thread and use that CUBLAS handle for the entire life of the thread.

```
integer(4) function cublasCreate(handle)
  type(cublasHandle) :: handle
```

2.1.2. cublasDestroy

This function releases hardware resources used by the CUBLAS library. This function is usually the last call with a particular handle to the CUBLAS library. Because cublasCreate allocates some internal resources and the release of those resources by calling cublasDestroy will implicitly call cublasDeviceSynchronize, it is recommended to minimize the number of cublasCreate/cublasDestroy occurences.

```
integer(4) function cublasDestroy(handle)
  type(cublasHandle) :: handle
```

2.1.3. cublasGetVersion

This function returns the version number of the cuBLAS library.

```
integer(4) function cublasGetVersion(handle, version)
  type(cublasHandle) :: handle
  integer(4) :: version
```

2.1.4. cublasSetStream

This function sets the cuBLAS library stream, which will be used to execute all subsequent calls to the cuBLAS library functions. If the cuBLAS library stream is not set, all kernels use the default NULL stream. In particular, this routine can be used to change

the stream between kernel launches and then to reset the cuBLAS library stream back to NULL.

```
integer(4) function cublasSetStream(handle, stream)
  type(cublasHandle) :: handle
  integer(kind=cuda_stream_kind()) :: stream
```

2.1.5. cublasGetStream

This function gets the cuBLAS library stream, which is being used to execute all calls to the cuBLAS library functions. If the cuBLAS library stream is not set, all kernels use the default NULL stream.

```
integer(4) function cublasGetStream(handle, stream)
  type(cublasHandle) :: handle
  integer(kind=cuda_stream_kind()) :: stream
```

2.1.6. cublasGetPointerMode

This function obtains the pointer mode used by the cuBLAS library. In the **`cublas`** module, the pointer mode is set and reset on a call-by-call basis depending on the whether the device attribute is set on scalar actual arguments. See section 1.6 for a discussion of pointer modes.

```
integer(4) function cublasGetPointerMode(handle, mode)
  type(cublasHandle) :: handle
  integer(4) :: mode
```

2.1.7. cublasSetPointerMode

This function sets the pointer mode used by the cuBLAS library. When using the **`cublas`** module, the pointer mode is set on a call-by-call basis depending on the whether the device attribute is set on scalar actual arguments. When using the **`cublas_v2`** module with v2 interfaces, it is the programmer's responsibility to make calls to **`cublasSetPointerMode`** so scalar arguments are handled correctly by the library. See section 1.6 for a discussion of pointer modes.

```
integer(4) function cublasSetPointerMode(handle, mode)
  type(cublasHandle) :: handle
  integer(4) :: mode
```

2.1.8. cublasGetAtomicsMode

This function obtains the atomics mode used by the cuBLAS library.

```
integer(4) function cublasGetAtomicsMode(handle, mode)
  type(cublasHandle) :: handle
  integer(4) :: mode
```

2.1.9. cublasSetAtomicsMode

This function sets the atomics mode used by the cuBLAS library. Some routines in the cuBLAS library have alternate implementations that use atomics to accumulate results. These alternate implementations may run faster but may also generate results which are not identical from one run to the other. The default is to not allow atomics in cuBLAS functions.

```
integer(4) function cublasSetAtomicsMode(handle, mode)
```

```
  type(cublasHandle) :: handle
  integer(4) :: mode
```

2.1.10. cublasGetHandle

This function gets the cuBLAS handle currently in use by a thread. The CUDA Fortran runtime keeps track of a CPU thread's current handle, if you are either using the legacy BLAS API, or do not wish to pass the handle through to low-level functions or subroutines manually.

```
type(cublashandle) function cublasGetHandle()
```

```
integer(4) function cublasGetHandle(handle)
  type(cublasHandle) :: handle
```

2.1.11. cublasSetVector

This function copies n elements from a vector x in host memory space to a vector y in GPU memory space. It is assumed that each element requires storage of elemSize bytes. In CUDA Fortran, the type of vector x and y is overloaded to take any data type, but the size of the data type must still be specified in bytes. This functionality can also be implemented using cudaMemcpy or array assignment statements.

```
integer(4) function cublassetvector(n, elemsize, x, incx, y, incy)
  integer :: n, elemsize, incx, incy
  integer*1, dimension(*) :: x
  integer*1, device, dimension(*) :: y
```

2.1.12. cublasGetVector

This function copies n elements from a vector x in GPU memory space to a vector y in host memory space. It is assumed that each element requires storage of elemSize bytes. In CUDA Fortran, the type of vector x and y is overloaded to take any data type, but the size of the data type must still be specified in bytes. This functionality can also be implemented using cudaMemcpy or array assignment statements.

```
integer(4) function cublasgetvector(n, elemsize, x, incx, y, incy)
  integer :: n, elemsize, incx, incy
  integer*1, device, dimension(*) :: x
  integer*1, dimension(*) :: y
```

2.1.13. cublasSetMatrix

This function copies a tile of rows x cols elements from a matrix A in host memory space to a matrix B in GPU memory space. It is assumed that each element requires storage of elemSize bytes. In CUDA Fortran, the type of Matrix A and B is overloaded to take any data type, but the size of the data type must still be specified in bytes. This functionality can also be implemented using cudaMemcpy, cudaMemcpy2D, or array assignment statements.

```
integer(4) function cublassetmatrix(rows, cols, elemsize, a, lda, b, ldb)
  integer :: rows, cols, elemsize, lda, ldb
  integer*1, dimension(lda, *) :: a
  integer*1, device, dimension(ldb, *) :: b
```

2.1.14. cublasGetMatrix

This function copies a tile of rows x cols elements from a matrix A in GPU memory space to a matrix B in host memory space. It is assumed that each element requires storage of elemSize bytes. In CUDA Fortran, the type of Matrix A and B is overloaded to take any data type, but the size of the data type must still be specified in bytes. This functionality can also be implemented using cudaMemcpy, cudaMemcpy2D, or array assignment statements.

```
integer(4) function cublasgetmatrix(rows, cols, elemsize, a, lda, b, ldb)
  integer :: rows, cols, elemsize, lda, ldb
  integer*1, device, dimension(lda, *) :: a
  integer*1, dimension(ldb, *) :: b
```

2.1.15. cublasSetVectorAsync

This function copies n elements from a vector x in host memory space to a vector y in GPU memory space, asynchronously, on the given CUDA stream. It is assumed that each element requires storage of elemSize bytes. In CUDA Fortran, the type of vector x and y is overloaded to take any data type, but the size of the data type must still be specified in bytes. This functionality can also be implemented using cudaMemcpyAsync.

```
integer(4) function cublassetvectorasync(n, elemsize, x, incx, y, incy, stream)
  integer :: n, elemsize, incx, incy
  integer*1, dimension(*) :: x
  integer*1, device, dimension(*) :: y
  integer(kind=cuda_stream_kind()) :: stream
```

2.1.16. cublasGetVectorAsync

This function copies n elements from a vector x in host memory space to a vector y in GPU memory space, asynchronously, on the given CUDA stream. It is assumed that each element requires storage of elemSize bytes. In CUDA Fortran, the type of vector x and y is overloaded to take any data type, but the size of the data type must still be specified in bytes. This functionality can also be implemented using cudaMemcpyAsync.

```
integer(4) function cublasgetvectorasync(n, elemsize, x, incx, y, incy, stream)
  integer :: n, elemsize, incx, incy
  integer*1, device, dimension(*) :: x
  integer*1, dimension(*) :: y
  integer(kind=cuda_stream_kind()) :: stream
```

2.1.17. cublasSetMatrixAsync

This function copies a tile of rows x cols elements from a matrix A in host memory space to a matrix B in GPU memory space, asynchronously using the specified stream. It is assumed that each element requires storage of elemSize bytes. In CUDA Fortran, the type of Matrix A and B is overloaded to take any data type, but the size of the data type must still be specified in bytes. This functionality can also be implemented using cudaMemcpyAsync or cudaMemcpy2DAsync.

```
integer(4) function cublassetmatrixasync(rows, cols, elemsize, a, lda, b, ldb,
stream)
  integer :: rows, cols, elemsize, lda, ldb
  integer*1, dimension(lda, *) :: a
  integer*1, device, dimension(ldb, *) :: b
```

```
  integer(kind=cuda_stream_kind()) :: stream
```

2.1.18. cublasGetMatrixAsync

This function copies a tile of rows x cols elements from a matrix A in GPU memory space to a matrix B in host memory space, asynchronously, using the specified stream. It is assumed that each element requires storage of elemSize bytes. In CUDA Fortran, the type of Matrix A and B is overloaded to take any data type, but the size of the data type must still be specified in bytes. This functionality can also be implemented using cudaMemcpyAsync or cudaMemcpy2DAsync.

```
integer(4) function cublasgetmatrixasync(rows, cols, elemsize, a, lda, b, ldb,
stream)
  integer :: rows, cols, elemsize, lda, ldb
  integer*1, device, dimension(lda, *) :: a
  integer*1, dimension(ldb, *) :: b
  integer(kind=cuda_stream_kind()) :: stream
```

2.2. Single Precision Functions and Subroutines

This section contains interfaces to the single precision BLAS and cuBLAS functions and subroutines.

2.2.1. isamax

ISAMAX finds the index of the element having the maximum absolute value.

```
integer(4) function isamax(n, x, incx)
  integer :: n
  real(4), device, dimension(*) :: x ! device or host variable
  integer :: incx
```

```
integer(4) function cublasIsamax(n, x, incx)
  integer :: n
  real(4), device, dimension(*) :: x
  integer :: incx
```

```
integer(4) function cublasIsamax_v2(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  real(4), device, dimension(*) :: x
  integer :: incx
  integer, device :: res ! device or host variable
```

2.2.2. isamin

ISAMIN finds the index of the element having the minimum absolute value.

```
integer(4) function isamin(n, x, incx)
  integer :: n
  real(4), device, dimension(*) :: x ! device or host variable
  integer :: incx
```

```
integer(4) function cublasIsamin(n, x, incx)
  integer :: n
  real(4), device, dimension(*) :: x
  integer :: incx
```

```
integer(4) function cublasIsamin_v2(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
```

```
  real(4), device, dimension(*) :: x
  integer :: incx
  integer, device :: res ! device or host variable
```

2.2.3. sasum

SASUM takes the sum of the absolute values.

```
real(4) function sasum(n, x, incx)
  integer :: n
  real(4), device, dimension(*) :: x ! device or host variable
  integer :: incx
```

```
real(4) function cublasSasum(n, x, incx)
  integer :: n
  real(4), device, dimension(*) :: x
  integer :: incx
```

```
integer(4) function cublasSasum_v2(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  real(4), device, dimension(*) :: x
  integer :: incx
  real(4), device :: res ! device or host variable
```

2.2.4. saxpy

SAXPY constant times a vector plus a vector.

```
subroutine saxpy(n, a, x, incx, y, incy)
  integer :: n
  real(4), device :: a ! device or host variable
  real(4), device, dimension(*) :: x, y ! device or host variable
  integer :: incx, incy
```

```
subroutine cublasSaxpy(n, a, x, incx, y, incy)
  integer :: n
  real(4), device :: a ! device or host variable
  real(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

```
integer(4) function cublasSaxpy_v2(h, n, a, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  real(4), device :: a ! device or host variable
  real(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

2.2.5. scopy

SCOPY copies a vector, x, to a vector, y.

```
subroutine scopy(n, x, incx, y, incy)
  integer :: n
  real(4), device, dimension(*) :: x, y ! device or host variable
  integer :: incx, incy
```

```
subroutine cublasScopy(n, x, incx, y, incy)
  integer :: n
  real(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

```
integer(4) function cublasScopy_v2(h, n, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  real(4), device, dimension(*) :: x, y
```

```
  integer :: incx, incy
```

2.2.6. sdot

SDOT forms the dot product of two vectors.

```
real(4) function sdot(n, x, incx, y, incy)
  integer :: n
  real(4), device, dimension(*) :: x, y ! device or host variable
  integer :: incx, incy
```

```
real(4) function cublasSdot(n, x, incx, y, incy)
  integer :: n
  real(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

```
integer(4) function cublasSdot_v2(h, n, x, incx, y, incy, res)
  type(cublasHandle) :: h
  integer :: n
  real(4), device, dimension(*) :: x, y
  integer :: incx, incy
  real(4), device :: res ! device or host variable
```

2.2.7. snrm2

SNRM2 returns the euclidean norm of a vector via the function name, so that SNRM2 := sqrt(x'*x).

```
real(4) function snrm2(n, x, incx)
  integer :: n
  real(4), device, dimension(*) :: x ! device or host variable
  integer :: incx
```

```
real(4) function cublasSnrm2(n, x, incx)
  integer :: n
  real(4), device, dimension(*) :: x
  integer :: incx
```

```
integer(4) function cublasSnrm2_v2(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  real(4), device, dimension(*) :: x
  integer :: incx
  real(4), device :: res ! device or host variable
```

2.2.8. srot

SROT applies a plane rotation.

```
subroutine srot(n, x, incx, y, incy, sc, ss)
  integer :: n
  real(4), device :: sc, ss ! device or host variable
  real(4), device, dimension(*) :: x, y ! device or host variable
  integer :: incx, incy
```

```
subroutine cublasSrot(n, x, incx, y, incy, sc, ss)
  integer :: n
  real(4), device :: sc, ss ! device or host variable
  real(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

```
integer(4) function cublasSrot_v2(h, n, x, incx, y, incy, sc, ss)
  type(cublasHandle) :: h
  integer :: n
  real(4), device :: sc, ss ! device or host variable
  real(4), device, dimension(*) :: x, y
```

```
  integer :: incx, incy
```

2.2.9. srotg

SROTG constructs a Givens plane rotation.

```
subroutine srotg(sa, sb, sc, ss)
  real(4), device :: sa, sb, sc, ss ! device or host variable
```

```
subroutine cublasSrotg(sa, sb, sc, ss)
  real(4), device :: sa, sb, sc, ss ! device or host variable
```

```
integer(4) function cublasSrotg_v2(h, sa, sb, sc, ss)
  type(cublasHandle) :: h
  real(4), device :: sa, sb, sc, ss ! device or host variable
```

2.2.10. srotm

SROTM applies the modified Givens transformation, H, to the 2 by N matrix (SX**T) , where **T indicates transpose. The elements of SX are in (SX**T) SX(LX+I*INCX), I = 0 to N-1, where LX = 1 if INCX .GE. 0, ELSE LX = (-INCX)*N, and similarly for SY using LY and INCY. With SPARAM(1)=SFLAG, H has one of the following forms.. SFLAG=-1.E0 SFLAG=0.E0 SFLAG=1.E0 SFLAG=-2.E0 (SH11 SH12) (1.E0 SH12) (SH11 1.E0) (1.E0 0.E0) H=( ) ( ) ( ) ( ) (SH21 SH22), (SH21 1.E0), (-1.E0 SH22), (0.E0 1.E0). See SROTMG for a description of data storage in SPARAM.

```
subroutine srotm(n, x, incx, y, incy, param)
  integer :: n
  real(4), device, dimension(*) :: x, y ! device or host variable
  integer :: incx, incy
```

```
subroutine cublasSrotm(n, x, incx, y, incy, param)
  integer :: n
  real(4), device, dimension(*) :: x, y
  integer :: incx, incy
  real(4), device :: param(*) ! device or host variable
```

```
integer(4) function cublasSrotm_v2(h, n, x, incx, y, incy, param)
  type(cublasHandle) :: h
  integer :: n
  real(4), device :: param(*) ! device or host variable
  real(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

2.2.11. srotmg

SROTMG constructs the modified Givens transformation matrix H which zeros the second component of the 2-vector (SQRT(SD1)*SX1,SQRT(SD2)*SY2)**T. With SPARAM(1)=SFLAG, H has one of the following forms.. **SFLAG=-1.E0 SFLAG=0.E0 SFLAG=1.E0 SFLAG=-2.E0 (SH11 SH12) (1.E0 SH12) (SH11 1.E0) (1.E0 0.E0) H=( ) ( ) ( ) ( ) (SH21 SH22), (SH21 1.E0), (-1.E0 SH22), (0.E0 1.E0).** Locations 2-4 of SPARAM contain SH11,SH21,SH12, and SH22 respectively. (Values of 1.E0, -1.E0, or 0.E0 implied by the value of SPARAM(1) are not stored in SPARAM.)

```
subroutine srotmg(d1, d2, x1, y1, param)
  real(4), device :: d1, d2, x1, y1, param(*) ! device or host variable
```

```
subroutine cublasSrotmg(d1, d2, x1, y1, param)
```

```
  real(4), device :: d1, d2, x1, y1, param(*) ! device or host variable
```

```
integer(4) function cublasSrotmg_v2(h, d1, d2, x1, y1, param)
  type(cublasHandle) :: h
  real(4), device :: d1, d2, x1, y1, param(*) ! device or host variable
```

2.2.12. sscal

SSCAL scales a vector by a constant.

```
subroutine sscal(n, a, x, incx)
  integer :: n
  real(4), device :: a ! device or host variable
  real(4), device, dimension(*) :: x ! device or host variable
  integer :: incx
```

```
subroutine cublasSscal(n, a, x, incx)
  integer :: n
  real(4), device :: a ! device or host variable
  real(4), device, dimension(*) :: x
  integer :: incx
```

```
integer(4) function cublasSscal_v2(h, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: n
  real(4), device :: a ! device or host variable
  real(4), device, dimension(*) :: x
  integer :: incx
```

2.2.13. sswap

SSWAP interchanges two vectors.

```
subroutine sswap(n, x, incx, y, incy)
  integer :: n
  real(4), device, dimension(*) :: x, y ! device or host variable
  integer :: incx, incy
```

```
subroutine cublasSswap(n, x, incx, y, incy)
  integer :: n
  real(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

```
integer(4) function cublasSswap_v2(h, n, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  real(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

2.2.14. sgbmv

SGBMV performs one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A**T*x + beta*y, where alpha and beta are scalars, x and y are vectors and A is an m by n band matrix, with kl sub-diagonals and ku super-diagonals.

```
subroutine sgbmv(t, m, n, kl, ku, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: t
  integer :: m, n, kl, ku, lda, incx, incy
  real(4), device, dimension(lda, *) :: a ! device or host variable
  real(4), device, dimension(*) :: x, y ! device or host variable
  real(4), device :: alpha, beta ! device or host variable
```

```
subroutine cublasSgbmv(t, m, n, kl, ku, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: t
  integer :: m, n, kl, ku, lda, incx, incy
  real(4), device, dimension(lda, *) :: a
```

```
  real(4), device, dimension(*) :: x, y
  real(4), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasSgbmv_v2(h, t, m, n, kl, ku, alpha, a, lda, x, incx,
beta, y, incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: m, n, kl, ku, lda, incx, incy
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x, y
  real(4), device :: alpha, beta ! device or host variable
```

2.2.15. sgemv

SGEMV performs one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A**T*x + beta*y, where alpha and beta are scalars, x and y are vectors and A is an m by n matrix.

```
subroutine sgemv(t, m, n, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: t
  integer :: m, n, lda, incx, incy
  real(4), device, dimension(lda, *) :: a ! device or host variable
  real(4), device, dimension(*) :: x, y ! device or host variable
  real(4), device :: alpha, beta ! device or host variable
```

```
subroutine cublasSgemv(t, m, n, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: t
  integer :: m, n, lda, incx, incy
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x, y
  real(4), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasSgemv_v2(h, t, m, n, alpha, a, lda, x, incx, beta, y,
incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: m, n, lda, incx, incy
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x, y
  real(4), device :: alpha, beta ! device or host variable
```

2.2.16. sger

SGER performs the rank 1 operation A := alpha*x*y**T + A, where alpha is a scalar, x is an m element vector, y is an n element vector and A is an m by n matrix.

```
subroutine sger(m, n, alpha, x, incx, y, incy, a, lda)
  integer :: m, n, lda, incx, incy
  real(4), device, dimension(lda, *) :: a ! device or host variable
  real(4), device, dimension(*) :: x, y ! device or host variable
  real(4), device :: alpha ! device or host variable
```

```
subroutine cublasSger(m, n, alpha, x, incx, y, incy, a, lda)
  integer :: m, n, lda, incx, incy
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x, y
  real(4), device :: alpha ! device or host variable
```

```
integer(4) function cublasSger_v2(h, m, n, alpha, x, incx, y, incy, a, lda)
  type(cublasHandle) :: h
  integer :: m, n, lda, incx, incy
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x, y
  real(4), device :: alpha ! device or host variable
```

2.2.17. ssbmv

SSBMV performs the matrix-vector operation y := alpha*A*x + beta*y, where alpha and beta are scalars, x and y are n element vectors and A is an n by n symmetric band matrix, with k super-diagonals.

```
subroutine ssbmv(t, n, k, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: t
  integer :: k, n, lda, incx, incy
  real(4), device, dimension(lda, *) :: a ! device or host variable
  real(4), device, dimension(*) :: x, y ! device or host variable
  real(4), device :: alpha, beta ! device or host variable
```

```
subroutine cublasSsbmv(t, n, k, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: t
  integer :: k, n, lda, incx, incy
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x, y
  real(4), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasSsbmv_v2(h, t, n, k, alpha, a, lda, x, incx, beta, y,
incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: k, n, lda, incx, incy
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x, y
  real(4), device :: alpha, beta ! device or host variable
```

2.2.18. sspmv

SSPMV performs the matrix-vector operation y := alpha*A*x + beta*y, where alpha and beta are scalars, x and y are n element vectors and A is an n by n symmetric matrix, supplied in packed form.

```
subroutine sspmv(t, n, alpha, a, x, incx, beta, y, incy)
  character*1 :: t
  integer :: n, incx, incy
  real(4), device, dimension(*) :: a, x, y ! device or host variable
  real(4), device :: alpha, beta ! device or host variable
```

```
subroutine cublasSspmv(t, n, alpha, a, x, incx, beta, y, incy)
  character*1 :: t
  integer :: n, incx, incy
  real(4), device, dimension(*) :: a, x, y
  real(4), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasSspmv_v2(h, t, n, alpha, a, x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy
  real(4), device, dimension(*) :: a, x, y
  real(4), device :: alpha, beta ! device or host variable
```

2.2.19. sspr

SSPR performs the symmetric rank 1 operation A := alpha*x*x**T + A, where alpha is a real scalar, x is an n element vector and A is an n by n symmetric matrix, supplied in packed form.

```
subroutine sspr(t, n, alpha, x, incx, a)
  character*1 :: t
  integer :: n, incx
```

```
  real(4), device, dimension(*) :: a, x ! device or host variable
  real(4), device :: alpha ! device or host variable
```

```
subroutine cublasSspr(t, n, alpha, x, incx, a)
  character*1 :: t
  integer :: n, incx
  real(4), device, dimension(*) :: a, x
  real(4), device :: alpha ! device or host variable
```

```
integer(4) function cublasSspr_v2(h, t, n, alpha, x, incx, a)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx
  real(4), device, dimension(*) :: a, x
  real(4), device :: alpha ! device or host variable
```

2.2.20. sspr2

SSPR2 performs the symmetric rank 2 operation A := alpha*x*y**T + alpha*y*x**T + A, where alpha is a scalar, x and y are n element vectors and A is an n by n symmetric matrix, supplied in packed form.

```
subroutine sspr2(t, n, alpha, x, incx, y, incy, a)
  character*1 :: t
  integer :: n, incx, incy
  real(4), device, dimension(*) :: a, x, y ! device or host variable
  real(4), device :: alpha ! device or host variable
```

```
subroutine cublasSspr2(t, n, alpha, x, incx, y, incy, a)
  character*1 :: t
  integer :: n, incx, incy
  real(4), device, dimension(*) :: a, x, y
  real(4), device :: alpha ! device or host variable
```

```
integer(4) function cublasSspr2_v2(h, t, n, alpha, x, incx, y, incy, a)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy
  real(4), device, dimension(*) :: a, x, y
  real(4), device :: alpha ! device or host variable
```

2.2.21. ssymv

SSYMV performs the matrix-vector operation y := alpha*A*x + beta*y, where alpha and beta are scalars, x and y are n element vectors and A is an n by n symmetric matrix.

```
subroutine ssymv(uplo, n, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: uplo
  integer :: n, lda, incx, incy
  real(4), device, dimension(lda, *) :: a ! device or host variable
  real(4), device, dimension(*) :: x, y ! device or host variable
  real(4), device :: alpha, beta ! device or host variable
```

```
subroutine cublasSsymv(uplo, n, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: uplo
  integer :: n, lda, incx, incy
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x, y
  real(4), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasSsymv_v2(h, uplo, n, alpha, a, lda, x, incx, beta, y,
incy)
  type(cublasHandle) :: h
  integer :: uplo
  integer :: n, lda, incx, incy
  real(4), device, dimension(lda, *) :: a
```

```
  real(4), device, dimension(*) :: x, y
  real(4), device :: alpha, beta ! device or host variable
```

2.2.22. ssyr

SSYR performs the symmetric rank 1 operation A := alpha*x*x**T + A, where alpha is a real scalar, x is an n element vector and A is an n by n symmetric matrix.

```
subroutine ssyr(t, n, alpha, x, incx, a, lda)
  character*1 :: t
  integer :: n, incx, lda
  real(4), device, dimension(lda, *) :: a ! device or host variable
  real(4), device, dimension(*) :: x ! device or host variable
  real(4), device :: alpha ! device or host variable
```

```
subroutine cublasSsyr(t, n, alpha, x, incx, a, lda)
  character*1 :: t
  integer :: n, incx, lda
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x
  real(4), device :: alpha ! device or host variable
```

```
integer(4) function cublasSsyr_v2(h, t, n, alpha, x, incx, a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, lda
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x
  real(4), device :: alpha ! device or host variable
```

2.2.23. ssyr2

SSYR2 performs the symmetric rank 2 operation A := alpha*x*y**T + alpha*y*x**T + A, where alpha is a scalar, x and y are n element vectors and A is an n by n symmetric matrix.

```
subroutine ssyr2(t, n, alpha, x, incx, y, incy, a, lda)
  character*1 :: t
  integer :: n, incx, incy, lda
  real(4), device, dimension(lda, *) :: a ! device or host variable
  real(4), device, dimension(*) :: x, y ! device or host variable
  real(4), device :: alpha ! device or host variable
```

```
subroutine cublasSsyr2(t, n, alpha, x, incx, y, incy, a, lda)
  character*1 :: t
  integer :: n, incx, incy, lda
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x, y
  real(4), device :: alpha ! device or host variable
```

```
integer(4) function cublasSsyr2_v2(h, t, n, alpha, x, incx, y, incy, a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy, lda
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x, y
  real(4), device :: alpha ! device or host variable
```

2.2.24. stbmv

STBMV performs one of the matrix-vector operations x := A*x, or x := A**T*x, where x is an n element vector and A is an n by n unit, or non-unit, upper or lower triangular band matrix, with (k + 1) diagonals.

```
subroutine stbmv(u, t, d, n, k, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, k, incx, lda
  real(4), device, dimension(lda, *) :: a ! device or host variable
  real(4), device, dimension(*) :: x ! device or host variable
```

```
subroutine cublasStbmv(u, t, d, n, k, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, k, incx, lda
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x
```

```
integer(4) function cublasStbmv_v2(h, u, t, d, n, k, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, k, incx, lda
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x
```

2.2.25. stbsv

STBSV solves one of the systems of equations A*x = b, or A**T*x = b, where b and x are n element vectors and A is an n by 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 routine. Such tests must be performed before calling this routine.

```
subroutine stbsv(u, t, d, n, k, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, k, incx, lda
  real(4), device, dimension(lda, *) :: a ! device or host variable
  real(4), device, dimension(*) :: x ! device or host variable
```

```
subroutine cublasStbsv(u, t, d, n, k, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, k, incx, lda
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x
```

```
integer(4) function cublasStbsv_v2(h, u, t, d, n, k, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, k, incx, lda
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x
```

2.2.26. stpmv

STPMV performs one of the matrix-vector operations x := A*x, or x := A**T*x, where x is an n element vector and A is an n by n unit, or non-unit, upper or lower triangular matrix, supplied in packed form.

```
subroutine stpmv(u, t, d, n, a, x, incx)
  character*1 :: u, t, d
  integer :: n, incx
```

```
  real(4), device, dimension(*) :: a, x ! device or host variable
```

```
subroutine cublasStpmv(u, t, d, n, a, x, incx)
  character*1 :: u, t, d
  integer :: n, incx
  real(4), device, dimension(*) :: a, x
```

```
integer(4) function cublasStpmv_v2(h, u, t, d, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx
  real(4), device, dimension(*) :: a, x
```

2.2.27. stpsv

STPSV solves one of the systems of equations A*x = b, or A**T*x = b, where b and x are n element vectors and A is an n by n unit, or non-unit, upper or lower triangular matrix, supplied in packed form. No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine.

```
subroutine stpsv(u, t, d, n, a, x, incx)
  character*1 :: u, t, d
  integer :: n, incx
  real(4), device, dimension(*) :: a, x ! device or host variable
```

```
subroutine cublasStpsv(u, t, d, n, a, x, incx)
  character*1 :: u, t, d
  integer :: n, incx
  real(4), device, dimension(*) :: a, x
```

```
integer(4) function cublasStpsv_v2(h, u, t, d, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx
  real(4), device, dimension(*) :: a, x
```

2.2.28. strmv

STRMV performs one of the matrix-vector operations x := A*x, or x := A**T*x, where x is an n element vector and A is an n by n unit, or non-unit, upper or lower triangular matrix.

```
subroutine strmv(u, t, d, n, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, incx, lda
  real(4), device, dimension(lda, *) :: a ! device or host variable
  real(4), device, dimension(*) :: x ! device or host variable
```

```
subroutine cublasStrmv(u, t, d, n, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, incx, lda
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x
```

```
integer(4) function cublasStrmv_v2(h, u, t, d, n, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx, lda
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x
```

2.2.29. strsv

STRSV solves one of the systems of equations A*x = b, or A**T*x = b, where b and x are n element vectors and A is an n by n unit, or non-unit, upper or lower triangular matrix. No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine.

```
subroutine strsv(u, t, d, n, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, incx, lda
  real(4), device, dimension(lda, *) :: a ! device or host variable
  real(4), device, dimension(*) :: x ! device or host variable
```

```
subroutine cublasStrsv(u, t, d, n, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, incx, lda
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x
```

```
integer(4) function cublasStrsv_v2(h, u, t, d, n, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx, lda
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x
```

2.2.30. sgemm

SGEMM performs one of the matrix-matrix operations C := alpha*op(A)*op(B) + beta*C, where op(X) is one of op(X) = X or op(X) = X**T, alpha and beta are scalars, and A, B and C are matrices, with op(A) an m by k matrix, op(B) a k by n matrix and C an m by n matrix.

```
subroutine sgemm(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: transa, transb
  integer :: m, n, k, lda, ldb, ldc
  real(4), device, dimension(lda, *) :: a ! device or host variable
  real(4), device, dimension(ldb, *) :: b ! device or host variable
  real(4), device, dimension(ldc, *) :: c ! device or host variable
  real(4), device :: alpha, beta ! device or host variable
```

```
subroutine cublasSgemm(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c,
ldc)
  character*1 :: transa, transb
  integer :: m, n, k, lda, ldb, ldc
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(ldb, *) :: b
  real(4), device, dimension(ldc, *) :: c
  real(4), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasSgemm_v2(h, transa, transb, m, n, k, alpha, a, lda,
b, ldb, beta, c, ldc)
  type(cublasHandle) :: h
  integer :: transa, transb
  integer :: m, n, k, lda, ldb, ldc
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(ldb, *) :: b
  real(4), device, dimension(ldc, *) :: c
  real(4), device :: alpha, beta ! device or host variable
```

2.2.31. ssymm

SSYMM 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 scalars, A is a symmetric matrix and B and C are m by n matrices.

```
subroutine ssymm(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: side, uplo
  integer :: m, n, lda, ldb, ldc
  real(4), device, dimension(lda, *) :: a ! device or host variable
  real(4), device, dimension(ldb, *) :: b ! device or host variable
  real(4), device, dimension(ldc, *) :: c ! device or host variable
  real(4), device :: alpha, beta ! device or host variable
```

```
subroutine cublasSsymm(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: side, uplo
  integer :: m, n, lda, ldb, ldc
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(ldb, *) :: b
  real(4), device, dimension(ldc, *) :: c
  real(4), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasSsymm_v2(h, side, uplo, m, n, alpha, a, lda, b, ldb,
beta, c, ldc)
  type(cublasHandle) :: h
  integer :: side, uplo
  integer :: m, n, lda, ldb, ldc
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(ldb, *) :: b
  real(4), device, dimension(ldc, *) :: c
  real(4), device :: alpha, beta ! device or host variable
```

2.2.32. ssyrk

SSYRK performs one of the symmetric rank k operations C := alpha*A*A**T + beta*C, or C := alpha*A**T*A + beta*C, where alpha and beta are scalars, C is an n by n symmetric matrix and A is an n by k matrix in the first case and a k by n matrix in the second case.

```
subroutine ssyrk(uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldc
  real(4), device, dimension(lda, *) :: a ! device or host variable
  real(4), device, dimension(ldc, *) :: c ! device or host variable
  real(4), device :: alpha, beta ! device or host variable
```

```
subroutine cublasSsyrk(uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldc
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(ldc, *) :: c
  real(4), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasSsyrk_v2(h, uplo, trans, n, k, alpha, a, lda, beta,
c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldc
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(ldc, *) :: c
  real(4), device :: alpha, beta ! device or host variable
```

2.2.33. ssyr2k

SSYR2K 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 scalars, C is an n by n symmetric matrix and A and B are n by k matrices in the first case and k by n matrices in the second case.

```
subroutine ssyr2k(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  real(4), device, dimension(lda, *) :: a ! device or host variable
  real(4), device, dimension(ldb, *) :: b ! device or host variable
  real(4), device, dimension(ldc, *) :: c ! device or host variable
  real(4), device :: alpha, beta ! device or host variable
```

```
subroutine cublasSsyr2k(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(ldb, *) :: b
  real(4), device, dimension(ldc, *) :: c
  real(4), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasSsyr2k_v2(h, uplo, trans, n, k, alpha, a, lda, b,
ldb, beta, c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(ldb, *) :: b
  real(4), device, dimension(ldc, *) :: c
  real(4), device :: alpha, beta ! device or host variable
```

2.2.34. ssyrkx

SSYRKX performs a variation of the symmetric rank k update C := alpha*A*B**T + beta*C, where alpha and beta are scalars, C is an n by n symmetric matrix stored in lower or upper mode, and A and B are n by k matrices. This routine can be used when B is in such a way that the result is guaranteed to be symmetric. See the CUBLAS documentation for more details.

```
subroutine ssyrkx(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  real(4), device, dimension(lda, *) :: a ! device or host variable
  real(4), device, dimension(ldb, *) :: b ! device or host variable
  real(4), device, dimension(ldc, *) :: c ! device or host variable
  real(4), device :: alpha, beta ! device or host variable
```

```
subroutine cublasSsyrkx(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(ldb, *) :: b
  real(4), device, dimension(ldc, *) :: c
  real(4), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasSsyrkx_v2(h, uplo, trans, n, k, alpha, a, lda, b,
ldb, beta, c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  real(4), device, dimension(lda, *) :: a
```

```
  real(4), device, dimension(ldb, *) :: b
  real(4), device, dimension(ldc, *) :: c
  real(4), device :: alpha, beta ! device or host variable
```

2.2.35. strmm

STRMM performs one of the matrix-matrix operations B := alpha*op(A)*B, or B := alpha*B*op(A), where alpha is a scalar, B is an m by n matrix, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A**T.

```
subroutine strmm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
  character*1 :: side, uplo, transa, diag
  integer :: m, n, lda, ldb
  real(4), device, dimension(lda, *) :: a ! device or host variable
  real(4), device, dimension(ldb, *) :: b ! device or host variable
  real(4), device :: alpha ! device or host variable
```

```
subroutine cublasStrmm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
  character*1 :: side, uplo, transa, diag
  integer :: m, n, lda, ldb
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(ldb, *) :: b
  real(4), device :: alpha ! device or host variable
```

```
integer(4) function cublasStrmm_v2(h, side, uplo, transa, diag, m, n, alpha, a,
lda, b, ldb, c, ldc)
  type(cublasHandle) :: h
  integer :: side, uplo, transa, diag
  integer :: m, n, lda, ldb, ldc
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(ldb, *) :: b
  real(4), device, dimension(ldc, *) :: c
  real(4), device :: alpha ! device or host variable
```

2.2.36. strsm

STRSM solves one of the matrix equations op(A)*X = alpha*B, or X*op(A) = alpha*B, where alpha is a scalar, X and B are m by n matrices, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A**T. The matrix X is overwritten on B.

```
subroutine strsm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
  character*1 :: side, uplo, transa, diag
  integer :: m, n, lda, ldb
  real(4), device, dimension(lda, *) :: a ! device or host variable
  real(4), device, dimension(ldb, *) :: b ! device or host variable
  real(4), device :: alpha ! device or host variable
```

```
subroutine cublasStrsm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
  character*1 :: side, uplo, transa, diag
  integer :: m, n, lda, ldb
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(ldb, *) :: b
  real(4), device :: alpha ! device or host variable
```

```
integer(4) function cublasStrsm_v2(h, side, uplo, transa, diag, m, n, alpha, a,
lda, b, ldb)
  type(cublasHandle) :: h
  integer :: side, uplo, transa, diag
  integer :: m, n, lda, ldb
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(ldb, *) :: b
  real(4), device :: alpha ! device or host variable
```

2.2.37. cublasSgetrfBatched

SGETRF computes an LU factorization of a general M-by-N matrix A using partial pivoting with row interchanges. The factorization has the form A = P * L * U where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if m > n), and U is upper triangular (upper trapezoidal if m < n). This is the right-looking Level 3 BLAS version of the algorithm.

```
integer(4) function cublasSgetrfBatched(h, n, Aarray, lda, ipvt, info,
batchCount)
  type(cublasHandle) :: h
  integer :: n
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  integer, device :: ipvt(*)
  integer, device :: info(*)
  integer :: batchCount
```

2.2.38. cublasSgetriBatched

SGETRI computes the inverse of a matrix using the LU factorization computed by SGETRF. This method inverts U and then computes inv(A) by solving the system inv(A)*L = inv(U) for inv(A).

```
integer(4) function cublasSgetriBatched(h, n, Aarray, lda, ipvt, Carray, ldc,
info, batchCount)
  type(cublasHandle) :: h
  integer :: n
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  integer, device :: ipvt(*)
  type(c_devptr), device :: Carray(*)
  integer :: ldc
  integer, device :: info(*)
  integer :: batchCount
```

2.2.39. cublasSgetrsBatched

SGETRS solves a system of linear equations A * X = B or A**T * X = B with a general N-by-N matrix A using the LU factorization computed by SGETRF.

```
integer(4) function cublasSgetrsBatched(h, trans, n, nrhs, Aarray, lda, ipvt,
Barray, ldb, info, batchCount)
  type(cublasHandle) :: h
  integer :: trans ! integer or character(1) variable
  integer :: n, nrhs
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  integer, device :: ipvt(*)
  type(c_devptr), device :: Barray(*)
  integer :: ldb
  integer :: info(*)
  integer :: batchCount
```

2.2.40. cublasSgemmBatched

SGEMM performs one of the matrix-matrix operations C := alpha*op(A)*op(B) + beta*C, where op(X) is one of op(X) = X or op(X) = X**T, alpha and beta are scalars,

and A, B and C are matrices, with op(A) an m by k matrix, op(B) a k by n matrix and C an m by n matrix.

```
integer(4) function cublasSgemmBatched(h, transa, transb, m, n, k, alpha,
Aarray, lda, Barray, ldb, beta, Carray, ldc, batchCount)
  type(cublasHandle) :: h
  integer :: transa ! integer or character(1) variable
  integer :: transb ! integer or character(1) variable
  integer :: m, n, k
  real(4), device :: alpha ! device or host variable
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: Barray(*)
  integer :: ldb
  real(4), device :: beta ! device or host variable
  type(c_devptr), device :: Carray(*)
  integer :: ldc
  integer :: batchCount
```

```
integer(4) function cublasSgemmBatched_v2(h, transa, transb, m, n, k, alpha,
       Aarray, lda, Barray, ldb, beta, Carray, ldc, batchCount)
  type(cublasHandle) :: h
  integer :: transa
  integer :: transb
  integer :: m, n, k
  real(4), device :: alpha ! device or host variable
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: Barray(*)
  integer :: ldb
  real(4), device :: beta ! device or host variable
  type(c_devptr), device :: Carray(*)
  integer :: ldc
  integer :: batchCount
```

2.2.41. cublasStrsmBatched

STRSM solves one of the matrix equations op(A)*X = alpha*B, or X*op(A) = alpha*B, where alpha is a scalar, X and B are m by n matrices, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A**T. The matrix X is overwritten on B.

```
integer(4) function cublasStrsmBatched( h, side, uplo, trans, diag, m, n,
alpha, A, lda, B, ldb, batchCount)
  type(cublasHandle) :: h
  integer :: side ! integer or character(1) variable
  integer :: uplo ! integer or character(1) variable
  integer :: trans ! integer or character(1) variable
  integer :: diag ! integer or character(1) variable
  integer :: m, n
  real(4), device :: alpha ! device or host variable
  type(c_devptr), device :: A(*)
  integer :: lda
  type(c_devptr), device :: B(*)
  integer :: ldb
  integer :: batchCount
```

```
integer(4) function cublasStrsmBatched_v2( h, side, uplo, trans, diag, m, n,
alpha, A, lda, B, ldb, batchCount)
  type(cublasHandle) :: h
  integer :: side
  integer :: uplo
  integer :: trans
  integer :: diag
  integer :: m, n
  real(4), device :: alpha ! device or host variable
```

```
  type(c_devptr), device :: A(*)
  integer :: lda
  type(c_devptr), device :: B(*)
  integer :: ldb
  integer :: batchCount
```

2.2.42. cublasSmatinvBatched

cublasSmatinvBatched is a short cut of cublasSgetrfBatched plus cublasSgetriBatched. However it only works if n is less than 32. If not, the user has to go through cublasSgetrfBatched and cublasSgetriBatched.

```
integer(4) function cublasSmatinvBatched(h, n, Aarray, lda, Ainv, lda_inv,
info, batchCount)
  type(cublasHandle) :: h
  integer :: n
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: Ainv(*)
  integer :: lda_inv
  integer, device :: info(*)
  integer :: batchCount
```

2.2.43. cublasSgeqrfBatched

SGEQRF computes a QR factorization of a real M-by-N matrix A: A = Q * R.

```
integer(4) function cublasSgeqrfBatched(h, m, n, Aarray, lda, Tau, info,
batchCount)
  type(cublasHandle) :: h
  integer :: m, n
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: Tau(*)
  integer :: info(*)
  integer :: batchCount
```

2.2.44. cublasSgelsBatched

SGELS solves overdetermined or underdetermined real linear systems involving an M-by-N matrix A, or its transpose, using a QR or LQ factorization of A. It is assumed that A has full rank. The following options are provided: 1. If TRANS = 'N' and m >= n: find the least squares solution of an overdetermined system, i.e., solve the least squares problem minimize || B - A*X ||. 2. If TRANS = 'N' and m < n: find the minimum norm solution of an underdetermined system A * X = B. 3. If TRANS = 'T' and m >= n: find the minimum norm solution of an undetermined system A**T * X = B. 4. If TRANS = 'T' and m < n: find the least squares solution of an overdetermined system, i.e., solve the least squares problem minimize || B - A**T * X ||. Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the M-by-NRHS right hand side matrix B and the N-by-NRHS solution matrix X.

```
integer(4) function cublasSgelsBatched(h, trans, m, n, nrhs, Aarray, lda,
Carray, ldc, info, devinfo, batchCount)
  type(cublasHandle) :: h
  integer :: trans ! integer or character(1) variable
  integer :: m, n, nrhs
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: Carray(*)
  integer :: ldc
```

```
  integer :: info(*)
  integer, device :: devinfo(*)
  integer :: batchCount
```

2.2.45. cublasSgemmStridedBatched

SGEMM performs one of the matrix-matrix operations C := alpha*op(A)*op(B) + beta*C, where op(X) is one of op(X) = X or op(X) = X**T, alpha and beta are scalars, and A, B and C are matrices, with op(A) an m by k matrix, op(B) a k by n matrix and C an m by n matrix.

```
integer(4) function cublasSgemmStridedBatched(h, transa, transb, m, n, k,
alpha, Aarray, lda, strideA, Barray, ldb, strideB, beta, Carray, ldc, strideC,
batchCount)
  type(cublasHandle) :: h
  integer :: transa ! integer or character(1) variable
  integer :: transb ! integer or character(1) variable
  integer :: m, n, k
  real(4), device :: alpha ! device or host variable
  real(4), device :: Aarray(*)
  integer :: lda
  integer :: strideA
  real(4), device :: Barray(*)
  integer :: ldb
  integer :: strideB
  real(4), device :: beta ! device or host variable
  real(4), device :: Carray(*)
  integer :: ldc
  integer :: strideC
  integer :: batchCount
```

```
integer(4) function cublasSgemmStridedBatched_v2(h, transa, transb, m, n, k,
alpha,           Aarray, lda, strideA, Barray, ldb, strideB, beta, Carray, ldc,
strideC, batchCount)
  type(cublasHandle) :: h
  integer :: transa
  integer :: transb
  integer :: m, n, k
  real(4), device :: alpha ! device or host variable
  real(4), device :: Aarray(*)
  integer :: lda
  integer :: strideA
  real(4), device :: Barray(*)
  integer :: ldb
  integer :: strideB
  real(4), device :: beta ! device or host variable
  real(4), device :: Carray(*)
  integer :: ldc
  integer :: strideC
  integer :: batchCount
```

2.3. Double Precision Functions and Subroutines

This section contains interfaces to the double precision BLAS and cuBLAS functions and subroutines.

2.3.1. idamax

IDAMAX finds the the index of the element having the maximum absolute value.

```
integer(4) function idamax(n, x, incx)
```

```
  integer :: n
  real(8), device, dimension(*) :: x ! device or host variable
  integer :: incx
```

```
integer(4) function cublasIdamax(n, x, incx)
  integer :: n
  real(8), device, dimension(*) :: x
  integer :: incx
```

```
integer(4) function cublasIdamax_v2(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  real(8), device, dimension(*) :: x
  integer :: incx
  integer, device :: res ! device or host variable
```

2.3.2. idamin

IDAMIN finds the index of the element having the minimum absolute value.

```
integer(4) function idamin(n, x, incx)
  integer :: n
  real(8), device, dimension(*) :: x ! device or host variable
  integer :: incx
```

```
integer(4) function cublasIdamin(n, x, incx)
  integer :: n
  real(8), device, dimension(*) :: x
  integer :: incx
```

```
integer(4) function cublasIdamin_v2(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  real(8), device, dimension(*) :: x
  integer :: incx
  integer, device :: res ! device or host variable
```

2.3.3. dasum

DASUM takes the sum of the absolute values.

```
real(8) function dasum(n, x, incx)
  integer :: n
  real(8), device, dimension(*) :: x ! device or host variable
  integer :: incx
```

```
real(8) function cublasDasum(n, x, incx)
  integer :: n
  real(8), device, dimension(*) :: x
  integer :: incx
```

```
integer(4) function cublasDasum_v2(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  real(8), device, dimension(*) :: x
  integer :: incx
  real(8), device :: res ! device or host variable
```

2.3.4. daxpy

DAXPY constant times a vector plus a vector.

```
subroutine daxpy(n, a, x, incx, y, incy)
  integer :: n
  real(8), device :: a ! device or host variable
  real(8), device, dimension(*) :: x, y ! device or host variable
```

```
  integer :: incx, incy
```

```
subroutine cublasDaxpy(n, a, x, incx, y, incy)
  integer :: n
  real(8), device :: a ! device or host variable
  real(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

```
integer(4) function cublasDaxpy_v2(h, n, a, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  real(8), device :: a ! device or host variable
  real(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

2.3.5. dcopy

DCOPY copies a vector, x, to a vector, y.

```
subroutine dcopy(n, x, incx, y, incy)
  integer :: n
  real(8), device, dimension(*) :: x, y ! device or host variable
  integer :: incx, incy
```

```
subroutine cublasDcopy(n, x, incx, y, incy)
  integer :: n
  real(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

```
integer(4) function cublasDcopy_v2(h, n, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  real(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

2.3.6. ddot

DDOT forms the dot product of two vectors.

```
real(8) function ddot(n, x, incx, y, incy)
  integer :: n
  real(8), device, dimension(*) :: x, y ! device or host variable
  integer :: incx, incy
```

```
real(8) function cublasDdot(n, x, incx, y, incy)
  integer :: n
  real(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

```
integer(4) function cublasDdot_v2(h, n, x, incx, y, incy, res)
  type(cublasHandle) :: h
  integer :: n
  real(8), device, dimension(*) :: x, y
  integer :: incx, incy
  real(8), device :: res ! device or host variable
```

2.3.7. dnrm2

DNRM2 returns the euclidean norm of a vector via the function name, so that DNRM2 := sqrt(x'*x)

```
real(8) function dnrm2(n, x, incx)
  integer :: n
  real(8), device, dimension(*) :: x ! device or host variable
```

```
  integer :: incx
```

```
real(8) function cublasDnrm2(n, x, incx)
  integer :: n
  real(8), device, dimension(*) :: x
  integer :: incx
```

```
integer(4) function cublasDnrm2_v2(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  real(8), device, dimension(*) :: x
  integer :: incx
  real(8), device :: res ! device or host variable
```

2.3.8. drot

DROT applies a plane rotation.

```
subroutine drot(n, x, incx, y, incy, sc, ss)
  integer :: n
  real(8), device :: sc, ss ! device or host variable
  real(8), device, dimension(*) :: x, y ! device or host variable
  integer :: incx, incy
```

```
subroutine cublasDrot(n, x, incx, y, incy, sc, ss)
  integer :: n
  real(8), device :: sc, ss ! device or host variable
  real(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

```
integer(4) function cublasDrot_v2(h, n, x, incx, y, incy, sc, ss)
  type(cublasHandle) :: h
  integer :: n
  real(8), device :: sc, ss ! device or host variable
  real(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

2.3.9. drotg

DROTG constructs a Givens plane rotation.

```
subroutine drotg(sa, sb, sc, ss)
  real(8), device :: sa, sb, sc, ss ! device or host variable
```

```
subroutine cublasDrotg(sa, sb, sc, ss)
  real(8), device :: sa, sb, sc, ss ! device or host variable
```

```
integer(4) function cublasDrotg_v2(h, sa, sb, sc, ss)
  type(cublasHandle) :: h
  real(8), device :: sa, sb, sc, ss ! device or host variable
```

2.3.10. drotm

DROTM applies the modified Givens transformation, H, to the 2 by N matrix (DX**T) , where **T indicates transpose. The elements of DX are in (DX**T) DX(LX+I*INCX), I = 0 to N-1, where LX = 1 if INCX .GE. 0, ELSE LX = (-INCX)*N, and similarly for DY using LY and INCY. With DPARAM(1)=DFLAG, H has one of the following forms.. DFLAG=-1.D0 DFLAG=0.D0 DFLAG=1.D0 DFLAG=-2.D0 (DH11 DH12) (1.D0 DH12) (DH11 1.D0) (1.D0 0.D0) H=( ) ( ) ( ) ( ) (DH21 DH22), (DH21 1.D0), (-1.D0 DH22), (0.D0 1.D0). See DROTMG for a description of data storage in DPARAM.

```
subroutine drotm(n, x, incx, y, incy, param)
  integer :: n
  real(8), device, dimension(*) :: x, y ! device or host variable
```

```
  integer :: incx, incy
```

```
subroutine cublasDrotm(n, x, incx, y, incy, param)
  integer :: n
  real(8), device, dimension(*) :: x, y
  integer :: incx, incy
  real(8), device :: param(*) ! device or host variable
```

```
integer(4) function cublasDrotm_v2(h, n, x, incx, y, incy, param)
  type(cublasHandle) :: h
  integer :: n
  real(8), device :: param(*) ! device or host variable
  real(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

2.3.11. drotmg

DROTMG constructs the modified Givens transformation matrix H which zeros the second component of the 2-vector (SQRT(DD1)*DX1,SQRT(DD2)*DY2)**T. With DPARAM(1)=DFLAG, H has one of the following forms.. DFLAG=-1.D0 DFLAG=0.D0 DFLAG=1.D0 DFLAG=-2.D0 (DH11 DH12) (1.D0 DH12) (DH11 1.D0) (1.D0 0.D0) H=( ) ( ) ( ) ( ) (DH21 DH22), (DH21 1.D0), (-1.D0 DH22), (0.D0 1.D0). Locations 2-4 of DPARAM contain DH11, DH21, DH12, and DH22 respectively. (Values of 1.D0, -1.D0, of 0.D0 implied by the value of DPARAM(1) are not stored in DPARAM.)

```
subroutine drotmg(d1, d2, x1, y1, param)
  real(8), device :: d1, d2, x1, y1, param(*) ! device or host variable
```

```
subroutine cublasDrotmg(d1, d2, x1, y1, param)
  real(8), device :: d1, d2, x1, y1, param(*) ! device or host variable
```

```
integer(4) function cublasDrotmg_v2(h, d1, d2, x1, y1, param)
  type(cublasHandle) :: h
  real(8), device :: d1, d2, x1, y1, param(*) ! device or host variable
```

2.3.12. dscal

DSCAL scales a vector by a constant.

```
subroutine dscal(n, a, x, incx)
  integer :: n
  real(8), device :: a ! device or host variable
  real(8), device, dimension(*) :: x ! device or host variable
  integer :: incx
```

```
subroutine cublasDscal(n, a, x, incx)
  integer :: n
  real(8), device :: a ! device or host variable
  real(8), device, dimension(*) :: x
  integer :: incx
```

```
integer(4) function cublasDscal_v2(h, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: n
  real(8), device :: a ! device or host variable
  real(8), device, dimension(*) :: x
  integer :: incx
```

2.3.13. dswap

interchanges two vectors.

```
subroutine dswap(n, x, incx, y, incy)
  integer :: n
```

```
  real(8), device, dimension(*) :: x, y ! device or host variable
  integer :: incx, incy
```

```
subroutine cublasDswap(n, x, incx, y, incy)
  integer :: n
  real(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

```
integer(4) function cublasDswap_v2(h, n, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  real(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

2.3.14. dgbmv

DGBMV performs one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A**T*x + beta*y, where alpha and beta are scalars, x and y are vectors and A is an m by n band matrix, with kl sub-diagonals and ku super-diagonals.

```
subroutine dgbmv(t, m, n, kl, ku, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: t
  integer :: m, n, kl, ku, lda, incx, incy
  real(8), device, dimension(lda, *) :: a ! device or host variable
  real(8), device, dimension(*) :: x, y ! device or host variable
  real(8), device :: alpha, beta ! device or host variable
```

```
subroutine cublasDgbmv(t, m, n, kl, ku, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: t
  integer :: m, n, kl, ku, lda, incx, incy
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x, y
  real(8), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasDgbmv_v2(h, t, m, n, kl, ku, alpha, a, lda, x, incx,
beta, y, incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: m, n, kl, ku, lda, incx, incy
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x, y
  real(8), device :: alpha, beta ! device or host variable
```

2.3.15. dgemv

DGEMV performs one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A**T*x + beta*y, where alpha and beta are scalars, x and y are vectors and A is an m by n matrix.

```
subroutine dgemv(t, m, n, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: t
  integer :: m, n, lda, incx, incy
  real(8), device, dimension(lda, *) :: a ! device or host variable
  real(8), device, dimension(*) :: x, y ! device or host variable
  real(8), device :: alpha, beta ! device or host variable
```

```
subroutine cublasDgemv(t, m, n, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: t
  integer :: m, n, lda, incx, incy
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x, y
  real(8), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasDgemv_v2(h, t, m, n, alpha, a, lda, x, incx, beta, y,
incy)
  type(cublasHandle) :: h
```

```
  integer :: t
  integer :: m, n, lda, incx, incy
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x, y
  real(8), device :: alpha, beta ! device or host variable
```

2.3.16. dger

DGER performs the rank 1 operation A := alpha*x*y**T + A, where alpha is a scalar, x is an m element vector, y is an n element vector and A is an m by n matrix.

```
subroutine dger(m, n, alpha, x, incx, y, incy, a, lda)
  integer :: m, n, lda, incx, incy
  real(8), device, dimension(lda, *) :: a ! device or host variable
  real(8), device, dimension(*) :: x, y ! device or host variable
  real(8), device :: alpha ! device or host variable
```

```
subroutine cublasDger(m, n, alpha, x, incx, y, incy, a, lda)
  integer :: m, n, lda, incx, incy
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x, y
  real(8), device :: alpha ! device or host variable
```

```
integer(4) function cublasDger_v2(h, m, n, alpha, x, incx, y, incy, a, lda)
  type(cublasHandle) :: h
  integer :: m, n, lda, incx, incy
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x, y
  real(8), device :: alpha ! device or host variable
```

2.3.17. dsbmv

DSBMV performs the matrix-vector operation y := alpha*A*x + beta*y, where alpha and beta are scalars, x and y are n element vectors and A is an n by n symmetric band matrix, with k super-diagonals.

```
subroutine dsbmv(t, n, k, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: t
  integer :: k, n, lda, incx, incy
  real(8), device, dimension(lda, *) :: a ! device or host variable
  real(8), device, dimension(*) :: x, y ! device or host variable
  real(8), device :: alpha, beta ! device or host variable
```

```
subroutine cublasDsbmv(t, n, k, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: t
  integer :: k, n, lda, incx, incy
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x, y
  real(8), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasDsbmv_v2(h, t, n, k, alpha, a, lda, x, incx, beta, y,
incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: k, n, lda, incx, incy
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x, y
  real(8), device :: alpha, beta ! device or host variable
```

2.3.18. dspmv

DSPMV performs the matrix-vector operation y := alpha*A*x + beta*y, where alpha and beta are scalars, x and y are n element vectors and A is an n by n symmetric matrix, supplied in packed form.

```
subroutine dspmv(t, n, alpha, a, x, incx, beta, y, incy)
  character*1 :: t
  integer :: n, incx, incy
  real(8), device, dimension(*) :: a, x, y ! device or host variable
  real(8), device :: alpha, beta ! device or host variable
```

```
subroutine cublasDspmv(t, n, alpha, a, x, incx, beta, y, incy)
  character*1 :: t
  integer :: n, incx, incy
  real(8), device, dimension(*) :: a, x, y
  real(8), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasDspmv_v2(h, t, n, alpha, a, x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy
  real(8), device, dimension(*) :: a, x, y
  real(8), device :: alpha, beta ! device or host variable
```

2.3.19. dspr

DSPR performs the symmetric rank 1 operation A := alpha*x*x**T + A, where alpha is a real scalar, x is an n element vector and A is an n by n symmetric matrix, supplied in packed form.

```
subroutine dspr(t, n, alpha, x, incx, a)
  character*1 :: t
  integer :: n, incx
  real(8), device, dimension(*) :: a, x ! device or host variable
  real(8), device :: alpha ! device or host variable
```

```
subroutine cublasDspr(t, n, alpha, x, incx, a)
  character*1 :: t
  integer :: n, incx
  real(8), device, dimension(*) :: a, x
  real(8), device :: alpha ! device or host variable
```

```
integer(4) function cublasDspr_v2(h, t, n, alpha, x, incx, a)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx
  real(8), device, dimension(*) :: a, x
  real(8), device :: alpha ! device or host variable
```

2.3.20. dspr2

DSPR2 performs the symmetric rank 2 operation A := alpha*x*y**T + alpha*y*x**T + A, where alpha is a scalar, x and y are n element vectors and A is an n by n symmetric matrix, supplied in packed form.

```
subroutine dspr2(t, n, alpha, x, incx, y, incy, a)
  character*1 :: t
  integer :: n, incx, incy
  real(8), device, dimension(*) :: a, x, y ! device or host variable
  real(8), device :: alpha ! device or host variable
```

```
subroutine cublasDspr2(t, n, alpha, x, incx, y, incy, a)
```

```
  character*1 :: t
  integer :: n, incx, incy
  real(8), device, dimension(*) :: a, x, y
  real(8), device :: alpha ! device or host variable
```

```
integer(4) function cublasDspr2_v2(h, t, n, alpha, x, incx, y, incy, a)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy
  real(8), device, dimension(*) :: a, x, y
  real(8), device :: alpha ! device or host variable
```

2.3.21. dsymv

DSYMV performs the matrix-vector operation y := alpha*A*x + beta*y, where alpha and beta are scalars, x and y are n element vectors and A is an n by n symmetric matrix.

```
subroutine dsymv(uplo, n, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: uplo
  integer :: n, lda, incx, incy
  real(8), device, dimension(lda, *) :: a ! device or host variable
  real(8), device, dimension(*) :: x, y ! device or host variable
  real(8), device :: alpha, beta ! device or host variable
```

```
subroutine cublasDsymv(uplo, n, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: uplo
  integer :: n, lda, incx, incy
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x, y
  real(8), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasDsymv_v2(h, uplo, n, alpha, a, lda, x, incx, beta, y,
incy)
  type(cublasHandle) :: h
  integer :: uplo
  integer :: n, lda, incx, incy
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x, y
  real(8), device :: alpha, beta ! device or host variable
```

2.3.22. dsyr

DSYR performs the symmetric rank 1 operation A := alpha*x*x**T + A, where alpha is a real scalar, x is an n element vector and A is an n by n symmetric matrix.

```
subroutine dsyr(t, n, alpha, x, incx, a, lda)
  character*1 :: t
  integer :: n, incx, lda
  real(8), device, dimension(lda, *) :: a ! device or host variable
  real(8), device, dimension(*) :: x ! device or host variable
  real(8), device :: alpha ! device or host variable
```

```
subroutine cublasDsyr(t, n, alpha, x, incx, a, lda)
  character*1 :: t
  integer :: n, incx, lda
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x
  real(8), device :: alpha ! device or host variable
```

```
integer(4) function cublasDsyr_v2(h, t, n, alpha, x, incx, a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, lda
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x
  real(8), device :: alpha ! device or host variable
```

2.3.23. dsyr2

DSYR2 performs the symmetric rank 2 operation A := alpha*x*y**T + alpha*y*x**T + A, where alpha is a scalar, x and y are n element vectors and A is an n by n symmetric matrix.

```
subroutine dsyr2(t, n, alpha, x, incx, y, incy, a, lda)
  character*1 :: t
  integer :: n, incx, incy, lda
  real(8), device, dimension(lda, *) :: a ! device or host variable
  real(8), device, dimension(*) :: x, y ! device or host variable
  real(8), device :: alpha ! device or host variable
```

```
subroutine cublasDsyr2(t, n, alpha, x, incx, y, incy, a, lda)
  character*1 :: t
  integer :: n, incx, incy, lda
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x, y
  real(8), device :: alpha ! device or host variable
```

```
integer(4) function cublasDsyr2_v2(h, t, n, alpha, x, incx, y, incy, a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy, lda
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x, y
  real(8), device :: alpha ! device or host variable
```

2.3.24. dtbmv

DTBMV performs one of the matrix-vector operations x := A*x, or x := A**T*x, where x is an n element vector and A is an n by n unit, or non-unit, upper or lower triangular band matrix, with (k + 1) diagonals.

```
subroutine dtbmv(u, t, d, n, k, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, k, incx, lda
  real(8), device, dimension(lda, *) :: a ! device or host variable
  real(8), device, dimension(*) :: x ! device or host variable
```

```
subroutine cublasDtbmv(u, t, d, n, k, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, k, incx, lda
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x
```

```
integer(4) function cublasDtbmv_v2(h, u, t, d, n, k, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, k, incx, lda
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x
```

2.3.25. dtbsv

DTBSV solves one of the systems of equations A*x = b, or A**T*x = b, where b and x are n element vectors and A is an n by 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 routine. Such tests must be performed before calling this routine.

```
subroutine dtbsv(u, t, d, n, k, a, lda, x, incx)
  character*1 :: u, t, d
```

```
  integer :: n, k, incx, lda
  real(8), device, dimension(lda, *) :: a ! device or host variable
  real(8), device, dimension(*) :: x ! device or host variable
```

```
subroutine cublasDtbsv(u, t, d, n, k, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, k, incx, lda
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x
```

```
integer(4) function cublasDtbsv_v2(h, u, t, d, n, k, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, k, incx, lda
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x
```

2.3.26. dtpmv

DTPMV performs one of the matrix-vector operations x := A*x, or x := A**T*x, where x is an n element vector and A is an n by n unit, or non-unit, upper or lower triangular matrix, supplied in packed form.

```
subroutine dtpmv(u, t, d, n, a, x, incx)
  character*1 :: u, t, d
  integer :: n, incx
  real(8), device, dimension(*) :: a, x ! device or host variable
```

```
subroutine cublasDtpmv(u, t, d, n, a, x, incx)
  character*1 :: u, t, d
  integer :: n, incx
  real(8), device, dimension(*) :: a, x
```

```
integer(4) function cublasDtpmv_v2(h, u, t, d, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx
  real(8), device, dimension(*) :: a, x
```

2.3.27. dtpsv

DTPSV solves one of the systems of equations A*x = b, or A**T*x = b, where b and x are n element vectors and A is an n by n unit, or non-unit, upper or lower triangular matrix, supplied in packed form. No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine.

```
subroutine dtpsv(u, t, d, n, a, x, incx)
  character*1 :: u, t, d
  integer :: n, incx
  real(8), device, dimension(*) :: a, x ! device or host variable
```

```
subroutine cublasDtpsv(u, t, d, n, a, x, incx)
  character*1 :: u, t, d
  integer :: n, incx
  real(8), device, dimension(*) :: a, x
```

```
integer(4) function cublasDtpsv_v2(h, u, t, d, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx
  real(8), device, dimension(*) :: a, x
```

2.3.28. dtrmv

DTRMV performs one of the matrix-vector operations x := A*x, or x := A**T*x, where x is an n element vector and A is an n by n unit, or non-unit, upper or lower triangular matrix.

```
subroutine dtrmv(u, t, d, n, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, incx, lda
  real(8), device, dimension(lda, *) :: a ! device or host variable
  real(8), device, dimension(*) :: x ! device or host variable
```

```
subroutine cublasDtrmv(u, t, d, n, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, incx, lda
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x
```

```
integer(4) function cublasDtrmv_v2(h, u, t, d, n, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx, lda
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x
```

2.3.29. dtrsv

DTRSV solves one of the systems of equations A*x = b, or A**T*x = b, where b and x are n element vectors and A is an n by n unit, or non-unit, upper or lower triangular matrix. No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine.

```
subroutine dtrsv(u, t, d, n, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, incx, lda
  real(8), device, dimension(lda, *) :: a ! device or host variable
  real(8), device, dimension(*) :: x ! device or host variable
```

```
subroutine cublasDtrsv(u, t, d, n, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, incx, lda
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x
```

```
integer(4) function cublasDtrsv_v2(h, u, t, d, n, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx, lda
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x
```

2.3.30. dgemm

DGEMM performs one of the matrix-matrix operations C := alpha*op(A)*op(B) + beta*C, where op(X) is one of op(X) = X or op(X) = X**T, alpha and beta are scalars, and A, B and C are matrices, with op(A) an m by k matrix, op(B) a k by n matrix and C an m by n matrix.

```
subroutine dgemm(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: transa, transb
  integer :: m, n, k, lda, ldb, ldc
  real(8), device, dimension(lda, *) :: a ! device or host variable
```

```
  real(8), device, dimension(ldb, *) :: b ! device or host variable
  real(8), device, dimension(ldc, *) :: c ! device or host variable
  real(8), device :: alpha, beta ! device or host variable
```

```
subroutine cublasDgemm(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c,
ldc)
  character*1 :: transa, transb
  integer :: m, n, k, lda, ldb, ldc
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(ldb, *) :: b
  real(8), device, dimension(ldc, *) :: c
  real(8), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasDgemm_v2(h, transa, transb, m, n, k, alpha, a, lda,
b, ldb, beta, c, ldc)
  type(cublasHandle) :: h
  integer :: transa, transb
  integer :: m, n, k, lda, ldb, ldc
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(ldb, *) :: b
  real(8), device, dimension(ldc, *) :: c
  real(8), device :: alpha, beta ! device or host variable
```

2.3.31. dsymm

DSYMM 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 scalars, A is a symmetric matrix and B and C are m by n matrices.

```
subroutine dsymm(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: side, uplo
  integer :: m, n, lda, ldb, ldc
  real(8), device, dimension(lda, *) :: a ! device or host variable
  real(8), device, dimension(ldb, *) :: b ! device or host variable
  real(8), device, dimension(ldc, *) :: c ! device or host variable
  real(8), device :: alpha, beta ! device or host variable
```

```
subroutine cublasDsymm(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: side, uplo
  integer :: m, n, lda, ldb, ldc
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(ldb, *) :: b
  real(8), device, dimension(ldc, *) :: c
  real(8), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasDsymm_v2(h, side, uplo, m, n, alpha, a, lda, b, ldb,
beta, c, ldc)
  type(cublasHandle) :: h
  integer :: side, uplo
  integer :: m, n, lda, ldb, ldc
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(ldb, *) :: b
  real(8), device, dimension(ldc, *) :: c
  real(8), device :: alpha, beta ! device or host variable
```

2.3.32. dsyrk

DSYRK performs one of the symmetric rank k operations C := alpha*A*A**T + beta*C, or C := alpha*A**T*A + beta*C, where alpha and beta are scalars, C is an n by n symmetric matrix and A is an n by k matrix in the first case and a k by n matrix in the second case.

```
subroutine dsyrk(uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldc
  real(8), device, dimension(lda, *) :: a ! device or host variable
```

```
  real(8), device, dimension(ldc, *) :: c ! device or host variable
  real(8), device :: alpha, beta ! device or host variable
```

```
subroutine cublasDsyrk(uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldc
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(ldc, *) :: c
  real(8), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasDsyrk_v2(h, uplo, trans, n, k, alpha, a, lda, beta,
c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldc
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(ldc, *) :: c
  real(8), device :: alpha, beta ! device or host variable
```

2.3.33. dsyr2k

DSYR2K 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 scalars, C is an n by n symmetric matrix and A and B are n by k matrices in the first case and k by n matrices in the second case.

```
subroutine dsyr2k(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  real(8), device, dimension(lda, *) :: a ! device or host variable
  real(8), device, dimension(ldb, *) :: b ! device or host variable
  real(8), device, dimension(ldc, *) :: c ! device or host variable
  real(8), device :: alpha, beta ! device or host variable
```

```
subroutine cublasDsyr2k(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(ldb, *) :: b
  real(8), device, dimension(ldc, *) :: c
  real(8), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasDsyr2k_v2(h, uplo, trans, n, k, alpha, a, lda, b,
ldb, beta, c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(ldb, *) :: b
  real(8), device, dimension(ldc, *) :: c
  real(8), device :: alpha, beta ! device or host variable
```

2.3.34. dsyrkx

DSYRKX performs a variation of the symmetric rank k update C := alpha*A*B**T + beta*C, where alpha and beta are scalars, C is an n by n symmetric matrix stored in lower or upper mode, and A and B are n by k matrices. This routine can be used when B is in such a way that the result is guaranteed to be symmetric. See the CUBLAS documentation for more details.

```
subroutine dsyrkx(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  real(8), device, dimension(lda, *) :: a ! device or host variable
```

```
  real(8), device, dimension(ldb, *) :: b ! device or host variable
  real(8), device, dimension(ldc, *) :: c ! device or host variable
  real(8), device :: alpha, beta ! device or host variable
```

```
subroutine cublasDsyrkx(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(ldb, *) :: b
  real(8), device, dimension(ldc, *) :: c
  real(8), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasDsyrkx_v2(h, uplo, trans, n, k, alpha, a, lda, b,
ldb, beta, c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(ldb, *) :: b
  real(8), device, dimension(ldc, *) :: c
  real(8), device :: alpha, beta ! device or host variable
```

2.3.35. dtrmm

DTRMM performs one of the matrix-matrix operations B := alpha*op(A)*B, or B := alpha*B*op(A), where alpha is a scalar, B is an m by n matrix, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A**T.

```
subroutine dtrmm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
  character*1 :: side, uplo, transa, diag
  integer :: m, n, lda, ldb
  real(8), device, dimension(lda, *) :: a ! device or host variable
  real(8), device, dimension(ldb, *) :: b ! device or host variable
  real(8), device :: alpha ! device or host variable
```

```
subroutine cublasDtrmm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
  character*1 :: side, uplo, transa, diag
  integer :: m, n, lda, ldb
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(ldb, *) :: b
  real(8), device :: alpha ! device or host variable
```

```
integer(4) function cublasDtrmm_v2(h, side, uplo, transa, diag, m, n, alpha, a,
lda, b, ldb, c, ldc)
  type(cublasHandle) :: h
  integer :: side, uplo, transa, diag
  integer :: m, n, lda, ldb, ldc
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(ldb, *) :: b
  real(8), device, dimension(ldc, *) :: c
  real(8), device :: alpha ! device or host variable
```

2.3.36. dtrsm

DTRSM solves one of the matrix equations op(A)*X = alpha*B, or X*op(A) = alpha*B, where alpha is a scalar, X and B are m by n matrices, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A**T. The matrix X is overwritten on B.

```
subroutine dtrsm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
  character*1 :: side, uplo, transa, diag
  integer :: m, n, lda, ldb
  real(8), device, dimension(lda, *) :: a ! device or host variable
  real(8), device, dimension(ldb, *) :: b ! device or host variable
```

```
  real(8), device :: alpha ! device or host variable
```

```
subroutine cublasDtrsm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
  character*1 :: side, uplo, transa, diag
  integer :: m, n, lda, ldb
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(ldb, *) :: b
  real(8), device :: alpha ! device or host variable
```

```
integer(4) function cublasDtrsm_v2(h, side, uplo, transa, diag, m, n, alpha, a,
lda, b, ldb)
  type(cublasHandle) :: h
  integer :: side, uplo, transa, diag
  integer :: m, n, lda, ldb
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(ldb, *) :: b
  real(8), device :: alpha ! device or host variable
```

2.3.37. cublasDgetrfBatched

DGETRF computes an LU factorization of a general M-by-N matrix A using partial pivoting with row interchanges. The factorization has the form A = P * L * U where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if m > n), and U is upper triangular (upper trapezoidal if m < n). This is the right-looking Level 3 BLAS version of the algorithm.

```
integer(4) function cublasDgetrfBatched(h, n, Aarray, lda, ipvt, info,
batchCount)
  type(cublasHandle) :: h
  integer :: n
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  integer, device :: ipvt(*)
  integer, device :: info(*)
  integer :: batchCount
```

2.3.38. cublasDgetriBatched

DGETRI computes the inverse of a matrix using the LU factorization computed by DGETRF. This method inverts U and then computes inv(A) by solving the system inv(A)*L = inv(U) for inv(A).

```
integer(4) function cublasDgetriBatched(h, n, Aarray, lda, ipvt, Carray, ldc,
info, batchCount)
  type(cublasHandle) :: h
  integer :: n
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  integer, device :: ipvt(*)
  type(c_devptr), device :: Carray(*)
  integer :: ldc
  integer, device :: info(*)
  integer :: batchCount
```

2.3.39. cublasDgetrsBatched

DGETRS solves a system of linear equations A * X = B or A**T * X = B with a general N-by-N matrix A using the LU factorization computed by DGETRF.

```
integer(4) function cublasDgetrsBatched(h, trans, n, nrhs, Aarray, lda, ipvt,
Barray, ldb, info, batchCount)
  type(cublasHandle) :: h
  integer :: trans ! integer or character(1) variable
```

```
  integer :: n, nrhs
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  integer, device :: ipvt(*)
  type(c_devptr), device :: Barray(*)
  integer :: ldb
  integer :: info(*)
  integer :: batchCount
```

2.3.40. cublasDgemmBatched

DGEMM performs one of the matrix-matrix operations C := alpha*op(A)*op(B) + beta*C, where op(X) is one of op(X) = X or op(X) = X**T, alpha and beta are scalars, and A, B and C are matrices, with op(A) an m by k matrix, op(B) a k by n matrix and C an m by n matrix.

```
integer(4) function cublasDgemmBatched(h, transa, transb, m, n, k, alpha,
Aarray, lda, Barray, ldb, beta, Carray, ldc, batchCount)
  type(cublasHandle) :: h
  integer :: transa ! integer or character(1) variable
  integer :: transb ! integer or character(1) variable
  integer :: m, n, k
  real(8), device :: alpha ! device or host variable
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: Barray(*)
  integer :: ldb
  real(8), device :: beta ! device or host variable
  type(c_devptr), device :: Carray(*)
  integer :: ldc
  integer :: batchCount
```

```
integer(4) function cublasDgemmBatched_v2(h, transa, transb, m, n, k, alpha,
       Aarray, lda, Barray, ldb, beta, Carray, ldc, batchCount)
  type(cublasHandle) :: h
  integer :: transa
  integer :: transb
  integer :: m, n, k
  real(8), device :: alpha ! device or host variable
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: Barray(*)
  integer :: ldb
  real(8), device :: beta ! device or host variable
  type(c_devptr), device :: Carray(*)
  integer :: ldc
  integer :: batchCount
```

2.3.41. cublasDtrsmBatched

DTRSM solves one of the matrix equations op(A)*X = alpha*B, or X*op(A) = alpha*B, where alpha is a scalar, X and B are m by n matrices, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A**T. The matrix X is overwritten on B.

```
integer(4) function cublasDtrsmBatched( h, side, uplo, trans, diag, m, n,
alpha, A, lda, B, ldb, batchCount)
  type(cublasHandle) :: h
  integer :: side ! integer or character(1) variable
  integer :: uplo ! integer or character(1) variable
  integer :: trans ! integer or character(1) variable
  integer :: diag ! integer or character(1) variable
  integer :: m, n
  real(8), device :: alpha ! device or host variable
```

```
  type(c_devptr), device :: A(*)
  integer :: lda
  type(c_devptr), device :: B(*)
  integer :: ldb
  integer :: batchCount
```

```
integer(4) function cublasDtrsmBatched_v2( h, side, uplo, trans, diag, m, n,
alpha, A, lda, B, ldb, batchCount)
  type(cublasHandle) :: h
  integer :: side
  integer :: uplo
  integer :: trans
  integer :: diag
  integer :: m, n
  real(8), device :: alpha ! device or host variable
  type(c_devptr), device :: A(*)
  integer :: lda
  type(c_devptr), device :: B(*)
  integer :: ldb
  integer :: batchCount
```

2.3.42. cublasDmatinvBatched

cublasDmatinvBatched is a short cut of cublasDgetrfBatched plus cublasDgetriBatched. However it only works if n is less than 32. If not, the user has to go through cublasDgetrfBatched and cublasDgetriBatched.

```
integer(4) function cublasDmatinvBatched(h, n, Aarray, lda, Ainv, lda_inv,
info, batchCount)
  type(cublasHandle) :: h
  integer :: n
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: Ainv(*)
  integer :: lda_inv
  integer, device :: info(*)
  integer :: batchCount
```

2.3.43. cublasDgeqrfBatched

DGEQRF computes a QR factorization of a real M-by-N matrix A: A = Q * R.

```
integer(4) function cublasDgeqrfBatched(h, m, n, Aarray, lda, Tau, info,
batchCount)
  type(cublasHandle) :: h
  integer :: m, n
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: Tau(*)
  integer :: info(*)
  integer :: batchCount
```

2.3.44. cublasDgelsBatched

DGELS solves overdetermined or underdetermined real linear systems involving an M-by-N matrix A, or its transpose, using a QR or LQ factorization of A. It is assumed that A has full rank. The following options are provided: 1. If TRANS = 'N' and m >= n: find the least squares solution of an overdetermined system, i.e., solve the least squares problem minimize || B - A*X ||. 2. If TRANS = 'N' and m < n: find the minimum norm solution of an underdetermined system A * X = B. 3. If TRANS = 'T' and m >= n: find the minimum norm solution of an undetermined system A**T * X = B. 4. If TRANS = 'T'

and m < n: find the least squares solution of an overdetermined system, i.e., solve the least squares problem minimize || B - A**T * X ||. Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the M-by-NRHS right hand side matrix B and the N-by-NRHS solution matrix X.

```
integer(4) function cublasDgelsBatched(h, trans, m, n, nrhs, Aarray, lda,
Carray, ldc, info, devinfo, batchCount)
  type(cublasHandle) :: h
  integer :: trans ! integer or character(1) variable
  integer :: m, n, nrhs
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: Carray(*)
  integer :: ldc
  integer :: info(*)
  integer, device :: devinfo(*)
  integer :: batchCount
```

2.3.45. cublasDgemmStridedBatched

DGEMM performs one of the matrix-matrix operations C := alpha*op(A)*op(B) + beta*C, where op(X) is one of op(X) = X or op(X) = X**T, alpha and beta are scalars, and A, B and C are matrices, with op(A) an m by k matrix, op(B) a k by n matrix and C an m by n matrix.

```
integer(4) function cublasDgemmStridedBatched(h, transa, transb, m, n, k,
alpha, Aarray, lda, strideA, Barray, ldb, strideB, beta, Carray, ldc, strideC,
batchCount)
  type(cublasHandle) :: h
  integer :: transa ! integer or character(1) variable
  integer :: transb ! integer or character(1) variable
  integer :: m, n, k
  real(8), device :: alpha ! device or host variable
  real(8), device :: Aarray(*)
  integer :: lda
  integer :: strideA
  real(8), device :: Barray(*)
  integer :: ldb
  integer :: strideB
  real(8), device :: beta ! device or host variable
  real(8), device :: Carray(*)
  integer :: ldc
  integer :: strideC
  integer :: batchCount
```

```
integer(4) function cublasDgemmStridedBatched_v2(h, transa, transb, m, n, k,
alpha,           Aarray, lda, strideA, Barray, ldb, strideB, beta, Carray, ldc,
strideC, batchCount)
  type(cublasHandle) :: h
  integer :: transa
  integer :: transb
  integer :: m, n, k
  real(8), device :: alpha ! device or host variable
  real(8), device :: Aarray(*)
  integer :: lda
  integer :: strideA
  real(8), device :: Barray(*)
  integer :: ldb
  integer :: strideB
  real(8), device :: beta ! device or host variable
  real(8), device :: Carray(*)
  integer :: ldc
  integer :: strideC
  integer :: batchCount
```

2.4. Single Precision Complex Functions and Subroutines

This section contains interfaces to the single precision complex BLAS and cuBLAS functions and subroutines.

2.4.1. icamax

ICAMAX finds the index of the element having the maximum absolute value.

```
integer(4) function icamax(n, x, incx)
  integer :: n
  complex(4), device, dimension(*) :: x ! device or host variable
  integer :: incx
```

```
integer(4) function cublasIcamax(n, x, incx)
  integer :: n
  complex(4), device, dimension(*) :: x
  integer :: incx
```

```
integer(4) function cublasIcamax_v2(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  complex(4), device, dimension(*) :: x
  integer :: incx
  integer, device :: res ! device or host variable
```

2.4.2. icamin

ICAMIN finds the index of the element having the minimum absolute value.

```
integer(4) function icamin(n, x, incx)
  integer :: n
  complex(4), device, dimension(*) :: x ! device or host variable
  integer :: incx
```

```
integer(4) function cublasIcamin(n, x, incx)
  integer :: n
  complex(4), device, dimension(*) :: x
  integer :: incx
```

```
integer(4) function cublasIcamin_v2(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  complex(4), device, dimension(*) :: x
  integer :: incx
  integer, device :: res ! device or host variable
```

2.4.3. scasum

SCASUM takes the sum of the absolute values of a complex vector and returns a single precision result.

```
real(4) function scasum(n, x, incx)
  integer :: n
  complex(4), device, dimension(*) :: x ! device or host variable
  integer :: incx
```

```
real(4) function cublasScasum(n, x, incx)
  integer :: n
```

```
  complex(4), device, dimension(*) :: x
  integer :: incx
```

```
integer(4) function cublasScasum_v2(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  complex(4), device, dimension(*) :: x
  integer :: incx
  real(4), device :: res ! device or host variable
```

2.4.4. caxpy

CAXPY constant times a vector plus a vector.

```
subroutine caxpy(n, a, x, incx, y, incy)
  integer :: n
  complex(4), device :: a ! device or host variable
  complex(4), device, dimension(*) :: x, y ! device or host variable
  integer :: incx, incy
```

```
subroutine cublasCaxpy(n, a, x, incx, y, incy)
  integer :: n
  complex(4), device :: a ! device or host variable
  complex(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

```
integer(4) function cublasCaxpy_v2(h, n, a, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  complex(4), device :: a ! device or host variable
  complex(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

2.4.5. ccopy

CCOPY copies a vector x to a vector y.

```
subroutine ccopy(n, x, incx, y, incy)
  integer :: n
  complex(4), device, dimension(*) :: x, y ! device or host variable
  integer :: incx, incy
```

```
subroutine cublasCcopy(n, x, incx, y, incy)
  integer :: n
  complex(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

```
integer(4) function cublasCcopy_v2(h, n, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  complex(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

2.4.6. cdotc

forms the dot product of two vectors, conjugating the first vector.

```
complex(4) function cdotc(n, x, incx, y, incy)
  integer :: n
  complex(4), device, dimension(*) :: x, y ! device or host variable
  integer :: incx, incy
```

```
complex(4) function cublasCdotc(n, x, incx, y, incy)
  integer :: n
  complex(4), device, dimension(*) :: x, y
```

```
  integer :: incx, incy
```

```
integer(4) function cublasCdotc_v2(h, n, x, incx, y, incy, res)
  type(cublasHandle) :: h
  integer :: n
  complex(4), device, dimension(*) :: x, y
  integer :: incx, incy
  complex(4), device :: res ! device or host variable
```

2.4.7. cdotu

CDOTU forms the dot product of two vectors.

```
complex(4) function cdotu(n, x, incx, y, incy)
  integer :: n
  complex(4), device, dimension(*) :: x, y ! device or host variable
  integer :: incx, incy
```

```
complex(4) function cublasCdotu(n, x, incx, y, incy)
  integer :: n
  complex(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

```
integer(4) function cublasCdotu_v2(h, n, x, incx, y, incy, res)
  type(cublasHandle) :: h
  integer :: n
  complex(4), device, dimension(*) :: x, y
  integer :: incx, incy
  complex(4), device :: res ! device or host variable
```

2.4.8. scnrm2

SCNRM2 returns the euclidean norm of a vector via the function name, so that SCNRM2 := sqrt(x**H*x)

```
real(4) function scnrm2(n, x, incx)
  integer :: n
  complex(4), device, dimension(*) :: x ! device or host variable
  integer :: incx
```

```
real(4) function cublasScnrm2(n, x, incx)
  integer :: n
  complex(4), device, dimension(*) :: x
  integer :: incx
```

```
integer(4) function cublasScnrm2_v2(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  complex(4), device, dimension(*) :: x
  integer :: incx
  real(4), device :: res ! device or host variable
```

2.4.9. crot

CROT applies a plane rotation, where the cos (C) is real and the sin (S) is complex, and the vectors CX and CY are complex.

```
subroutine crot(n, x, incx, y, incy, sc, ss)
  integer :: n
  real(4), device :: sc ! device or host variable
  complex(4), device :: ss ! device or host variable
  complex(4), device, dimension(*) :: x, y ! device or host variable
  integer :: incx, incy
```

```
subroutine cublasCrot(n, x, incx, y, incy, sc, ss)
  integer :: n
```

```
  real(4), device :: sc ! device or host variable
  complex(4), device :: ss ! device or host variable
  complex(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

```
integer(4) function cublasCrot_v2(h, n, x, incx, y, incy, sc, ss)
  type(cublasHandle) :: h
  integer :: n
  real(4), device :: sc ! device or host variable
  complex(4), device :: ss ! device or host variable
  complex(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

2.4.10. csrot

CSROT applies a plane rotation, where the cos and sin (c and s) are real and the vectors cx and cy are complex.

```
subroutine csrot(n, x, incx, y, incy, sc, ss)
  integer :: n
  real(4), device :: sc, ss ! device or host variable
  complex(4), device, dimension(*) :: x, y ! device or host variable
  integer :: incx, incy
```

```
subroutine cublasCsrot(n, x, incx, y, incy, sc, ss)
  integer :: n
  real(4), device :: sc, ss ! device or host variable
  complex(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

```
integer(4) function cublasCsrot_v2(h, n, x, incx, y, incy, sc, ss)
  type(cublasHandle) :: h
  integer :: n
  real(4), device :: sc, ss ! device or host variable
  complex(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

2.4.11. crotg

CROTG determines a complex Givens rotation.

```
subroutine crotg(sa, sb, sc, ss)
  complex(4), device :: sa, sb, ss ! device or host variable
  real(4), device :: sc ! device or host variable
```

```
subroutine cublasCrotg(sa, sb, sc, ss)
  complex(4), device :: sa, sb, ss ! device or host variable
  real(4), device :: sc ! device or host variable
```

```
integer(4) function cublasCrotg_v2(h, sa, sb, sc, ss)
  type(cublasHandle) :: h
  complex(4), device :: sa, sb, ss ! device or host variable
  real(4), device :: sc ! device or host variable
```

2.4.12. cscal

CSCAL scales a vector by a constant.

```
subroutine cscal(n, a, x, incx)
  integer :: n
  complex(4), device :: a ! device or host variable
  complex(4), device, dimension(*) :: x ! device or host variable
  integer :: incx
```

```
subroutine cublasCscal(n, a, x, incx)
  integer :: n
```

```
  complex(4), device :: a ! device or host variable
  complex(4), device, dimension(*) :: x
  integer :: incx
```

```
integer(4) function cublasCscal_v2(h, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: n
  complex(4), device :: a ! device or host variable
  complex(4), device, dimension(*) :: x
  integer :: incx
```

2.4.13. csscal

CSSCAL scales a complex vector by a real constant.

```
subroutine csscal(n, a, x, incx)
  integer :: n
  real(4), device :: a ! device or host variable
  complex(4), device, dimension(*) :: x ! device or host variable
  integer :: incx
```

```
subroutine cublasCsscal(n, a, x, incx)
  integer :: n
  real(4), device :: a ! device or host variable
  complex(4), device, dimension(*) :: x
  integer :: incx
```

```
integer(4) function cublasCsscal_v2(h, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: n
  real(4), device :: a ! device or host variable
  complex(4), device, dimension(*) :: x
  integer :: incx
```

2.4.14. cswap

CSWAP interchanges two vectors.

```
subroutine cswap(n, x, incx, y, incy)
  integer :: n
  complex(4), device, dimension(*) :: x, y ! device or host variable
  integer :: incx, incy
```

```
subroutine cublasCswap(n, x, incx, y, incy)
  integer :: n
  complex(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

```
integer(4) function cublasCswap_v2(h, n, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  complex(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

2.4.15. cgbmv

CGBMV performs one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A**T*x + beta*y, or y := alpha*A**H*x + beta*y, where alpha and beta are scalars, x and y are vectors and A is an m by n band matrix, with kl sub-diagonals and ku super-diagonals.

```
subroutine cgbmv(t, m, n, kl, ku, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: t
  integer :: m, n, kl, ku, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a ! device or host variable
```

```
  complex(4), device, dimension(*) :: x, y ! device or host variable
  complex(4), device :: alpha, beta ! device or host variable
```

```
subroutine cublasCgbmv(t, m, n, kl, ku, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: t
  integer :: m, n, kl, ku, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x, y
  complex(4), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasCgbmv_v2(h, t, m, n, kl, ku, alpha, a, lda, x, incx,
beta, y, incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: m, n, kl, ku, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x, y
  complex(4), device :: alpha, beta ! device or host variable
```

2.4.16. cgemv

CGEMV performs one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A**T*x + beta*y, or y := alpha*A**H*x + beta*y, where alpha and beta are scalars, x and y are vectors and A is an m by n matrix.

```
subroutine cgemv(t, m, n, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: t
  integer :: m, n, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a ! device or host variable
  complex(4), device, dimension(*) :: x, y ! device or host variable
  complex(4), device :: alpha, beta ! device or host variable
```

```
subroutine cublasCgemv(t, m, n, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: t
  integer :: m, n, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x, y
  complex(4), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasCgemv_v2(h, t, m, n, alpha, a, lda, x, incx, beta, y,
incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: m, n, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x, y
  complex(4), device :: alpha, beta ! device or host variable
```

2.4.17. cgerc

CGERC performs the rank 1 operation A := alpha*x*y**H + A, where alpha is a scalar, x is an m element vector, y is an n element vector and A is an m by n matrix.

```
subroutine cgerc(m, n, alpha, x, incx, y, incy, a, lda)
  integer :: m, n, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a ! device or host variable
  complex(4), device, dimension(*) :: x, y ! device or host variable
  complex(4), device :: alpha ! device or host variable
```

```
subroutine cublasCgerc(m, n, alpha, x, incx, y, incy, a, lda)
  integer :: m, n, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x, y
  complex(4), device :: alpha ! device or host variable
```

```
integer(4) function cublasCgerc_v2(h, m, n, alpha, x, incx, y, incy, a, lda)
```

```
  type(cublasHandle) :: h
  integer :: m, n, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x, y
  complex(4), device :: alpha ! device or host variable
```

2.4.18. cgeru

CGERU performs the rank 1 operation A := alpha*x*y**T + A, where alpha is a scalar, x is an m element vector, y is an n element vector and A is an m by n matrix.

```
subroutine cgeru(m, n, alpha, x, incx, y, incy, a, lda)
  integer :: m, n, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a ! device or host variable
  complex(4), device, dimension(*) :: x, y ! device or host variable
  complex(4), device :: alpha ! device or host variable
```

```
subroutine cublasCgeru(m, n, alpha, x, incx, y, incy, a, lda)
  integer :: m, n, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x, y
  complex(4), device :: alpha ! device or host variable
```

```
integer(4) function cublasCgeru_v2(h, m, n, alpha, x, incx, y, incy, a, lda)
  type(cublasHandle) :: h
  integer :: m, n, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x, y
  complex(4), device :: alpha ! device or host variable
```

2.4.19. csymv

CSYMV performs the matrix-vector operation y := alpha*A*x + beta*y, where alpha and beta are scalars, x and y are n element vectors and A is an n by n symmetric matrix.

```
subroutine csymv(uplo, n, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: uplo
  integer :: n, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a ! device or host variable
  complex(4), device, dimension(*) :: x, y ! device or host variable
  complex(4), device :: alpha, beta ! device or host variable
```

```
subroutine cublasCsymv(uplo, n, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: uplo
  integer :: n, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x, y
  complex(4), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasCsymv_v2(h, uplo, n, alpha, a, lda, x, incx, beta, y,
incy)
  type(cublasHandle) :: h
  integer :: uplo
  integer :: n, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x, y
  complex(4), device :: alpha, beta ! device or host variable
```

2.4.20. csyr

CSYR performs the symmetric rank 1 operation A := alpha*x*x**H + A, where alpha is a complex scalar, x is an n element vector and A is an n by n symmetric matrix.

```
subroutine csyr(t, n, alpha, x, incx, a, lda)
  character*1 :: t
```

```
  integer :: n, incx, lda
  complex(4), device, dimension(lda, *) :: a ! device or host variable
  complex(4), device, dimension(*) :: x ! device or host variable
  complex(4), device :: alpha ! device or host variable
```

```
subroutine cublasCsyr(t, n, alpha, x, incx, a, lda)
  character*1 :: t
  integer :: n, incx, lda
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x
  complex(4), device :: alpha ! device or host variable
```

```
integer(4) function cublasCsyr_v2(h, t, n, alpha, x, incx, a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, lda
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x
  complex(4), device :: alpha ! device or host variable
```

2.4.21. csyr2

CSYR2 performs the symmetric rank 2 operation A := alpha*x*y' + alpha*y*x' + A, where alpha is a complex scalar, x and y are n element vectors and A is an n by n SY matrix.

```
subroutine csyr2(t, n, alpha, x, incx, y, incy, a, lda)
  character*1 :: t
  integer :: n, incx, incy, lda
  complex(4), device, dimension(lda, *) :: a ! device or host variable
  complex(4), device, dimension(*) :: x, y ! device or host variable
  complex(4), device :: alpha ! device or host variable
```

```
subroutine cublasCsyr2(t, n, alpha, x, incx, y, incy, a, lda)
  character*1 :: t
  integer :: n, incx, incy, lda
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x, y
  complex(4), device :: alpha ! device or host variable
```

```
integer(4) function cublasCsyr2_v2(h, t, n, alpha, x, incx, y, incy, a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy, lda
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x, y
  complex(4), device :: alpha ! device or host variable
```

2.4.22. ctbmv

CTBMV performs one of the matrix-vector operations x := A*x, or x := A**T*x, or x := A**H*x, where x is an n element vector and A is an n by n unit, or non-unit, upper or lower triangular band matrix, with (k + 1) diagonals.

```
subroutine ctbmv(u, t, d, n, k, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, k, incx, lda
  complex(4), device, dimension(lda, *) :: a ! device or host variable
  complex(4), device, dimension(*) :: x ! device or host variable
```

```
subroutine cublasCtbmv(u, t, d, n, k, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, k, incx, lda
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x
```

```
integer(4) function cublasCtbmv_v2(h, u, t, d, n, k, a, lda, x, incx)
```

```
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, k, incx, lda
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x
```

2.4.23. ctbsv

CTBSV solves one of the systems of equations A*x = b, or A**T*x = b, or A**H*x = b, where b and x are n element vectors and A is an n by 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 routine. Such tests must be performed before calling this routine.

```
subroutine ctbsv(u, t, d, n, k, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, k, incx, lda
  complex(4), device, dimension(lda, *) :: a ! device or host variable
  complex(4), device, dimension(*) :: x ! device or host variable
```

```
subroutine cublasCtbsv(u, t, d, n, k, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, k, incx, lda
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x
```

```
integer(4) function cublasCtbsv_v2(h, u, t, d, n, k, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, k, incx, lda
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x
```

2.4.24. ctpmv

CTPMV performs one of the matrix-vector operations x := A*x, or x := A**T*x, or x := A**H*x, where x is an n element vector and A is an n by n unit, or non-unit, upper or lower triangular matrix, supplied in packed form.

```
subroutine ctpmv(u, t, d, n, a, x, incx)
  character*1 :: u, t, d
  integer :: n, incx
  complex(4), device, dimension(*) :: a, x ! device or host variable
```

```
subroutine cublasCtpmv(u, t, d, n, a, x, incx)
  character*1 :: u, t, d
  integer :: n, incx
  complex(4), device, dimension(*) :: a, x
```

```
integer(4) function cublasCtpmv_v2(h, u, t, d, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx
  complex(4), device, dimension(*) :: a, x
```

2.4.25. ctpsv

CTPSV solves one of the systems of equations A*x = b, or A**T*x = b, or A**H*x = b, where b and x are n element vectors and A is an n by n unit, or non-unit, upper or lower

triangular matrix, supplied in packed form. No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine.

```
subroutine ctpsv(u, t, d, n, a, x, incx)
  character*1 :: u, t, d
  integer :: n, incx
  complex(4), device, dimension(*) :: a, x ! device or host variable
```

```
subroutine cublasCtpsv(u, t, d, n, a, x, incx)
  character*1 :: u, t, d
  integer :: n, incx
  complex(4), device, dimension(*) :: a, x
```

```
integer(4) function cublasCtpsv_v2(h, u, t, d, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx
  complex(4), device, dimension(*) :: a, x
```

2.4.26. ctrmv

CTRMV performs one of the matrix-vector operations x := A*x, or x := A**T*x, or x := A**H*x, where x is an n element vector and A is an n by n unit, or non-unit, upper or lower triangular matrix.

```
subroutine ctrmv(u, t, d, n, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, incx, lda
  complex(4), device, dimension(lda, *) :: a ! device or host variable
  complex(4), device, dimension(*) :: x ! device or host variable
```

```
subroutine cublasCtrmv(u, t, d, n, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, incx, lda
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x
```

```
integer(4) function cublasCtrmv_v2(h, u, t, d, n, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx, lda
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x
```

2.4.27. ctrsv

CTRSV solves one of the systems of equations A*x = b, or A**T*x = b, or A**H*x = b, where b and x are n element vectors and A is an n by n unit, or non-unit, upper or lower triangular matrix. No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine.

```
subroutine ctrsv(u, t, d, n, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, incx, lda
  complex(4), device, dimension(lda, *) :: a ! device or host variable
  complex(4), device, dimension(*) :: x ! device or host variable
```

```
subroutine cublasCtrsv(u, t, d, n, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, incx, lda
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x
```

```
integer(4) function cublasCtrsv_v2(h, u, t, d, n, a, lda, x, incx)
  type(cublasHandle) :: h
```

```
  integer :: u, t, d
  integer :: n, incx, lda
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x
```

2.4.28. chbmv

CHBMV performs the matrix-vector operation y := alpha*A*x + beta*y, where alpha and beta are scalars, x and y are n element vectors and A is an n by n hermitian band matrix, with k super-diagonals.

```
subroutine chbmv(uplo, n, k, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: uplo
  integer :: k, n, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a ! device or host variable
  complex(4), device, dimension(*) :: x, y ! device or host variable
  complex(4), device :: alpha, beta ! device or host variable
```

```
subroutine cublasChbmv(uplo, n, k, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: uplo
  integer :: k, n, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x, y
  complex(4), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasChbmv_v2(h, uplo, n, k, alpha, a, lda, x, incx, beta,
y, incy)
  type(cublasHandle) :: h
  integer :: uplo
  integer :: k, n, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x, y
  complex(4), device :: alpha, beta ! device or host variable
```

2.4.29. chemv

CHEMV performs the matrix-vector operation y := alpha*A*x + beta*y, where alpha and beta are scalars, x and y are n element vectors and A is an n by n hermitian matrix.

```
subroutine chemv(uplo, n, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: uplo
  integer :: n, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a ! device or host variable
  complex(4), device, dimension(*) :: x, y ! device or host variable
  complex(4), device :: alpha, beta ! device or host variable
```

```
subroutine cublasChemv(uplo, n, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: uplo
  integer :: n, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x, y
  complex(4), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasChemv_v2(h, uplo, n, alpha, a, lda, x, incx, beta, y,
incy)
  type(cublasHandle) :: h
  integer :: uplo
  integer :: n, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x, y
  complex(4), device :: alpha, beta ! device or host variable
```

2.4.30. chpmv

CHPMV performs the matrix-vector operation y := alpha*A*x + beta*y, where alpha and beta are scalars, x and y are n element vectors and A is an n by n hermitian matrix, supplied in packed form.

```
subroutine chpmv(uplo, n, alpha, a, x, incx, beta, y, incy)
  character*1 :: uplo
  integer :: n, incx, incy
  complex(4), device, dimension(*) :: a, x, y ! device or host variable
  complex(4), device :: alpha, beta ! device or host variable
```

```
subroutine cublasChpmv(uplo, n, alpha, a, x, incx, beta, y, incy)
  character*1 :: uplo
  integer :: n, incx, incy
  complex(4), device, dimension(*) :: a, x, y
  complex(4), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasChpmv_v2(h, uplo, n, alpha, a, x, incx, beta, y,
incy)
  type(cublasHandle) :: h
  integer :: uplo
  integer :: n, incx, incy
  complex(4), device, dimension(*) :: a, x, y
  complex(4), device :: alpha, beta ! device or host variable
```

2.4.31. cher

CHER performs the hermitian rank 1 operation A := alpha*x*x**H + A, where alpha is a real scalar, x is an n element vector and A is an n by n hermitian matrix.

```
subroutine cher(t, n, alpha, x, incx, a, lda)
  character*1 :: t
  integer :: n, incx, lda
  complex(4), device, dimension(*) :: a, x ! device or host variable
  real(4), device :: alpha ! device or host variable
```

```
subroutine cublasCher(t, n, alpha, x, incx, a, lda)
  character*1 :: t
  integer :: n, incx, lda
  complex(4), device, dimension(*) :: a, x
  real(4), device :: alpha ! device or host variable
```

```
integer(4) function cublasCher_v2(h, t, n, alpha, x, incx, a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, lda
  complex(4), device, dimension(*) :: a, x
  real(4), device :: alpha ! device or host variable
```

2.4.32. cher2

CHER2 performs the hermitian rank 2 operation A := alpha*x*y**H + conjg(alpha)*y*x**H + A, where alpha is a scalar, x and y are n element vectors and A is an n by n hermitian matrix.

```
subroutine cher2(t, n, alpha, x, incx, y, incy, a, lda)
  character*1 :: t
  integer :: n, incx, incy, lda
  complex(4), device, dimension(*) :: a, x, y ! device or host variable
  complex(4), device :: alpha ! device or host variable
```

```
subroutine cublasCher2(t, n, alpha, x, incx, y, incy, a, lda)
```

```
  character*1 :: t
  integer :: n, incx, incy, lda
  complex(4), device, dimension(*) :: a, x, y
  complex(4), device :: alpha ! device or host variable
```

```
integer(4) function cublasCher2_v2(h, t, n, alpha, x, incx, y, incy, a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy, lda
  complex(4), device, dimension(*) :: a, x, y
  complex(4), device :: alpha ! device or host variable
```

2.4.33. chpr

CHPR performs the hermitian rank 1 operation A := alpha*x*x**H + A, where alpha is a real scalar, x is an n element vector and A is an n by n hermitian matrix, supplied in packed form.

```
subroutine chpr(t, n, alpha, x, incx, a)
  character*1 :: t
  integer :: n, incx
  complex(4), device, dimension(*) :: a, x ! device or host variable
  real(4), device :: alpha ! device or host variable
```

```
subroutine cublasChpr(t, n, alpha, x, incx, a)
  character*1 :: t
  integer :: n, incx
  complex(4), device, dimension(*) :: a, x
  real(4), device :: alpha ! device or host variable
```

```
integer(4) function cublasChpr_v2(h, t, n, alpha, x, incx, a)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx
  complex(4), device, dimension(*) :: a, x
  real(4), device :: alpha ! device or host variable
```

2.4.34. chpr2

CHPR2 performs the hermitian rank 2 operation A := alpha*x*y**H + conjg(alpha)*y*x**H + A, where alpha is a scalar, x and y are n element vectors and A is an n by n hermitian matrix, supplied in packed form.

```
subroutine chpr2(t, n, alpha, x, incx, y, incy, a)
  character*1 :: t
  integer :: n, incx, incy
  complex(4), device, dimension(*) :: a, x, y ! device or host variable
  complex(4), device :: alpha ! device or host variable
```

```
subroutine cublasChpr2(t, n, alpha, x, incx, y, incy, a)
  character*1 :: t
  integer :: n, incx, incy
  complex(4), device, dimension(*) :: a, x, y
  complex(4), device :: alpha ! device or host variable
```

```
integer(4) function cublasChpr2_v2(h, t, n, alpha, x, incx, y, incy, a)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy
  complex(4), device, dimension(*) :: a, x, y
  complex(4), device :: alpha ! device or host variable
```

2.4.35. cgemm

CGEMM performs one of the matrix-matrix operations C := alpha*op(A)*op(B) + beta*C, where op(X) is one of op(X) = X or op(X) = X**T or op(X) = X**H, alpha and beta are scalars, and A, B and C are matrices, with op(A) an m by k matrix, op(B) a k by n matrix and C an m by n matrix.

```
subroutine cgemm(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: transa, transb
  integer :: m, n, k, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a ! device or host variable
  complex(4), device, dimension(ldb, *) :: b ! device or host variable
  complex(4), device, dimension(ldc, *) :: c ! device or host variable
  complex(4), device :: alpha, beta ! device or host variable
```

```
subroutine cublasCgemm(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c,
ldc)
  character*1 :: transa, transb
  integer :: m, n, k, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device, dimension(ldc, *) :: c
  complex(4), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasCgemm_v2(h, transa, transb, m, n, k, alpha, a, lda,
b, ldb, beta, c, ldc)
  type(cublasHandle) :: h
  integer :: transa, transb
  integer :: m, n, k, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device, dimension(ldc, *) :: c
  complex(4), device :: alpha, beta ! device or host variable
```

2.4.36. csymm

CSYMM 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 scalars, A is a symmetric matrix and B and C are m by n matrices.

```
subroutine csymm(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: side, uplo
  integer :: m, n, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a ! device or host variable
  complex(4), device, dimension(ldb, *) :: b ! device or host variable
  complex(4), device, dimension(ldc, *) :: c ! device or host variable
  complex(4), device :: alpha, beta ! device or host variable
```

```
subroutine cublasCsymm(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: side, uplo
  integer :: m, n, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device, dimension(ldc, *) :: c
  complex(4), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasCsymm_v2(h, side, uplo, m, n, alpha, a, lda, b, ldb,
beta, c, ldc)
  type(cublasHandle) :: h
  integer :: side, uplo
  integer :: m, n, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device, dimension(ldc, *) :: c
```

```
  complex(4), device :: alpha, beta ! device or host variable
```

2.4.37. csyrk

CSYRK performs one of the symmetric rank k operations C := alpha*A*A**T + beta*C, or C := alpha*A**T*A + beta*C, where alpha and beta are scalars, C is an n by n symmetric matrix and A is an n by k matrix in the first case and a k by n matrix in the second case.

```
subroutine csyrk(uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldc
  complex(4), device, dimension(lda, *) :: a ! device or host variable
  complex(4), device, dimension(ldc, *) :: c ! device or host variable
  complex(4), device :: alpha, beta ! device or host variable
```

```
subroutine cublasCsyrk(uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldc, *) :: c
  complex(4), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasCsyrk_v2(h, uplo, trans, n, k, alpha, a, lda, beta,
c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldc, *) :: c
  complex(4), device :: alpha, beta ! device or host variable
```

2.4.38. csyr2k

CSYR2K 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 scalars, C is an n by n symmetric matrix and A and B are n by k matrices in the first case and k by n matrices in the second case.

```
subroutine csyr2k(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a ! device or host variable
  complex(4), device, dimension(ldb, *) :: b ! device or host variable
  complex(4), device, dimension(ldc, *) :: c ! device or host variable
  complex(4), device :: alpha, beta ! device or host variable
```

```
subroutine cublasCsyr2k(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device, dimension(ldc, *) :: c
  complex(4), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasCsyr2k_v2(h, uplo, trans, n, k, alpha, a, lda, b,
ldb, beta, c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device, dimension(ldc, *) :: c
  complex(4), device :: alpha, beta ! device or host variable
```

2.4.39. csyrkx

CSYRKX performs a variation of the symmetric rank k update C := alpha*A*B**T + beta*C, where alpha and beta are scalars, C is an n by n symmetric matrix stored in lower or upper mode, and A and B are n by k matrices. This routine can be used when B is in such a way that the result is guaranteed to be symmetric. See the CUBLAS documentation for more details.

```
subroutine csyrkx(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a ! device or host variable
  complex(4), device, dimension(ldb, *) :: b ! device or host variable
  complex(4), device, dimension(ldc, *) :: c ! device or host variable
  complex(4), device :: alpha, beta ! device or host variable
```

```
subroutine cublasCsyrkx(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device, dimension(ldc, *) :: c
  complex(4), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasCsyrkx_v2(h, uplo, trans, n, k, alpha, a, lda, b,
ldb, beta, c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device, dimension(ldc, *) :: c
  complex(4), device :: alpha, beta ! device or host variable
```

2.4.40. ctrmm

CTRMM performs one of the matrix-matrix operations B := alpha*op(A)*B, or B := alpha*B*op(A) where alpha is a scalar, B is an m by n matrix, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A**T or op(A) = A**H.

```
subroutine ctrmm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
  character*1 :: side, uplo, transa, diag
  integer :: m, n, lda, ldb
  complex(4), device, dimension(lda, *) :: a ! device or host variable
  complex(4), device, dimension(ldb, *) :: b ! device or host variable
  complex(4), device :: alpha ! device or host variable
```

```
subroutine cublasCtrmm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
  character*1 :: side, uplo, transa, diag
  integer :: m, n, lda, ldb
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device :: alpha ! device or host variable
```

```
integer(4) function cublasCtrmm_v2(h, side, uplo, transa, diag, m, n, alpha, a,
lda, b, ldb, c, ldc)
  type(cublasHandle) :: h
  integer :: side, uplo, transa, diag
  integer :: m, n, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device, dimension(ldc, *) :: c
```

```
  complex(4), device :: alpha ! device or host variable
```

2.4.41. ctrsm

CTRSM solves one of the matrix equations op(A)*X = alpha*B, or X*op(A) = alpha*B, where alpha is a scalar, X and B are m by n matrices, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A**T or op(A) = A**H. The matrix X is overwritten on B.

```
subroutine ctrsm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
  character*1 :: side, uplo, transa, diag
  integer :: m, n, lda, ldb
  complex(4), device, dimension(lda, *) :: a ! device or host variable
  complex(4), device, dimension(ldb, *) :: b ! device or host variable
  complex(4), device :: alpha ! device or host variable
```

```
subroutine cublasCtrsm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
  character*1 :: side, uplo, transa, diag
  integer :: m, n, lda, ldb
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device :: alpha ! device or host variable
```

```
integer(4) function cublasCtrsm_v2(h, side, uplo, transa, diag, m, n, alpha, a,
lda, b, ldb)
  type(cublasHandle) :: h
  integer :: side, uplo, transa, diag
  integer :: m, n, lda, ldb
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device :: alpha ! device or host variable
```

2.4.42. chemm

CHEMM 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 scalars, A is an hermitian matrix and B and C are m by n matrices.

```
subroutine chemm(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: side, uplo
  integer :: m, n, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a ! device or host variable
  complex(4), device, dimension(ldb, *) :: b ! device or host variable
  complex(4), device, dimension(ldc, *) :: c ! device or host variable
  complex(4), device :: alpha, beta ! device or host variable
```

```
subroutine cublasChemm(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: side, uplo
  integer :: m, n, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device, dimension(ldc, *) :: c
  complex(4), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasChemm_v2(h, side, uplo, m, n, alpha, a, lda, b, ldb,
beta, c, ldc)
  type(cublasHandle) :: h
  integer :: side, uplo
  integer :: m, n, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device, dimension(ldc, *) :: c
  complex(4), device :: alpha, beta ! device or host variable
```

2.4.43. cherk

CHERK performs one of the hermitian rank k operations C := alpha*A*A**H + beta*C, or C := alpha*A**H*A + beta*C, where alpha and beta are real scalars, C is an n by n hermitian matrix and A is an n by k matrix in the first case and a k by n matrix in the second case.

```
subroutine cherk(uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldc
  complex(4), device, dimension(lda, *) :: a ! device or host variable
  complex(4), device, dimension(ldc, *) :: c ! device or host variable
  real(4), device :: alpha, beta ! device or host variable
```

```
subroutine cublasCherk(uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldc, *) :: c
  real(4), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasCherk_v2(h, uplo, trans, n, k, alpha, a, lda, beta,
c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldc, *) :: c
  real(4), device :: alpha, beta ! device or host variable
```

2.4.44. cher2k

CHER2K performs one of the hermitian rank 2k operations C := alpha*A*B**H + conjg(alpha)*B*A**H + beta*C, or C := alpha*A**H*B + conjg(alpha)*B**H*A + beta*C, where alpha and beta are scalars with beta real, C is an n by n hermitian matrix and A and B are n by k matrices in the first case and k by n matrices in the second case.

```
subroutine cher2k(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a ! device or host variable
  complex(4), device, dimension(ldb, *) :: b ! device or host variable
  complex(4), device, dimension(ldc, *) :: c ! device or host variable
  complex(4), device :: alpha ! device or host variable
  real(4), device :: beta ! device or host variable
```

```
subroutine cublasCher2k(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device, dimension(ldc, *) :: c
  complex(4), device :: alpha ! device or host variable
  real(4), device :: beta ! device or host variable
```

```
integer(4) function cublasCher2k_v2(h, uplo, trans, n, k, alpha, a, lda, b,
ldb, beta, c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device, dimension(ldc, *) :: c
```

```
  complex(4), device :: alpha ! device or host variable
  real(4), device :: beta ! device or host variable
```

2.4.45. cherkx

CHERKX performs a variation of the hermitian rank k operations C := alpha*A*B**H + beta*C, where alpha and beta are real scalars, C is an n by n hermitian matrix stored in lower or upper mode, and A and B are n by k matrices. See the CUBLAS documentation for more details.

```
subroutine cherkx(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a ! device or host variable
  complex(4), device, dimension(ldb, *) :: b ! device or host variable
  complex(4), device, dimension(ldc, *) :: c ! device or host variable
  complex(4), device :: alpha ! device or host variable
  real(4), device :: beta ! device or host variable
```

```
subroutine cublasCherkx(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device, dimension(ldc, *) :: c
  complex(4), device :: alpha ! device or host variable
  real(4), device :: beta ! device or host variable
```

```
integer(4) function cublasCherkx_v2(h, uplo, trans, n, k, alpha, a, lda, b,
ldb, beta, c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device, dimension(ldc, *) :: c
  complex(4), device :: alpha ! device or host variable
  real(4), device :: beta ! device or host variable
```

2.4.46. cublasCgetrfBatched

CGETRF computes an LU factorization of a general M-by-N matrix A using partial pivoting with row interchanges. The factorization has the form A = P * L * U where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if m > n), and U is upper triangular (upper trapezoidal if m < n). This is the right-looking Level 3 BLAS version of the algorithm.

```
integer(4) function cublasCgetrfBatched(h, n, Aarray, lda, ipvt, info,
batchCount)
  type(cublasHandle) :: h
  integer :: n
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  integer, device :: ipvt(*)
  integer, device :: info(*)
  integer :: batchCount
```

2.4.47. cublasCgetriBatched

CGETRI computes the inverse of a matrix using the LU factorization computed by CGETRF. This method inverts U and then computes inv(A) by solving the system inv(A)*L = inv(U) for inv(A).

```
integer(4) function cublasCgetriBatched(h, n, Aarray, lda, ipvt, Carray, ldc,
info, batchCount)
  type(cublasHandle) :: h
  integer :: n
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  integer, device :: ipvt(*)
  type(c_devptr), device :: Carray(*)
  integer :: ldc
  integer, device :: info(*)
  integer :: batchCount
```

2.4.48. cublasCgetrsBatched

CGETRS solves a system of linear equations A * X = B, A**T * X = B, or A**H * X = B with a general N-by-N matrix A using the LU factorization computed by CGETRF.

```
integer(4) function cublasCgetrsBatched(h, trans, n, nrhs, Aarray, lda, ipvt,
Barray, ldb, info, batchCount)
  type(cublasHandle) :: h
  integer :: trans ! integer or character(1) variable
  integer :: n, nrhs
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  integer, device :: ipvt(*)
  type(c_devptr), device :: Barray(*)
  integer :: ldb
  integer :: info(*)
  integer :: batchCount
```

2.4.49. cublasCgemmBatched

CGEMM performs one of the matrix-matrix operations C := alpha*op(A)*op(B) + beta*C, where op(X) is one of op(X) = X or op(X) = X**T or op(X) = X**H, alpha and beta are scalars, and A, B and C are matrices, with op(A) an m by k matrix, op(B) a k by n matrix and C an m by n matrix.

```
integer(4) function cublasCgemmBatched(h, transa, transb, m, n, k, alpha,
Aarray, lda, Barray, ldb, beta, Carray, ldc, batchCount)
  type(cublasHandle) :: h
  integer :: transa ! integer or character(1) variable
  integer :: transb ! integer or character(1) variable
  integer :: m, n, k
  complex(4), device :: alpha ! device or host variable
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: Barray(*)
  integer :: ldb
  complex(4), device :: beta ! device or host variable
  type(c_devptr), device :: Carray(*)
  integer :: ldc
  integer :: batchCount
```

```
integer(4) function cublasCgemmBatched_v2(h, transa, transb, m, n, k, alpha,
       Aarray, lda, Barray, ldb, beta, Carray, ldc, batchCount)
  type(cublasHandle) :: h
```

```
  integer :: transa
  integer :: transb
  integer :: m, n, k
  complex(4), device :: alpha ! device or host variable
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: Barray(*)
  integer :: ldb
  complex(4), device :: beta ! device or host variable
  type(c_devptr), device :: Carray(*)
  integer :: ldc
  integer :: batchCount
```

2.4.50. cublasCtrsmBatched

CTRSM solves one of the matrix equations op(A)*X = alpha*B, or X*op(A) = alpha*B, where alpha is a scalar, X and B are m by n matrices, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A**T or op(A) = A**H. The matrix X is overwritten on B.

```
integer(4) function cublasCtrsmBatched( h, side, uplo, trans, diag, m, n,
alpha, A, lda, B, ldb, batchCount)
  type(cublasHandle) :: h
  integer :: side ! integer or character(1) variable
  integer :: uplo ! integer or character(1) variable
  integer :: trans ! integer or character(1) variable
  integer :: diag ! integer or character(1) variable
  integer :: m, n
  complex(4), device :: alpha ! device or host variable
  type(c_devptr), device :: A(*)
  integer :: lda
  type(c_devptr), device :: B(*)
  integer :: ldb
  integer :: batchCount
```

```
integer(4) function cublasCtrsmBatched_v2( h, side, uplo, trans, diag, m, n,
alpha, A, lda, B, ldb, batchCount)
  type(cublasHandle) :: h
  integer :: side
  integer :: uplo
  integer :: trans
  integer :: diag
  integer :: m, n
  complex(4), device :: alpha ! device or host variable
  type(c_devptr), device :: A(*)
  integer :: lda
  type(c_devptr), device :: B(*)
  integer :: ldb
  integer :: batchCount
```

2.4.51. cublasCmatinvBatched

cublasCmatinvBatched is a short cut of cublasCgetrfBatched plus cublasCgetriBatched. However it only works if n is less than 32. If not, the user has to go through cublasCgetrfBatched and cublasCgetriBatched.

```
integer(4) function cublasCmatinvBatched(h, n, Aarray, lda, Ainv, lda_inv,
info, batchCount)
  type(cublasHandle) :: h
  integer :: n
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: Ainv(*)
  integer :: lda_inv
```

```
  integer, device :: info(*)
  integer :: batchCount
```

2.4.52. cublasCgeqrfBatched

CGEQRF computes a QR factorization of a complex M-by-N matrix A: A = Q * R.

```
integer(4) function cublasCgeqrfBatched(h, m, n, Aarray, lda, Tau, info,
batchCount)
  type(cublasHandle) :: h
  integer :: m, n
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: Tau(*)
  integer :: info(*)
  integer :: batchCount
```

2.4.53. cublasCgelsBatched

CGELS solves overdetermined or underdetermined complex linear systems involving an M-by-N matrix A, or its conjugate-transpose, using a QR or LQ factorization of A. It is assumed that A has full rank. The following options are provided: 1. If TRANS = 'N' and m >= n: find the least squares solution of an overdetermined system, i.e., solve the least squares problem minimize || B - A*X ||. 2. If TRANS = 'N' and m < n: find the minimum norm solution of an underdetermined system A * X = B. 3. If TRANS = 'C' and m >= n: find the minimum norm solution of an undetermined system A**H * X = B. 4. If TRANS = 'C' and m < n: find the least squares solution of an overdetermined system, i.e., solve the least squares problem minimize || B - A**H * X ||. Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the M-by-NRHS right hand side matrix B and the N-by-NRHS solution matrix X.

```
integer(4) function cublasCgelsBatched(h, trans, m, n, nrhs, Aarray, lda,
Carray, ldc, info, devinfo, batchCount)
  type(cublasHandle) :: h
  integer :: trans ! integer or character(1) variable
  integer :: m, n, nrhs
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: Carray(*)
  integer :: ldc
  integer :: info(*)
  integer, device :: devinfo(*)
  integer :: batchCount
```

2.4.54. cublasCgemmStridedBatched

CGEMM performs one of the matrix-matrix operations C := alpha*op(A)*op(B) + beta*C, where op(X) is one of op(X) = X or op(X) = X**T, alpha and beta are scalars, and A, B and C are matrices, with op(A) an m by k matrix, op(B) a k by n matrix and C an m by n matrix.

```
integer(4) function cublasCgemmStridedBatched(h, transa, transb, m, n, k,
alpha, Aarray, lda, strideA, Barray, ldb, strideB, beta, Carray, ldc, strideC,
batchCount)
  type(cublasHandle) :: h
  integer :: transa ! integer or character(1) variable
  integer :: transb ! integer or character(1) variable
  integer :: m, n, k
  complex(4), device :: alpha ! device or host variable
  complex(4), device :: Aarray(*)
```

```
  integer :: lda
  integer :: strideA
  complex(4), device :: Barray(*)
  integer :: ldb
  integer :: strideB
  complex(4), device :: beta ! device or host variable
  complex(4), device :: Carray(*)
  integer :: ldc
  integer :: strideC
  integer :: batchCount
```

```
integer(4) function cublasCgemmStridedBatched_v2(h, transa, transb, m, n, k,
alpha,           Aarray, lda, strideA, Barray, ldb, strideB, beta, Carray, ldc,
strideC, batchCount)
  type(cublasHandle) :: h
  integer :: transa
  integer :: transb
  integer :: m, n, k
  complex(4), device :: alpha ! device or host variable
  complex(4), device :: Aarray(*)
  integer :: lda
  integer :: strideA
  complex(4), device :: Barray(*)
  integer :: ldb
  integer :: strideB
  complex(4), device :: beta ! device or host variable
  complex(4), device :: Carray(*)
  integer :: ldc
  integer :: strideC
  integer :: batchCount
```

2.5. Double Precision Complex Functions and Subroutines

This section contains interfaces to the double precision complex BLAS and cuBLAS functions and subroutines.

2.5.1. izamax

IZAMAX finds the index of the element having the maximum absolute value.

```
integer(4) function izamax(n, x, incx)
  integer :: n
  complex(8), device, dimension(*) :: x ! device or host variable
  integer :: incx
```

```
integer(4) function cublasIzamax(n, x, incx)
  integer :: n
  complex(8), device, dimension(*) :: x
  integer :: incx
```

```
integer(4) function cublasIzamax_v2(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  complex(8), device, dimension(*) :: x
  integer :: incx
  integer, device :: res ! device or host variable
```

2.5.2. izamin

IZAMIN finds the index of the element having the minimum absolute value.

```
integer(4) function izamin(n, x, incx)
  integer :: n
  complex(8), device, dimension(*) :: x ! device or host variable
  integer :: incx
```

```
integer(4) function cublasIzamin(n, x, incx)
  integer :: n
  complex(8), device, dimension(*) :: x
  integer :: incx
```

```
integer(4) function cublasIzamin_v2(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  complex(8), device, dimension(*) :: x
  integer :: incx
  integer, device :: res ! device or host variable
```

2.5.3. dzasum

DZASUM takes the sum of the absolute values.

```
real(8) function dzasum(n, x, incx)
  integer :: n
  complex(8), device, dimension(*) :: x ! device or host variable
  integer :: incx
```

```
real(8) function cublasDzasum(n, x, incx)
  integer :: n
  complex(8), device, dimension(*) :: x
  integer :: incx
```

```
integer(4) function cublasDzasum_v2(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  complex(8), device, dimension(*) :: x
  integer :: incx
  real(8), device :: res ! device or host variable
```

2.5.4. zaxpy

ZAXPY constant times a vector plus a vector.

```
subroutine zaxpy(n, a, x, incx, y, incy)
  integer :: n
  complex(8), device :: a ! device or host variable
  complex(8), device, dimension(*) :: x, y ! device or host variable
  integer :: incx, incy
```

```
subroutine cublasZaxpy(n, a, x, incx, y, incy)
  integer :: n
  complex(8), device :: a ! device or host variable
  complex(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

```
integer(4) function cublasZaxpy_v2(h, n, a, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  complex(8), device :: a ! device or host variable
  complex(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

2.5.5. zcopy

ZCOPY copies a vector, x, to a vector, y.

```
subroutine zcopy(n, x, incx, y, incy)
  integer :: n
  complex(8), device, dimension(*) :: x, y ! device or host variable
  integer :: incx, incy
```

```
subroutine cublasZcopy(n, x, incx, y, incy)
  integer :: n
  complex(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

```
integer(4) function cublasZcopy_v2(h, n, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  complex(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

2.5.6. zdotc

ZDOTC forms the dot product of a vector.

```
complex(8) function zdotc(n, x, incx, y, incy)
  integer :: n
  complex(8), device, dimension(*) :: x, y ! device or host variable
  integer :: incx, incy
```

```
complex(8) function cublasZdotc(n, x, incx, y, incy)
  integer :: n
  complex(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

```
integer(4) function cublasZdotc_v2(h, n, x, incx, y, incy, res)
  type(cublasHandle) :: h
  integer :: n
  complex(8), device, dimension(*) :: x, y
  integer :: incx, incy
  complex(8), device :: res ! device or host variable
```

2.5.7. zdotu

ZDOTU forms the dot product of two vectors.

```
complex(8) function zdotu(n, x, incx, y, incy)
  integer :: n
  complex(8), device, dimension(*) :: x, y ! device or host variable
  integer :: incx, incy
```

```
complex(8) function cublasZdotu(n, x, incx, y, incy)
  integer :: n
  complex(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

```
integer(4) function cublasZdotu_v2(h, n, x, incx, y, incy, res)
  type(cublasHandle) :: h
  integer :: n
  complex(8), device, dimension(*) :: x, y
  integer :: incx, incy
  complex(8), device :: res ! device or host variable
```

2.5.8. dznrm2

DZNRM2 returns the euclidean norm of a vector via the function name, so that DZNRM2 := sqrt(x**H*x)

```
real(8) function dznrm2(n, x, incx)
  integer :: n
  complex(8), device, dimension(*) :: x ! device or host variable
  integer :: incx
```

```
real(8) function cublasDznrm2(n, x, incx)
  integer :: n
  complex(8), device, dimension(*) :: x
  integer :: incx
```

```
integer(4) function cublasDznrm2_v2(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  complex(8), device, dimension(*) :: x
  integer :: incx
  real(8), device :: res ! device or host variable
```

2.5.9. zrot

ZROT applies a plane rotation, where the cos (C) is real and the sin (S) is complex, and the vectors CX and CY are complex.

```
subroutine zrot(n, x, incx, y, incy, sc, ss)
  integer :: n
  real(8), device :: sc ! device or host variable
  complex(8), device :: ss ! device or host variable
  complex(8), device, dimension(*) :: x, y ! device or host variable
  integer :: incx, incy
```

```
subroutine cublasZrot(n, x, incx, y, incy, sc, ss)
  integer :: n
  real(8), device :: sc ! device or host variable
  complex(8), device :: ss ! device or host variable
  complex(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

```
integer(4) function cublasZrot_v2(h, n, x, incx, y, incy, sc, ss)
  type(cublasHandle) :: h
  integer :: n
  real(8), device :: sc ! device or host variable
  complex(8), device :: ss ! device or host variable
  complex(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

2.5.10. zsrot

ZSROT applies a plane rotation, where the cos and sin (c and s) are real and the vectors cx and cy are complex.

```
subroutine zsrot(n, x, incx, y, incy, sc, ss)
  integer :: n
  real(8), device :: sc, ss ! device or host variable
  complex(8), device, dimension(*) :: x, y ! device or host variable
  integer :: incx, incy
```

```
subroutine cublasZsrot(n, x, incx, y, incy, sc, ss)
  integer :: n
  real(8), device :: sc, ss ! device or host variable
  complex(8), device, dimension(*) :: x, y
```

```
  integer :: incx, incy
```

```
integer(4) function cublasZsrot_v2(h, n, x, incx, y, incy, sc, ss)
  type(cublasHandle) :: h
  integer :: n
  real(8), device :: sc, ss ! device or host variable
  complex(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

2.5.11. zrotg

ZROTG determines a double complex Givens rotation.

```
subroutine zrotg(sa, sb, sc, ss)
  complex(8), device :: sa, sb, ss ! device or host variable
  real(8), device :: sc ! device or host variable
```

```
subroutine cublasZrotg(sa, sb, sc, ss)
  complex(8), device :: sa, sb, ss ! device or host variable
  real(8), device :: sc ! device or host variable
```

```
integer(4) function cublasZrotg_v2(h, sa, sb, sc, ss)
  type(cublasHandle) :: h
  complex(8), device :: sa, sb, ss ! device or host variable
  real(8), device :: sc ! device or host variable
```

2.5.12. zscal

ZSCAL scales a vector by a constant.

```
subroutine zscal(n, a, x, incx)
  integer :: n
  complex(8), device :: a ! device or host variable
  complex(8), device, dimension(*) :: x ! device or host variable
  integer :: incx
```

```
subroutine cublasZscal(n, a, x, incx)
  integer :: n
  complex(8), device :: a ! device or host variable
  complex(8), device, dimension(*) :: x
  integer :: incx
```

```
integer(4) function cublasZscal_v2(h, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: n
  complex(8), device :: a ! device or host variable
  complex(8), device, dimension(*) :: x
  integer :: incx
```

2.5.13. zdscal

ZDSCAL scales a vector by a constant.

```
subroutine zdscal(n, a, x, incx)
  integer :: n
  real(8), device :: a ! device or host variable
  complex(8), device, dimension(*) :: x ! device or host variable
  integer :: incx
```

```
subroutine cublasZdscal(n, a, x, incx)
  integer :: n
  real(8), device :: a ! device or host variable
  complex(8), device, dimension(*) :: x
  integer :: incx
```

```
integer(4) function cublasZdscal_v2(h, n, a, x, incx)
  type(cublasHandle) :: h
```

```
  integer :: n
  real(8), device :: a ! device or host variable
  complex(8), device, dimension(*) :: x
  integer :: incx
```

2.5.14. zswap

ZSWAP interchanges two vectors.

```
subroutine zswap(n, x, incx, y, incy)
  integer :: n
  complex(8), device, dimension(*) :: x, y ! device or host variable
  integer :: incx, incy
```

```
subroutine cublasZswap(n, x, incx, y, incy)
  integer :: n
  complex(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

```
integer(4) function cublasZswap_v2(h, n, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  complex(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

2.5.15. zgbmv

ZGBMV performs one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A**T*x + beta*y, or y := alpha*A**H*x + beta*y, where alpha and beta are scalars, x and y are vectors and A is an m by n band matrix, with kl sub-diagonals and ku super-diagonals.

```
subroutine zgbmv(t, m, n, kl, ku, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: t
  integer :: m, n, kl, ku, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a ! device or host variable
  complex(8), device, dimension(*) :: x, y ! device or host variable
  complex(8), device :: alpha, beta ! device or host variable
```

```
subroutine cublasZgbmv(t, m, n, kl, ku, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: t
  integer :: m, n, kl, ku, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
  complex(8), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasZgbmv_v2(h, t, m, n, kl, ku, alpha, a, lda, x, incx,
beta, y, incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: m, n, kl, ku, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
  complex(8), device :: alpha, beta ! device or host variable
```

2.5.16. zgemv

ZGEMV performs one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A**T*x + beta*y, or y := alpha*A**H*x + beta*y, where alpha and beta are scalars, x and y are vectors and A is an m by n matrix.

```
subroutine zgemv(t, m, n, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: t
  integer :: m, n, lda, incx, incy
```

```
  complex(8), device, dimension(lda, *) :: a ! device or host variable
  complex(8), device, dimension(*) :: x, y ! device or host variable
  complex(8), device :: alpha, beta ! device or host variable
```

```
subroutine cublasZgemv(t, m, n, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: t
  integer :: m, n, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
  complex(8), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasZgemv_v2(h, t, m, n, alpha, a, lda, x, incx, beta, y,
incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: m, n, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
  complex(8), device :: alpha, beta ! device or host variable
```

2.5.17. zgerc

ZGERC performs the rank 1 operation A := alpha*x*y**H + A, where alpha is a scalar, x is an m element vector, y is an n element vector and A is an m by n matrix.

```
subroutine zgerc(m, n, alpha, x, incx, y, incy, a, lda)
  integer :: m, n, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a ! device or host variable
  complex(8), device, dimension(*) :: x, y ! device or host variable
  complex(8), device :: alpha ! device or host variable
```

```
subroutine cublasZgerc(m, n, alpha, x, incx, y, incy, a, lda)
  integer :: m, n, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
  complex(8), device :: alpha ! device or host variable
```

```
integer(4) function cublasZgerc_v2(h, m, n, alpha, x, incx, y, incy, a, lda)
  type(cublasHandle) :: h
  integer :: m, n, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
  complex(8), device :: alpha ! device or host variable
```

2.5.18. zgeru

ZGERU performs the rank 1 operation A := alpha*x*y**T + A, where alpha is a scalar, x is an m element vector, y is an n element vector and A is an m by n matrix.

```
subroutine zgeru(m, n, alpha, x, incx, y, incy, a, lda)
  integer :: m, n, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a ! device or host variable
  complex(8), device, dimension(*) :: x, y ! device or host variable
  complex(8), device :: alpha ! device or host variable
```

```
subroutine cublasZgeru(m, n, alpha, x, incx, y, incy, a, lda)
  integer :: m, n, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
  complex(8), device :: alpha ! device or host variable
```

```
integer(4) function cublasZgeru_v2(h, m, n, alpha, x, incx, y, incy, a, lda)
  type(cublasHandle) :: h
  integer :: m, n, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
```

```
  complex(8), device :: alpha ! device or host variable
```

2.5.19. zsymv

ZSYMV performs the matrix-vector operation y := alpha*A*x + beta*y, where alpha and beta are scalars, x and y are n element vectors and A is an n by n symmetric matrix.

```
subroutine zsymv(uplo, n, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: uplo
  integer :: n, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a ! device or host variable
  complex(8), device, dimension(*) :: x, y ! device or host variable
  complex(8), device :: alpha, beta ! device or host variable
```

```
subroutine cublasZsymv(uplo, n, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: uplo
  integer :: n, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
  complex(8), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasZsymv_v2(h, uplo, n, alpha, a, lda, x, incx, beta, y,
incy)
  type(cublasHandle) :: h
  integer :: uplo
  integer :: n, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
  complex(8), device :: alpha, beta ! device or host variable
```

2.5.20. zsyr

ZSYR performs the symmetric rank 1 operation A := alpha*x*x**H + A, where alpha is a complex scalar, x is an n element vector and A is an n by n symmetric matrix.

```
subroutine zsyr(t, n, alpha, x, incx, a, lda)
  character*1 :: t
  integer :: n, incx, lda
  complex(8), device, dimension(lda, *) :: a ! device or host variable
  complex(8), device, dimension(*) :: x ! device or host variable
  complex(8), device :: alpha ! device or host variable
```

```
subroutine cublasZsyr(t, n, alpha, x, incx, a, lda)
  character*1 :: t
  integer :: n, incx, lda
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x
  complex(8), device :: alpha ! device or host variable
```

```
integer(4) function cublasZsyr_v2(h, t, n, alpha, x, incx, a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, lda
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x
  complex(8), device :: alpha ! device or host variable
```

2.5.21. zsyr2

ZSYR2 performs the symmetric rank 2 operation A := alpha*x*y' + alpha*y*x' + A, where alpha is a complex scalar, x and y are n element vectors and A is an n by n SY matrix.

```
subroutine zsyr2(t, n, alpha, x, incx, y, incy, a, lda)
  character*1 :: t
  integer :: n, incx, incy, lda
```

```
  complex(8), device, dimension(lda, *) :: a ! device or host variable
  complex(8), device, dimension(*) :: x, y ! device or host variable
  complex(8), device :: alpha ! device or host variable
```

```
subroutine cublasZsyr2(t, n, alpha, x, incx, y, incy, a, lda)
  character*1 :: t
  integer :: n, incx, incy, lda
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
  complex(8), device :: alpha ! device or host variable
```

```
integer(4) function cublasZsyr2_v2(h, t, n, alpha, x, incx, y, incy, a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy, lda
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
  complex(8), device :: alpha ! device or host variable
```

2.5.22. ztbmv

ZTBMV performs one of the matrix-vector operations x := A*x, or x := A**T*x, or x := A**H*x, where x is an n element vector and A is an n by n unit, or non-unit, upper or lower triangular band matrix, with (k + 1) diagonals.

```
subroutine ztbmv(u, t, d, n, k, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, k, incx, lda
  complex(8), device, dimension(lda, *) :: a ! device or host variable
  complex(8), device, dimension(*) :: x ! device or host variable
```

```
subroutine cublasZtbmv(u, t, d, n, k, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, k, incx, lda
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x
```

```
integer(4) function cublasZtbmv_v2(h, u, t, d, n, k, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, k, incx, lda
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x
```

2.5.23. ztbsv

ZTBSV solves one of the systems of equations A*x = b, or A**T*x = b, or A**H*x = b, where b and x are n element vectors and A is an n by 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 routine. Such tests must be performed before calling this routine.

```
subroutine ztbsv(u, t, d, n, k, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, k, incx, lda
  complex(8), device, dimension(lda, *) :: a ! device or host variable
  complex(8), device, dimension(*) :: x ! device or host variable
```

```
subroutine cublasZtbsv(u, t, d, n, k, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, k, incx, lda
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x
```

```
integer(4) function cublasZtbsv_v2(h, u, t, d, n, k, a, lda, x, incx)
```

```
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, k, incx, lda
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x
```

2.5.24. ztpmv

ZTPMV performs one of the matrix-vector operations x := A*x, or x := A**T*x, or x := A**H*x, where x is an n element vector and A is an n by n unit, or non-unit, upper or lower triangular matrix, supplied in packed form.

```
subroutine ztpmv(u, t, d, n, a, x, incx)
  character*1 :: u, t, d
  integer :: n, incx
  complex(8), device, dimension(*) :: a, x ! device or host variable
```

```
subroutine cublasZtpmv(u, t, d, n, a, x, incx)
  character*1 :: u, t, d
  integer :: n, incx
  complex(8), device, dimension(*) :: a, x
```

```
integer(4) function cublasZtpmv_v2(h, u, t, d, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx
  complex(8), device, dimension(*) :: a, x
```

2.5.25. ztpsv

ZTPSV solves one of the systems of equations A*x = b, or A**T*x = b, or A**H*x = b, where b and x are n element vectors and A is an n by n unit, or non-unit, upper or lower triangular matrix, supplied in packed form. No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine.

```
subroutine ztpsv(u, t, d, n, a, x, incx)
  character*1 :: u, t, d
  integer :: n, incx
  complex(8), device, dimension(*) :: a, x ! device or host variable
```

```
subroutine cublasZtpsv(u, t, d, n, a, x, incx)
  character*1 :: u, t, d
  integer :: n, incx
  complex(8), device, dimension(*) :: a, x
```

```
integer(4) function cublasZtpsv_v2(h, u, t, d, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx
  complex(8), device, dimension(*) :: a, x
```

2.5.26. ztrmv

ZTRMV performs one of the matrix-vector operations x := A*x, or x := A**T*x, or x := A**H*x, where x is an n element vector and A is an n by n unit, or non-unit, upper or lower triangular matrix.

```
subroutine ztrmv(u, t, d, n, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, incx, lda
  complex(8), device, dimension(lda, *) :: a ! device or host variable
```

```
  complex(8), device, dimension(*) :: x ! device or host variable
```

```
subroutine cublasZtrmv(u, t, d, n, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, incx, lda
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x
```

```
integer(4) function cublasZtrmv_v2(h, u, t, d, n, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx, lda
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x
```

2.5.27. ztrsv

ZTRSV solves one of the systems of equations A*x = b, or A**T*x = b, or A**H*x = b, where b and x are n element vectors and A is an n by n unit, or non-unit, upper or lower triangular matrix. No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine.

```
subroutine ztrsv(u, t, d, n, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, incx, lda
  complex(8), device, dimension(lda, *) :: a ! device or host variable
  complex(8), device, dimension(*) :: x ! device or host variable
```

```
subroutine cublasZtrsv(u, t, d, n, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, incx, lda
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x
```

```
integer(4) function cublasZtrsv_v2(h, u, t, d, n, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx, lda
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x
```

2.5.28. zhbmv

ZHBMV performs the matrix-vector operation y := alpha*A*x + beta*y, where alpha and beta are scalars, x and y are n element vectors and A is an n by n hermitian band matrix, with k super-diagonals.

```
subroutine zhbmv(uplo, n, k, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: uplo
  integer :: k, n, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a ! device or host variable
  complex(8), device, dimension(*) :: x, y ! device or host variable
  complex(8), device :: alpha, beta ! device or host variable
```

```
subroutine cublasZhbmv(uplo, n, k, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: uplo
  integer :: k, n, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
  complex(8), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasZhbmv_v2(h, uplo, n, k, alpha, a, lda, x, incx, beta,
y, incy)
  type(cublasHandle) :: h
  integer :: uplo
```

```
  integer :: k, n, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
  complex(8), device :: alpha, beta ! device or host variable
```

2.5.29. zhemv

ZHEMV performs the matrix-vector operation y := alpha*A*x + beta*y, where alpha and beta are scalars, x and y are n element vectors and A is an n by n hermitian matrix.

```
subroutine zhemv(uplo, n, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: uplo
  integer :: n, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a ! device or host variable
  complex(8), device, dimension(*) :: x, y ! device or host variable
  complex(8), device :: alpha, beta ! device or host variable
```

```
subroutine cublasZhemv(uplo, n, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: uplo
  integer :: n, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
  complex(8), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasZhemv_v2(h, uplo, n, alpha, a, lda, x, incx, beta, y,
incy)
  type(cublasHandle) :: h
  integer :: uplo
  integer :: n, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
  complex(8), device :: alpha, beta ! device or host variable
```

2.5.30. zhpmv

ZHPMV performs the matrix-vector operation y := alpha*A*x + beta*y, where alpha and beta are scalars, x and y are n element vectors and A is an n by n hermitian matrix, supplied in packed form.

```
subroutine zhpmv(uplo, n, alpha, a, x, incx, beta, y, incy)
  character*1 :: uplo
  integer :: n, incx, incy
  complex(8), device, dimension(*) :: a, x, y ! device or host variable
  complex(8), device :: alpha, beta ! device or host variable
```

```
subroutine cublasZhpmv(uplo, n, alpha, a, x, incx, beta, y, incy)
  character*1 :: uplo
  integer :: n, incx, incy
  complex(8), device, dimension(*) :: a, x, y
  complex(8), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasZhpmv_v2(h, uplo, n, alpha, a, x, incx, beta, y,
incy)
  type(cublasHandle) :: h
  integer :: uplo
  integer :: n, incx, incy
  complex(8), device, dimension(*) :: a, x, y
  complex(8), device :: alpha, beta ! device or host variable
```

2.5.31. zher

ZHER performs the hermitian rank 1 operation A := alpha*x*x**H + A, where alpha is a real scalar, x is an n element vector and A is an n by n hermitian matrix.

```
subroutine zher(t, n, alpha, x, incx, a, lda)
  character*1 :: t
  integer :: n, incx, lda
  complex(8), device, dimension(*) :: a, x ! device or host variable
  real(8), device :: alpha ! device or host variable
```

```
subroutine cublasZher(t, n, alpha, x, incx, a, lda)
  character*1 :: t
  integer :: n, incx, lda
  complex(8), device, dimension(*) :: a, x
  real(8), device :: alpha ! device or host variable
```

```
integer(4) function cublasZher_v2(h, t, n, alpha, x, incx, a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, lda
  complex(8), device, dimension(*) :: a, x
  real(8), device :: alpha ! device or host variable
```

2.5.32. zher2

ZHER2 performs the hermitian rank 2 operation A := alpha*x*y**H + conjg(alpha)*y*x**H + A, where alpha is a scalar, x and y are n element vectors and A is an n by n hermitian matrix.

```
subroutine zher2(t, n, alpha, x, incx, y, incy, a, lda)
  character*1 :: t
  integer :: n, incx, incy
  complex(8), device, dimension(*) :: a, x, y ! device or host variable
  complex(8), device :: alpha ! device or host variable
```

```
subroutine cublasZher2(t, n, alpha, x, incx, y, incy, a, lda)
  character*1 :: t
  integer :: n, incx, incy, lda
  complex(8), device, dimension(*) :: a, x, y
  complex(8), device :: alpha ! device or host variable
```

```
integer(4) function cublasZher2_v2(h, t, n, alpha, x, incx, y, incy, a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy, lda
  complex(8), device, dimension(*) :: a, x, y
  complex(8), device :: alpha ! device or host variable
```

2.5.33. zhpr

ZHPR performs the hermitian rank 1 operation A := alpha*x*x**H + A, where alpha is a real scalar, x is an n element vector and A is an n by n hermitian matrix, supplied in packed form.

```
subroutine zhpr(t, n, alpha, x, incx, a)
  character*1 :: t
  integer :: n, incx
  complex(8), device, dimension(*) :: a, x ! device or host variable
  real(8), device :: alpha ! device or host variable
```

```
subroutine cublasZhpr(t, n, alpha, x, incx, a)
  character*1 :: t
```

```
  integer :: n, incx
  complex(8), device, dimension(*) :: a, x
  real(8), device :: alpha ! device or host variable
```

```
integer(4) function cublasZhpr_v2(h, t, n, alpha, x, incx, a)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx
  complex(8), device, dimension(*) :: a, x
  real(8), device :: alpha ! device or host variable
```

2.5.34. zhpr2

ZHPR2 performs the hermitian rank 2 operation A := alpha*x*y**H + conjg(alpha)*y*x**H + A, where alpha is a scalar, x and y are n element vectors and A is an n by n hermitian matrix, supplied in packed form.

```
subroutine zhpr2(t, n, alpha, x, incx, y, incy, a)
  character*1 :: t
  integer :: n, incx, incy
  complex(8), device, dimension(*) :: a, x, y ! device or host variable
  complex(8), device :: alpha ! device or host variable
```

```
subroutine cublasZhpr2(t, n, alpha, x, incx, y, incy, a)
  character*1 :: t
  integer :: n, incx, incy
  complex(8), device, dimension(*) :: a, x, y
  complex(8), device :: alpha ! device or host variable
```

```
integer(4) function cublasZhpr2_v2(h, t, n, alpha, x, incx, y, incy, a)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy
  complex(8), device, dimension(*) :: a, x, y
  complex(8), device :: alpha ! device or host variable
```

2.5.35. zgemm

ZGEMM performs one of the matrix-matrix operations C := alpha*op(A)*op(B) + beta*C, where op(X) is one of op(X) = X or op(X) = X**T or op(X) = X**H, alpha and beta are scalars, and A, B and C are matrices, with op(A) an m by k matrix, op(B) a k by n matrix and C an m by n matrix.

```
subroutine zgemm(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: transa, transb
  integer :: m, n, k, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a ! device or host variable
  complex(8), device, dimension(ldb, *) :: b ! device or host variable
  complex(8), device, dimension(ldc, *) :: c ! device or host variable
  complex(8), device :: alpha, beta ! device or host variable
```

```
subroutine cublasZgemm(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c,
ldc)
  character*1 :: transa, transb
  integer :: m, n, k, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device, dimension(ldc, *) :: c
  complex(8), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasZgemm_v2(h, transa, transb, m, n, k, alpha, a, lda,
b, ldb, beta, c, ldc)
  type(cublasHandle) :: h
  integer :: transa, transb
  integer :: m, n, k, lda, ldb, ldc
```

```
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device, dimension(ldc, *) :: c
  complex(8), device :: alpha, beta ! device or host variable
```

2.5.36. zsymm

ZSYMM 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 scalars, A is a symmetric matrix and B and C are m by n matrices.

```
subroutine zsymm(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: side, uplo
  integer :: m, n, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a ! device or host variable
  complex(8), device, dimension(ldb, *) :: b ! device or host variable
  complex(8), device, dimension(ldc, *) :: c ! device or host variable
  complex(8), device :: alpha, beta ! device or host variable
```

```
subroutine cublasZsymm(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: side, uplo
  integer :: m, n, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device, dimension(ldc, *) :: c
  complex(8), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasZsymm_v2(h, side, uplo, m, n, alpha, a, lda, b, ldb,
beta, c, ldc)
  type(cublasHandle) :: h
  integer :: side, uplo
  integer :: m, n, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device, dimension(ldc, *) :: c
  complex(8), device :: alpha, beta ! device or host variable
```

2.5.37. zsyrk

ZSYRK performs one of the symmetric rank k operations C := alpha*A*A**T + beta*C, or C := alpha*A**T*A + beta*C, where alpha and beta are scalars, C is an n by n symmetric matrix and A is an n by k matrix in the first case and a k by n matrix in the second case.

```
subroutine zsyrk(uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldc
  complex(8), device, dimension(lda, *) :: a ! device or host variable
  complex(8), device, dimension(ldc, *) :: c ! device or host variable
  complex(8), device :: alpha, beta ! device or host variable
```

```
subroutine cublasZsyrk(uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldc, *) :: c
  complex(8), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasZsyrk_v2(h, uplo, trans, n, k, alpha, a, lda, beta,
c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldc, *) :: c
  complex(8), device :: alpha, beta ! device or host variable
```

2.5.38. zsyr2k

ZSYR2K 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 scalars, C is an n by n symmetric matrix and A and B are n by k matrices in the first case and k by n matrices in the second case.

```
subroutine zsyr2k(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a ! device or host variable
  complex(8), device, dimension(ldb, *) :: b ! device or host variable
  complex(8), device, dimension(ldc, *) :: c ! device or host variable
  complex(8), device :: alpha, beta ! device or host variable
```

```
subroutine cublasZsyr2k(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device, dimension(ldc, *) :: c
  complex(8), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasZsyr2k_v2(h, uplo, trans, n, k, alpha, a, lda, b,
ldb, beta, c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device, dimension(ldc, *) :: c
  complex(8), device :: alpha, beta ! device or host variable
```

2.5.39. zsyrkx

ZSYRKX performs a variation of the symmetric rank k update C := alpha*A*B**T + beta*C, where alpha and beta are scalars, C is an n by n symmetric matrix stored in lower or upper mode, and A and B are n by k matrices. This routine can be used when B is in such a way that the result is guaranteed to be symmetric. See the CUBLAS documentation for more details.

```
subroutine zsyrkx(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a ! device or host variable
  complex(8), device, dimension(ldb, *) :: b ! device or host variable
  complex(8), device, dimension(ldc, *) :: c ! device or host variable
  complex(8), device :: alpha, beta ! device or host variable
```

```
subroutine cublasZsyrkx(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device, dimension(ldc, *) :: c
  complex(8), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasZsyrkx_v2(h, uplo, trans, n, k, alpha, a, lda, b,
ldb, beta, c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a
```

```
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device, dimension(ldc, *) :: c
  complex(8), device :: alpha, beta ! device or host variable
```

2.5.40. ztrmm

ZTRMM performs one of the matrix-matrix operations B := alpha*op(A)*B, or B := alpha*B*op(A) where alpha is a scalar, B is an m by n matrix, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A**T or op(A) = A**H.

```
subroutine ztrmm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
  character*1 :: side, uplo, transa, diag
  integer :: m, n, lda, ldb
  complex(8), device, dimension(lda, *) :: a ! device or host variable
  complex(8), device, dimension(ldb, *) :: b ! device or host variable
  complex(8), device :: alpha ! device or host variable
```

```
subroutine cublasZtrmm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
  character*1 :: side, uplo, transa, diag
  integer :: m, n, lda, ldb
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device :: alpha ! device or host variable
```

```
integer(4) function cublasZtrmm_v2(h, side, uplo, transa, diag, m, n, alpha, a,
lda, b, ldb, c, ldc)
  type(cublasHandle) :: h
  integer :: side, uplo, transa, diag
  integer :: m, n, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device, dimension(ldc, *) :: c
  complex(8), device :: alpha ! device or host variable
```

2.5.41. ztrsm

ZTRSM solves one of the matrix equations op(A)*X = alpha*B, or X*op(A) = alpha*B, where alpha is a scalar, X and B are m by n matrices, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A**T or op(A) = A**H. The matrix X is overwritten on B.

```
subroutine ztrsm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
  character*1 :: side, uplo, transa, diag
  integer :: m, n, lda, ldb
  complex(8), device, dimension(lda, *) :: a ! device or host variable
  complex(8), device, dimension(ldb, *) :: b ! device or host variable
  complex(8), device :: alpha ! device or host variable
```

```
subroutine cublasZtrsm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
  character*1 :: side, uplo, transa, diag
  integer :: m, n, lda, ldb
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device :: alpha ! device or host variable
```

```
integer(4) function cublasZtrsm_v2(h, side, uplo, transa, diag, m, n, alpha, a,
lda, b, ldb)
  type(cublasHandle) :: h
  integer :: side, uplo, transa, diag
  integer :: m, n, lda, ldb
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device :: alpha ! device or host variable
```

2.5.42. zhemm

ZHEMM 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 scalars, A is an hermitian matrix and B and C are m by n matrices.

```
subroutine zhemm(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: side, uplo
  integer :: m, n, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a ! device or host variable
  complex(8), device, dimension(ldb, *) :: b ! device or host variable
  complex(8), device, dimension(ldc, *) :: c ! device or host variable
  complex(8), device :: alpha, beta ! device or host variable
```

```
subroutine cublasZhemm(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: side, uplo
  integer :: m, n, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device, dimension(ldc, *) :: c
  complex(8), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasZhemm_v2(h, side, uplo, m, n, alpha, a, lda, b, ldb,
beta, c, ldc)
  type(cublasHandle) :: h
  integer :: side, uplo
  integer :: m, n, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device, dimension(ldc, *) :: c
  complex(8), device :: alpha, beta ! device or host variable
```

2.5.43. zherk

ZHERK performs one of the hermitian rank k operations C := alpha*A*A**H + beta*C, or C := alpha*A**H*A + beta*C, where alpha and beta are real scalars, C is an n by n hermitian matrix and A is an n by k matrix in the first case and a k by n matrix in the second case.

```
subroutine zherk(uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldc
  complex(8), device, dimension(lda, *) :: a ! device or host variable
  complex(8), device, dimension(ldc, *) :: c ! device or host variable
  real(8), device :: alpha, beta ! device or host variable
```

```
subroutine cublasZherk(uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldc, *) :: c
  real(8), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasZherk_v2(h, uplo, trans, n, k, alpha, a, lda, beta,
c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldc, *) :: c
  real(8), device :: alpha, beta ! device or host variable
```

2.5.44. zher2k

ZHER2K performs one of the hermitian rank 2k operations C := alpha*A*B**H + conjg(alpha)*B*A**H + beta*C, or C := alpha*A**H*B + conjg(alpha)*B**H*A + beta*C, where alpha and beta are scalars with beta real, C is an n by n hermitian matrix and A and B are n by k matrices in the first case and k by n matrices in the second case.

```
subroutine zher2k(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a ! device or host variable
  complex(8), device, dimension(ldb, *) :: b ! device or host variable
  complex(8), device, dimension(ldc, *) :: c ! device or host variable
  complex(8), device :: alpha ! device or host variable
  real(8), device :: beta ! device or host variable
```

```
subroutine cublasZher2k(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device, dimension(ldc, *) :: c
  complex(8), device :: alpha ! device or host variable
  real(8), device :: beta ! device or host variable
```

```
integer(4) function cublasZher2k_v2(h, uplo, trans, n, k, alpha, a, lda, b,
ldb, beta, c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device, dimension(ldc, *) :: c
  complex(8), device :: alpha ! device or host variable
  real(8), device :: beta ! device or host variable
```

2.5.45. zherkx

ZHERKX performs a variation of the hermitian rank k operations C := alpha*A*B**H + beta*C, where alpha and beta are real scalars, C is an n by n hermitian matrix stored in lower or upper mode, and A and B are n by k matrices. See the CUBLAS documentation for more details.

```
subroutine zherkx(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a ! device or host variable
  complex(8), device, dimension(ldb, *) :: b ! device or host variable
  complex(8), device, dimension(ldc, *) :: c ! device or host variable
  complex(8), device :: alpha ! device or host variable
  real(8), device :: beta ! device or host variable
```

```
subroutine cublasZherkx(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device, dimension(ldc, *) :: c
  complex(8), device :: alpha ! device or host variable
  real(8), device :: beta ! device or host variable
```

```
integer(4) function cublasZherkx_v2(h, uplo, trans, n, k, alpha, a, lda, b,
ldb, beta, c, ldc)
```

```
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device, dimension(ldc, *) :: c
  complex(8), device :: alpha ! device or host variable
  real(8), device :: beta ! device or host variable
```

2.5.46. cublasZgetrfBatched

ZGETRF computes an LU factorization of a general M-by-N matrix A using partial pivoting with row interchanges. The factorization has the form A = P * L * U where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if m > n), and U is upper triangular (upper trapezoidal if m < n). This is the right-looking Level 3 BLAS version of the algorithm.

```
integer(4) function cublasZgetrfBatched(h, n, Aarray, lda, ipvt, info,
batchCount)
  type(cublasHandle) :: h
  integer :: n
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  integer, device :: ipvt(*)
  integer, device :: info(*)
  integer :: batchCount
```

2.5.47. cublasZgetriBatched

ZGETRI computes the inverse of a matrix using the LU factorization computed by ZGETRF. This method inverts U and then computes inv(A) by solving the system inv(A)*L = inv(U) for inv(A).

```
integer(4) function cublasZgetriBatched(h, n, Aarray, lda, ipvt, Carray, ldc,
info, batchCount)
  type(cublasHandle) :: h
  integer :: n
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  integer, device :: ipvt(*)
  type(c_devptr), device :: Carray(*)
  integer :: ldc
  integer, device :: info(*)
  integer :: batchCount
```

2.5.48. cublasZgetrsBatched

ZGETRS solves a system of linear equations A * X = B, A**T * X = B, or A**H * X = B with a general N-by-N matrix A using the LU factorization computed by ZGETRF.

```
integer(4) function cublasZgetrsBatched(h, trans, n, nrhs, Aarray, lda, ipvt,
Barray, ldb, info, batchCount)
  type(cublasHandle) :: h
  integer :: trans ! integer or character(1) variable
  integer :: n, nrhs
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  integer, device :: ipvt(*)
  type(c_devptr), device :: Barray(*)
  integer :: ldb
  integer :: info(*)
  integer :: batchCount
```

2.5.49. cublasZgemmBatched

ZGEMM performs one of the matrix-matrix operations C := alpha*op(A)*op(B) + beta*C, where op(X) is one of op(X) = X or op(X) = X**T or op(X) = X**H, alpha and beta are scalars, and A, B and C are matrices, with op(A) an m by k matrix, op(B) a k by n matrix and C an m by n matrix.

```
integer(4) function cublasZgemmBatched(h, transa, transb, m, n, k, alpha,
Aarray, lda, Barray, ldb, beta, Carray, ldc, batchCount)
  type(cublasHandle) :: h
  integer :: transa ! integer or character(1) variable
  integer :: transb ! integer or character(1) variable
  integer :: m, n, k
  complex(8), device :: alpha ! device or host variable
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: Barray(*)
  integer :: ldb
  complex(8), device :: beta ! device or host variable
  type(c_devptr), device :: Carray(*)
  integer :: ldc
  integer :: batchCount
```

```
integer(4) function cublasZgemmBatched_v2(h, transa, transb, m, n, k, alpha,
       Aarray, lda, Barray, ldb, beta, Carray, ldc, batchCount)
  type(cublasHandle) :: h
  integer :: transa
  integer :: transb
  integer :: m, n, k
  complex(8), device :: alpha ! device or host variable
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: Barray(*)
  integer :: ldb
  complex(8), device :: beta ! device or host variable
  type(c_devptr), device :: Carray(*)
  integer :: ldc
  integer :: batchCount
```

2.5.50. cublasZtrsmBatched

ZTRSM solves one of the matrix equations op(A)*X = alpha*B, or X*op(A) = alpha*B, where alpha is a scalar, X and B are m by n matrices, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A**T or op(A) = A**H. The matrix X is overwritten on B.

```
integer(4) function cublasZtrsmBatched( h, side, uplo, trans, diag, m, n,
alpha, A, lda, B, ldb, batchCount)
  type(cublasHandle) :: h
  integer :: side ! integer or character(1) variable
  integer :: uplo ! integer or character(1) variable
  integer :: trans ! integer or character(1) variable
  integer :: diag ! integer or character(1) variable
  integer :: m, n
  complex(8), device :: alpha ! device or host variable
  type(c_devptr), device :: A(*)
  integer :: lda
  type(c_devptr), device :: B(*)
  integer :: ldb
  integer :: batchCount
```

```
integer(4) function cublasZtrsmBatched_v2( h, side, uplo, trans, diag, m, n,
alpha, A, lda, B, ldb, batchCount)
```

```
  type(cublasHandle) :: h
  integer :: side
  integer :: uplo
  integer :: trans
  integer :: diag
  integer :: m, n
  complex(8), device :: alpha ! device or host variable
  type(c_devptr), device :: A(*)
  integer :: lda
  type(c_devptr), device :: B(*)
  integer :: ldb
  integer :: batchCount
```

2.5.51. cublasZmatinvBatched

cublasZmatinvBatched is a short cut of cublasZgetrfBatched plus cublasZgetriBatched. However it only works if n is less than 32. If not, the user has to go through cublasZgetrfBatched and cublasZgetriBatched.

```
integer(4) function cublasZmatinvBatched(h, n, Aarray, lda, Ainv, lda_inv,
info, batchCount)
  type(cublasHandle) :: h
  integer :: n
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: Ainv(*)
  integer :: lda_inv
  integer, device :: info(*)
  integer :: batchCount
```

2.5.52. cublasZgeqrfBatched

ZGEQRF computes a QR factorization of a complex M-by-N matrix A: A = Q * R.

```
integer(4) function cublasZgeqrfBatched(h, m, n, Aarray, lda, Tau, info,
batchCount)
  type(cublasHandle) :: h
  integer :: m, n
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: Tau(*)
  integer :: info(*)
  integer :: batchCount
```

2.5.53. cublasZgelsBatched

ZGELS solves overdetermined or underdetermined complex linear systems involving an M-by-N matrix A, or its conjugate-transpose, using a QR or LQ factorization of A. It is assumed that A has full rank. The following options are provided: 1. If TRANS = 'N' and m >= n: find the least squares solution of an overdetermined system, i.e., solve the least squares problem minimize || B - A*X ||. 2. If TRANS = 'N' and m < n: find the minimum norm solution of an underdetermined system A * X = B. 3. If TRANS = 'C' and m >= n: find the minimum norm solution of an undetermined system A**H * X = B. 4. If TRANS = 'C' and m < n: find the least squares solution of an overdetermined system, i.e., solve the least squares problem minimize || B - A**H * X ||. Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the M-by-NRHS right hand side matrix B and the N-by-NRHS solution matrix X.

```
integer(4) function cublasZgelsBatched(h, trans, m, n, nrhs, Aarray, lda,
Carray, ldc, info, devinfo, batchCount)
```

```
  type(cublasHandle) :: h
  integer :: trans ! integer or character(1) variable
  integer :: m, n, nrhs
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: Carray(*)
  integer :: ldc
  integer :: info(*)
  integer, device :: devinfo(*)
  integer :: batchCount
```

2.5.54. cublasZgemmStridedBatched

ZGEMM performs one of the matrix-matrix operations C := alpha*op(A)*op(B) + beta*C, where op(X) is one of op(X) = X or op(X) = X**T, alpha and beta are scalars, and A, B and C are matrices, with op(A) an m by k matrix, op(B) a k by n matrix and C an m by n matrix.

```
integer(4) function cublasZgemmStridedBatched(h, transa, transb, m, n, k,
alpha, Aarray, lda, strideA, Barray, ldb, strideB, beta, Carray, ldc, strideC,
batchCount)
  type(cublasHandle) :: h
  integer :: transa ! integer or character(1) variable
  integer :: transb ! integer or character(1) variable
  integer :: m, n, k
  complex(8), device :: alpha ! device or host variable
  complex(8), device :: Aarray(*)
  integer :: lda
  integer :: strideA
  complex(8), device :: Barray(*)
  integer :: ldb
  integer :: strideB
  complex(8), device :: beta ! device or host variable
  complex(8), device :: Carray(*)
  integer :: ldc
  integer :: strideC
  integer :: batchCount
```

```
integer(4) function cublasZgemmStridedBatched_v2(h, transa, transb, m, n, k,
alpha,           Aarray, lda, strideA, Barray, ldb, strideB, beta, Carray, ldc,
strideC, batchCount)
  type(cublasHandle) :: h
  integer :: transa
  integer :: transb
  integer :: m, n, k
  complex(8), device :: alpha ! device or host variable
  complex(8), device :: Aarray(*)
  integer :: lda
  integer :: strideA
  complex(8), device :: Barray(*)
  integer :: ldb
  integer :: strideB
  complex(8), device :: beta ! device or host variable
  complex(8), device :: Carray(*)
  integer :: ldc
  integer :: strideC
  integer :: batchCount
```

2.6. CUBLAS V2 Module Functions

This section contains interfaces to the cuBLAS V2 Module Functions. Users can access this module by inserting the line **`use cublas_v2`** into the program unit. One major

difference in the **cublas_v2** versus the **cublas** module is the cublas entry points, such as **cublasIsamax** are changed to take the handle as the first argument. The second difference in the **cublas_v2** module is the v2 entry points, such as **cublasIsamax_v2** do not implicitly handle the pointer modes for the user. It is up to the programmer to make calls to **cublasSetPointerMode** to tell the library if scalar arguments reside on the host or device. The actual interfaces to the v2 entry points do not change, and are not listed in this section.

2.6.1. Single Precision Functions and Subroutines

This section contains the V2 interfaces to the single precision BLAS and cuBLAS functions and subroutines.

2.6.1.1. isamax

If you use the cublas_v2 module, the interface for cublasIsamax is changed to the following:

```
integer(4) function cublasIsamax(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  real(4), device, dimension(*) :: x
  integer :: incx
  integer, device :: res ! device or host variable
```

2.6.1.2. isamin

If you use the cublas_v2 module, the interface for cublasIsamin is changed to the following:

```
integer(4) function cublasIsamin(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  real(4), device, dimension(*) :: x
  integer :: incx
  integer, device :: res ! device or host variable
```

2.6.1.3. sasum

If you use the cublas_v2 module, the interface for cublasSasum is changed to the following:

```
integer(4) function cublasSasum(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  real(4), device, dimension(*) :: x
  integer :: incx
  real(4), device :: res ! device or host variable
```

2.6.1.4. saxpy

If you use the cublas_v2 module, the interface for cublasSaxpy is changed to the following:

```
integer(4) function cublasSaxpy(h, n, a, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  real(4), device :: a ! device or host variable
  real(4), device, dimension(*) :: x, y
```

```
  integer :: incx, incy
```

2.6.1.5. scopy

If you use the cublas_v2 module, the interface for cublasScopy is changed to the following:

```
integer(4) function cublasScopy(h, n, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  real(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

2.6.1.6. sdot

If you use the cublas_v2 module, the interface for cublasSdot is changed to the following:

```
integer(4) function cublasSdot(h, n, x, incx, y, incy, res)
  type(cublasHandle) :: h
  integer :: n
  real(4), device, dimension(*) :: x, y
  integer :: incx, incy
  real(4), device :: res ! device or host variable
```

2.6.1.7. snrm2

If you use the cublas_v2 module, the interface for cublasSnrm2 is changed to the following:

```
integer(4) function cublasSnrm2(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  real(4), device, dimension(*) :: x
  integer :: incx
  real(4), device :: res ! device or host variable
```

2.6.1.8. srot

If you use the cublas_v2 module, the interface for cublasSrot is changed to the following:

```
integer(4) function cublasSrot(h, n, x, incx, y, incy, sc, ss)
  type(cublasHandle) :: h
  integer :: n
  real(4), device :: sc, ss ! device or host variable
  real(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

2.6.1.9. srotg

If you use the cublas_v2 module, the interface for cublasSrotg is changed to the following:

```
integer(4) function cublasSrotg(h, sa, sb, sc, ss)
  type(cublasHandle) :: h
  real(4), device :: sa, sb, sc, ss ! device or host variable
```

2.6.1.10. srotm

If you use the cublas_v2 module, the interface for cublasSrotm is changed to the following:

```
integer(4) function cublasSrotm(h, n, x, incx, y, incy, param)
```

```
  type(cublasHandle) :: h
  integer :: n
  real(4), device :: param(*) ! device or host variable
  real(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

2.6.1.11. srotmg

If you use the cublas_v2 module, the interface for cublasSrotmg is changed to the following:

```
integer(4) function cublasSrotmg(h, d1, d2, x1, y1, param)
  type(cublasHandle) :: h
  real(4), device :: d1, d2, x1, y1, param(*) ! device or host variable
```

2.6.1.12. sscal

If you use the cublas_v2 module, the interface for cublasSscal is changed to the following:

```
integer(4) function cublasSscal(h, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: n
  real(4), device :: a ! device or host variable
  real(4), device, dimension(*) :: x
  integer :: incx
```

2.6.1.13. sswap

If you use the cublas_v2 module, the interface for cublasSswap is changed to the following:

```
integer(4) function cublasSswap(h, n, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  real(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

2.6.1.14. sgbmv

If you use the cublas_v2 module, the interface for cublasSgbmv is changed to the following:

```
integer(4) function cublasSgbmv(h, t, m, n, kl, ku, alpha, a, lda, x, incx,
beta, y, incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: m, n, kl, ku, lda, incx, incy
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x, y
  real(4), device :: alpha, beta ! device or host variable
```

2.6.1.15. sgemv

If you use the cublas_v2 module, the interface for cublasSgemv is changed to the following:

```
integer(4) function cublasSgemv(h, t, m, n, alpha, a, lda, x, incx, beta, y,
incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: m, n, lda, incx, incy
  real(4), device, dimension(lda, *) :: a
```

```
  real(4), device, dimension(*) :: x, y
  real(4), device :: alpha, beta ! device or host variable
```

2.6.1.16. sger

If you use the cublas_v2 module, the interface for cublasSger is changed to the following:

```
integer(4) function cublasSger(h, m, n, alpha, x, incx, y, incy, a, lda)
  type(cublasHandle) :: h
  integer :: m, n, lda, incx, incy
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x, y
  real(4), device :: alpha ! device or host variable
```

2.6.1.17. ssbmv

If you use the cublas_v2 module, the interface for cublasSsbmv is changed to the following:

```
integer(4) function cublasSsbmv(h, t, n, k, alpha, a, lda, x, incx, beta, y,
incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: k, n, lda, incx, incy
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x, y
  real(4), device :: alpha, beta ! device or host variable
```

2.6.1.18. sspmv

If you use the cublas_v2 module, the interface for cublasSspmv is changed to the following:

```
integer(4) function cublasSspmv(h, t, n, alpha, a, x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy
  real(4), device, dimension(*) :: a, x, y
  real(4), device :: alpha, beta ! device or host variable
```

2.6.1.19. sspr

If you use the cublas_v2 module, the interface for cublasSspr is changed to the following:

```
integer(4) function cublasSspr(h, t, n, alpha, x, incx, a)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx
  real(4), device, dimension(*) :: a, x
  real(4), device :: alpha ! device or host variable
```

2.6.1.20. sspr2

If you use the cublas_v2 module, the interface for cublasSspr2 is changed to the following:

```
integer(4) function cublasSspr2(h, t, n, alpha, x, incx, y, incy, a)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy
  real(4), device, dimension(*) :: a, x, y
  real(4), device :: alpha ! device or host variable
```

2.6.1.21. ssymv

If you use the cublas_v2 module, the interface for cublasSsymv is changed to the following:

```
integer(4) function cublasSsymv(h, uplo, n, alpha, a, lda, x, incx, beta, y,
incy)
  type(cublasHandle) :: h
  integer :: uplo
  integer :: n, lda, incx, incy
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x, y
  real(4), device :: alpha, beta ! device or host variable
```

2.6.1.22. ssyr

If you use the cublas_v2 module, the interface for cublasSsyr is changed to the following:

```
integer(4) function cublasSsyr(h, t, n, alpha, x, incx, a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, lda
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x
  real(4), device :: alpha ! device or host variable
```

2.6.1.23. ssyr2

If you use the cublas_v2 module, the interface for cublasSsyr2 is changed to the following:

```
integer(4) function cublasSsyr2(h, t, n, alpha, x, incx, y, incy, a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy, lda
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x, y
  real(4), device :: alpha ! device or host variable
```

2.6.1.24. stbmv

If you use the cublas_v2 module, the interface for cublasStbmv is changed to the following:

```
integer(4) function cublasStbmv(h, u, t, d, n, k, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, k, incx, lda
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x
```

2.6.1.25. stbsv

If you use the cublas_v2 module, the interface for cublasStbsv is changed to the following:

```
integer(4) function cublasStbsv(h, u, t, d, n, k, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, k, incx, lda
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x
```

2.6.1.26. stpmv

If you use the cublas_v2 module, the interface for cublasStpmv is changed to the following:

```
integer(4) function cublasStpmv(h, u, t, d, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx
  real(4), device, dimension(*) :: a, x
```

2.6.1.27. stpsv

If you use the cublas_v2 module, the interface for cublasStpsv is changed to the following:

```
integer(4) function cublasStpsv(h, u, t, d, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx
  real(4), device, dimension(*) :: a, x
```

2.6.1.28. strmv

If you use the cublas_v2 module, the interface for cublasStrmv is changed to the following:

```
integer(4) function cublasStrmv(h, u, t, d, n, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx, lda
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x
```

2.6.1.29. strsv

If you use the cublas_v2 module, the interface for cublasStrsv is changed to the following:

```
integer(4) function cublasStrsv(h, u, t, d, n, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx, lda
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x
```

2.6.1.30. sgemm

If you use the cublas_v2 module, the interface for cublasSgemm is changed to the following:

```
integer(4) function cublasSgemm(h, transa, transb, m, n, k, alpha, a, lda, b,
ldb, beta, c, ldc)
  type(cublasHandle) :: h
  integer :: transa, transb
  integer :: m, n, k, lda, ldb, ldc
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(ldb, *) :: b
  real(4), device, dimension(ldc, *) :: c
  real(4), device :: alpha, beta ! device or host variable
```

2.6.1.31. ssymm

If you use the cublas_v2 module, the interface for cublasSsymm is changed to the following:

```
integer(4) function cublasSsymm(h, side, uplo, m, n, alpha, a, lda, b, ldb,
beta, c, ldc)
  type(cublasHandle) :: h
  integer :: side, uplo
  integer :: m, n, lda, ldb, ldc
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(ldb, *) :: b
  real(4), device, dimension(ldc, *) :: c
  real(4), device :: alpha, beta ! device or host variable
```

2.6.1.32. ssyrk

If you use the cublas_v2 module, the interface for cublasSsyrk is changed to the following:

```
integer(4) function cublasSsyrk(h, uplo, trans, n, k, alpha, a, lda, beta, c,
ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldc
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(ldc, *) :: c
  real(4), device :: alpha, beta ! device or host variable
```

2.6.1.33. ssyr2k

If you use the cublas_v2 module, the interface for cublasSsyr2k is changed to the following:

```
integer(4) function cublasSsyr2k(h, uplo, trans, n, k, alpha, a, lda, b, ldb,
beta, c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(ldb, *) :: b
  real(4), device, dimension(ldc, *) :: c
  real(4), device :: alpha, beta ! device or host variable
```

2.6.1.34. ssyrkx

If you use the cublas_v2 module, the interface for cublasSsyrkx is changed to the following:

```
integer(4) function cublasSsyrkx(h, uplo, trans, n, k, alpha, a, lda, b, ldb,
beta, c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(ldb, *) :: b
  real(4), device, dimension(ldc, *) :: c
  real(4), device :: alpha, beta ! device or host variable
```

2.6.1.35. strmm

If you use the cublas_v2 module, the interface for cublasStrmm is changed to the following:

```
integer(4) function cublasStrmm(h, side, uplo, transa, diag, m, n, alpha, a,
lda, b, ldb, c, ldc)
  type(cublasHandle) :: h
  integer :: side, uplo, transa, diag
  integer :: m, n, lda, ldb, ldc
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(ldb, *) :: b
  real(4), device, dimension(ldc, *) :: c
  real(4), device :: alpha ! device or host variable
```

2.6.1.36. strsm

If you use the cublas_v2 module, the interface for cublasStrsm is changed to the following:

```
integer(4) function cublasStrsm(h, side, uplo, transa, diag, m, n, alpha, a,
lda, b, ldb)
  type(cublasHandle) :: h
  integer :: side, uplo, transa, diag
  integer :: m, n, lda, ldb
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(ldb, *) :: b
  real(4), device :: alpha ! device or host variable
```

2.6.2. Double Precision Functions and Subroutines

This section contains the V2 interfaces to the double precision BLAS and cuBLAS functions and subroutines.

2.6.2.1. idamax

If you use the cublas_v2 module, the interface for cublasIdamax is changed to the following:

```
integer(4) function cublasIdamax(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  real(8), device, dimension(*) :: x
  integer :: incx
  integer, device :: res ! device or host variable
```

2.6.2.2. idamin

If you use the cublas_v2 module, the interface for cublasIdamin is changed to the following:

```
integer(4) function cublasIdamin(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  real(8), device, dimension(*) :: x
  integer :: incx
  integer, device :: res ! device or host variable
```

2.6.2.3. dasum

If you use the cublas_v2 module, the interface for cublasDasum is changed to the following:

```
integer(4) function cublasDasum(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  real(8), device, dimension(*) :: x
  integer :: incx
  real(8), device :: res ! device or host variable
```

2.6.2.4. daxpy

If you use the cublas_v2 module, the interface for cublasDaxpy is changed to the following:

```
integer(4) function cublasDaxpy(h, n, a, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  real(8), device :: a ! device or host variable
  real(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

2.6.2.5. dcopy

If you use the cublas_v2 module, the interface for cublasDcopy is changed to the following:

```
integer(4) function cublasDcopy(h, n, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  real(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

2.6.2.6. ddot

If you use the cublas_v2 module, the interface for cublasDdot is changed to the following:

```
integer(4) function cublasDdot(h, n, x, incx, y, incy, res)
  type(cublasHandle) :: h
  integer :: n
  real(8), device, dimension(*) :: x, y
  integer :: incx, incy
  real(8), device :: res ! device or host variable
```

2.6.2.7. dnrm2

If you use the cublas_v2 module, the interface for cublasDnrm2 is changed to the following:

```
integer(4) function cublasDnrm2(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  real(8), device, dimension(*) :: x
  integer :: incx
  real(8), device :: res ! device or host variable
```

2.6.2.8. drot

If you use the cublas_v2 module, the interface for cublasDrot is changed to the following:

```
integer(4) function cublasDrot(h, n, x, incx, y, incy, sc, ss)
  type(cublasHandle) :: h
  integer :: n
  real(8), device :: sc, ss ! device or host variable
  real(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

2.6.2.9. drotg

If you use the cublas_v2 module, the interface for cublasDrotg is changed to the following:

```
integer(4) function cublasDrotg(h, sa, sb, sc, ss)
  type(cublasHandle) :: h
  real(8), device :: sa, sb, sc, ss ! device or host variable
```

2.6.2.10. drotm

If you use the cublas_v2 module, the interface for cublasDrotm is changed to the following:

```
integer(4) function cublasDrotm(h, n, x, incx, y, incy, param)
  type(cublasHandle) :: h
  integer :: n
  real(8), device :: param(*) ! device or host variable
  real(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

2.6.2.11. drotmg

If you use the cublas_v2 module, the interface for cublasDrotmg is changed to the following:

```
integer(4) function cublasDrotmg(h, d1, d2, x1, y1, param)
  type(cublasHandle) :: h
  real(8), device :: d1, d2, x1, y1, param(*) ! device or host variable
```

2.6.2.12. dscal

If you use the cublas_v2 module, the interface for cublasDscal is changed to the following:

```
integer(4) function cublasDscal(h, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: n
  real(8), device :: a ! device or host variable
  real(8), device, dimension(*) :: x
  integer :: incx
```

2.6.2.13. dswap

If you use the cublas_v2 module, the interface for cublasDswap is changed to the following:

```
integer(4) function cublasDswap(h, n, x, incx, y, incy)
  type(cublasHandle) :: h
```

```
  integer :: n
  real(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

2.6.2.14. dgbmv

If you use the cublas_v2 module, the interface for cublasDgbmv is changed to the following:

```
integer(4) function cublasDgbmv(h, t, m, n, kl, ku, alpha, a, lda, x, incx,
beta, y, incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: m, n, kl, ku, lda, incx, incy
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x, y
  real(8), device :: alpha, beta ! device or host variable
```

2.6.2.15. dgemv

If you use the cublas_v2 module, the interface for cublasDgemv is changed to the following:

```
integer(4) function cublasDgemv(h, t, m, n, alpha, a, lda, x, incx, beta, y,
incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: m, n, lda, incx, incy
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x, y
  real(8), device :: alpha, beta ! device or host variable
```

2.6.2.16. dger

If you use the cublas_v2 module, the interface for cublasDger is changed to the following:

```
integer(4) function cublasDger(h, m, n, alpha, x, incx, y, incy, a, lda)
  type(cublasHandle) :: h
  integer :: m, n, lda, incx, incy
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x, y
  real(8), device :: alpha ! device or host variable
```

2.6.2.17. dsbmv

If you use the cublas_v2 module, the interface for cublasDsbmv is changed to the following:

```
integer(4) function cublasDsbmv(h, t, n, k, alpha, a, lda, x, incx, beta, y,
incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: k, n, lda, incx, incy
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x, y
  real(8), device :: alpha, beta ! device or host variable
```

2.6.2.18. dspmv

If you use the cublas_v2 module, the interface for cublasDspmv is changed to the following:

```
integer(4) function cublasDspmv(h, t, n, alpha, a, x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy
  real(8), device, dimension(*) :: a, x, y
  real(8), device :: alpha, beta ! device or host variable
```

2.6.2.19. dspr

If you use the cublas_v2 module, the interface for cublasDspr is changed to the following:

```
integer(4) function cublasDspr(h, t, n, alpha, x, incx, a)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx
  real(8), device, dimension(*) :: a, x
  real(8), device :: alpha ! device or host variable
```

2.6.2.20. dspr2

If you use the cublas_v2 module, the interface for cublasDspr2 is changed to the following:

```
integer(4) function cublasDspr2(h, t, n, alpha, x, incx, y, incy, a)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy
  real(8), device, dimension(*) :: a, x, y
  real(8), device :: alpha ! device or host variable
```

2.6.2.21. dsymv

If you use the cublas_v2 module, the interface for cublasDsymv is changed to the following:

```
integer(4) function cublasDsymv(h, uplo, n, alpha, a, lda, x, incx, beta, y,
incy)
  type(cublasHandle) :: h
  integer :: uplo
  integer :: n, lda, incx, incy
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x, y
  real(8), device :: alpha, beta ! device or host variable
```

2.6.2.22. dsyr

If you use the cublas_v2 module, the interface for cublasDsyr is changed to the following:

```
integer(4) function cublasDsyr(h, t, n, alpha, x, incx, a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, lda
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x
  real(8), device :: alpha ! device or host variable
```

2.6.2.23. dsyr2

If you use the cublas_v2 module, the interface for cublasDsyr2 is changed to the following:

```
integer(4) function cublasDsyr2(h, t, n, alpha, x, incx, y, incy, a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy, lda
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x, y
  real(8), device :: alpha ! device or host variable
```

2.6.2.24. dtbmv

If you use the cublas_v2 module, the interface for cublasDtbmv is changed to the following:

```
integer(4) function cublasDtbmv(h, u, t, d, n, k, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, k, incx, lda
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x
```

2.6.2.25. dtbsv

If you use the cublas_v2 module, the interface for cublasDtbsv is changed to the following:

```
integer(4) function cublasDtbsv(h, u, t, d, n, k, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, k, incx, lda
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x
```

2.6.2.26. dtpmv

If you use the cublas_v2 module, the interface for cublasDtpmv is changed to the following:

```
integer(4) function cublasDtpmv(h, u, t, d, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx
  real(8), device, dimension(*) :: a, x
```

2.6.2.27. dtpsv

If you use the cublas_v2 module, the interface for cublasDtpsv is changed to the following:

```
integer(4) function cublasDtpsv(h, u, t, d, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx
  real(8), device, dimension(*) :: a, x
```

2.6.2.28. dtrmv

If you use the cublas_v2 module, the interface for cublasDtrmv is changed to the following:

```
integer(4) function cublasDtrmv(h, u, t, d, n, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx, lda
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x
```

2.6.2.29. dtrsv

If you use the cublas_v2 module, the interface for cublasDtrsv is changed to the following:

```
integer(4) function cublasDtrsv(h, u, t, d, n, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx, lda
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x
```

2.6.2.30. dgemm

If you use the cublas_v2 module, the interface for cublasDgemm is changed to the following:

```
integer(4) function cublasDgemm(h, transa, transb, m, n, k, alpha, a, lda, b,
ldb, beta, c, ldc)
  type(cublasHandle) :: h
  integer :: transa, transb
  integer :: m, n, k, lda, ldb, ldc
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(ldb, *) :: b
  real(8), device, dimension(ldc, *) :: c
  real(8), device :: alpha, beta ! device or host variable
```

2.6.2.31. dsymm

If you use the cublas_v2 module, the interface for cublasDsymm is changed to the following:

```
integer(4) function cublasDsymm(h, side, uplo, m, n, alpha, a, lda, b, ldb,
beta, c, ldc)
  type(cublasHandle) :: h
  integer :: side, uplo
  integer :: m, n, lda, ldb, ldc
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(ldb, *) :: b
  real(8), device, dimension(ldc, *) :: c
  real(8), device :: alpha, beta ! device or host variable
```

2.6.2.32. dsyrk

If you use the cublas_v2 module, the interface for cublasDsyrk is changed to the following:

```
integer(4) function cublasDsyrk(h, uplo, trans, n, k, alpha, a, lda, beta, c,
ldc)
  type(cublasHandle) :: h
```

```
  integer :: uplo, trans
  integer :: n, k, lda, ldc
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(ldc, *) :: c
  real(8), device :: alpha, beta ! device or host variable
```

2.6.2.33. dsyr2k

If you use the cublas_v2 module, the interface for cublasDsyr2k is changed to the following:

```
integer(4) function cublasDsyr2k(h, uplo, trans, n, k, alpha, a, lda, b, ldb,
beta, c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(ldb, *) :: b
  real(8), device, dimension(ldc, *) :: c
  real(8), device :: alpha, beta ! device or host variable
```

2.6.2.34. dsyrkx

If you use the cublas_v2 module, the interface for cublasDsyrkx is changed to the following:

```
integer(4) function cublasDsyrkx(h, uplo, trans, n, k, alpha, a, lda, b, ldb,
beta, c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(ldb, *) :: b
  real(8), device, dimension(ldc, *) :: c
  real(8), device :: alpha, beta ! device or host variable
```

2.6.2.35. dtrmm

If you use the cublas_v2 module, the interface for cublasDtrmm is changed to the following:

```
integer(4) function cublasDtrmm(h, side, uplo, transa, diag, m, n, alpha, a,
lda, b, ldb, c, ldc)
  type(cublasHandle) :: h
  integer :: side, uplo, transa, diag
  integer :: m, n, lda, ldb, ldc
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(ldb, *) :: b
  real(8), device, dimension(ldc, *) :: c
  real(8), device :: alpha ! device or host variable
```

2.6.2.36. dtrsm

If you use the cublas_v2 module, the interface for cublasDtrsm is changed to the following:

```
integer(4) function cublasDtrsm(h, side, uplo, transa, diag, m, n, alpha, a,
lda, b, ldb)
  type(cublasHandle) :: h
  integer :: side, uplo, transa, diag
  integer :: m, n, lda, ldb
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(ldb, *) :: b
  real(8), device :: alpha ! device or host variable
```

2.6.3. Single Precision Complex Functions and Subroutines

This section contains the V2 interfaces to the single precision complex BLAS and cuBLAS functions and subroutines.

2.6.3.1. icamax

If you use the cublas_v2 module, the interface for cublasIcamax is changed to the following:

```
integer(4) function cublasIcamax(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  complex(4), device, dimension(*) :: x
  integer :: incx
  integer, device :: res ! device or host variable
```

2.6.3.2. icamin

If you use the cublas_v2 module, the interface for cublasIcamin is changed to the following:

```
integer(4) function cublasIcamin(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  complex(4), device, dimension(*) :: x
  integer :: incx
  integer, device :: res ! device or host variable
```

2.6.3.3. scasum

If you use the cublas_v2 module, the interface for cublasScasum is changed to the following:

```
integer(4) function cublasScasum(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  complex(4), device, dimension(*) :: x
  integer :: incx
  real(4), device :: res ! device or host variable
```

2.6.3.4. caxpy

If you use the cublas_v2 module, the interface for cublasCaxpy is changed to the following:

```
integer(4) function cublasCaxpy(h, n, a, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  complex(4), device :: a ! device or host variable
  complex(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

2.6.3.5. ccopy

If you use the cublas_v2 module, the interface for cublasCcopy is changed to the following:

```
integer(4) function cublasCcopy(h, n, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  complex(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

2.6.3.6. cdotc

If you use the cublas_v2 module, the interface for cublasCdotc is changed to the following:

```
integer(4) function cublasCdotc(h, n, x, incx, y, incy, res)
  type(cublasHandle) :: h
  integer :: n
  complex(4), device, dimension(*) :: x, y
  integer :: incx, incy
  complex(4), device :: res ! device or host variable
```

2.6.3.7. cdotu

If you use the cublas_v2 module, the interface for cublasCdotu is changed to the following:

```
integer(4) function cublasCdotu(h, n, x, incx, y, incy, res)
  type(cublasHandle) :: h
  integer :: n
  complex(4), device, dimension(*) :: x, y
  integer :: incx, incy
  complex(4), device :: res ! device or host variable
```

2.6.3.8. scnrm2

If you use the cublas_v2 module, the interface for cublasScnrm2 is changed to the following:

```
integer(4) function cublasScnrm2(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  complex(4), device, dimension(*) :: x
  integer :: incx
  real(4), device :: res ! device or host variable
```

2.6.3.9. crot

If you use the cublas_v2 module, the interface for cublasCrot is changed to the following:

```
integer(4) function cublasCrot(h, n, x, incx, y, incy, sc, ss)
  type(cublasHandle) :: h
  integer :: n
  real(4), device :: sc ! device or host variable
  complex(4), device :: ss ! device or host variable
  complex(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

2.6.3.10. csrot

If you use the cublas_v2 module, the interface for cublasCsrot is changed to the following:

```
integer(4) function cublasCsrot(h, n, x, incx, y, incy, sc, ss)
  type(cublasHandle) :: h
  integer :: n
  real(4), device :: sc, ss ! device or host variable
  complex(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

2.6.3.11. crotg

If you use the cublas_v2 module, the interface for cublasCrotg is changed to the following:

```
integer(4) function cublasCrotg(h, sa, sb, sc, ss)
  type(cublasHandle) :: h
  complex(4), device :: sa, sb, ss ! device or host variable
  real(4), device :: sc ! device or host variable
```

2.6.3.12. cscal

If you use the cublas_v2 module, the interface for cublasCscal is changed to the following:

```
integer(4) function cublasCscal(h, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: n
  complex(4), device :: a ! device or host variable
  complex(4), device, dimension(*) :: x
  integer :: incx
```

2.6.3.13. csscal

If you use the cublas_v2 module, the interface for cublasCsscal is changed to the following:

```
integer(4) function cublasCsscal(h, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: n
  real(4), device :: a ! device or host variable
  complex(4), device, dimension(*) :: x
  integer :: incx
```

2.6.3.14. cswap

If you use the cublas_v2 module, the interface for cublasCswap is changed to the following:

```
integer(4) function cublasCswap(h, n, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  complex(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

2.6.3.15. cgbmv

If you use the cublas_v2 module, the interface for cublasCgbmv is changed to the following:

```
integer(4) function cublasCgbmv(h, t, m, n, kl, ku, alpha, a, lda, x, incx,
beta, y, incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: m, n, kl, ku, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x, y
  complex(4), device :: alpha, beta ! device or host variable
```

2.6.3.16. cgemv

If you use the cublas_v2 module, the interface for cublasCgemv is changed to the following:

```
integer(4) function cublasCgemv(h, t, m, n, alpha, a, lda, x, incx, beta, y,
incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: m, n, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x, y
  complex(4), device :: alpha, beta ! device or host variable
```

2.6.3.17. cgerc

If you use the cublas_v2 module, the interface for cublasCgerc is changed to the following:

```
integer(4) function cublasCgerc(h, m, n, alpha, x, incx, y, incy, a, lda)
  type(cublasHandle) :: h
  integer :: m, n, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x, y
  complex(4), device :: alpha ! device or host variable
```

2.6.3.18. cgeru

If you use the cublas_v2 module, the interface for cublasCgeru is changed to the following:

```
integer(4) function cublasCgeru(h, m, n, alpha, x, incx, y, incy, a, lda)
  type(cublasHandle) :: h
  integer :: m, n, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x, y
  complex(4), device :: alpha ! device or host variable
```

2.6.3.19. csymv

If you use the cublas_v2 module, the interface for cublasCsymv is changed to the following:

```
integer(4) function cublasCsymv(h, uplo, n, alpha, a, lda, x, incx, beta, y,
incy)
  type(cublasHandle) :: h
  integer :: uplo
  integer :: n, lda, incx, incy
```

```
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x, y
  complex(4), device :: alpha, beta ! device or host variable
```

2.6.3.20. csyr

If you use the cublas_v2 module, the interface for cublasCsyr is changed to the following:

```
integer(4) function cublasCsyr(h, t, n, alpha, x, incx, a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, lda
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x
  complex(4), device :: alpha ! device or host variable
```

2.6.3.21. csyr2

If you use the cublas_v2 module, the interface for cublasCsyr2 is changed to the following:

```
integer(4) function cublasCsyr2(h, t, n, alpha, x, incx, y, incy, a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy, lda
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x, y
  complex(4), device :: alpha ! device or host variable
```

2.6.3.22. ctbmv

If you use the cublas_v2 module, the interface for cublasCtbmv is changed to the following:

```
integer(4) function cublasCtbmv(h, u, t, d, n, k, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, k, incx, lda
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x
```

2.6.3.23. ctbsv

If you use the cublas_v2 module, the interface for cublasCtbsv is changed to the following:

```
integer(4) function cublasCtbsv(h, u, t, d, n, k, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, k, incx, lda
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x
```

2.6.3.24. ctpmv

If you use the cublas_v2 module, the interface for cublasCtpmv is changed to the following:

```
integer(4) function cublasCtpmv(h, u, t, d, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx
```

```
  complex(4), device, dimension(*) :: a, x
```

2.6.3.25. ctpsv

If you use the cublas_v2 module, the interface for cublasCtpsv is changed to the following:

```
integer(4) function cublasCtpsv(h, u, t, d, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx
  complex(4), device, dimension(*) :: a, x
```

2.6.3.26. ctrmv

If you use the cublas_v2 module, the interface for cublasCtrmv is changed to the following:

```
integer(4) function cublasCtrmv(h, u, t, d, n, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx, lda
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x
```

2.6.3.27. ctrsv

If you use the cublas_v2 module, the interface for cublasCtrsv is changed to the following:

```
integer(4) function cublasCtrsv(h, u, t, d, n, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx, lda
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x
```

2.6.3.28. chbmv

If you use the cublas_v2 module, the interface for cublasChbmv is changed to the following:

```
integer(4) function cublasChbmv(h, uplo, n, k, alpha, a, lda, x, incx, beta, y,
incy)
  type(cublasHandle) :: h
  integer :: uplo
  integer :: k, n, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x, y
  complex(4), device :: alpha, beta ! device or host variable
```

2.6.3.29. chemv

If you use the cublas_v2 module, the interface for cublasChemv is changed to the following:

```
integer(4) function cublasChemv(h, uplo, n, alpha, a, lda, x, incx, beta, y,
incy)
  type(cublasHandle) :: h
  integer :: uplo
  integer :: n, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x, y
```

```
  complex(4), device :: alpha, beta ! device or host variable
```

2.6.3.30. chpmv

If you use the cublas_v2 module, the interface for cublasChpmv is changed to the following:

```
integer(4) function cublasChpmv(h, uplo, n, alpha, a, x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: uplo
  integer :: n, incx, incy
  complex(4), device, dimension(*) :: a, x, y
  complex(4), device :: alpha, beta ! device or host variable
```

2.6.3.31. cher

If you use the cublas_v2 module, the interface for cublasCher is changed to the following:

```
integer(4) function cublasCher(h, t, n, alpha, x, incx, a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, lda
  complex(4), device, dimension(*) :: a, x
  real(4), device :: alpha ! device or host variable
```

2.6.3.32. cher2

If you use the cublas_v2 module, the interface for cublasCher2 is changed to the following:

```
integer(4) function cublasCher2(h, t, n, alpha, x, incx, y, incy, a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy, lda
  complex(4), device, dimension(*) :: a, x, y
  complex(4), device :: alpha ! device or host variable
```

2.6.3.33. chpr

If you use the cublas_v2 module, the interface for cublasChpr is changed to the following:

```
integer(4) function cublasChpr(h, t, n, alpha, x, incx, a)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx
  complex(4), device, dimension(*) :: a, x
  real(4), device :: alpha ! device or host variable
```

2.6.3.34. chpr2

If you use the cublas_v2 module, the interface for cublasChpr2 is changed to the following:

```
integer(4) function cublasChpr2(h, t, n, alpha, x, incx, y, incy, a)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy
  complex(4), device, dimension(*) :: a, x, y
  complex(4), device :: alpha ! device or host variable
```

2.6.3.35. cgemm

If you use the cublas_v2 module, the interface for cublasCgemm is changed to the following:

```
integer(4) function cublasCgemm(h, transa, transb, m, n, k, alpha, a, lda, b,
ldb, beta, c, ldc)
  type(cublasHandle) :: h
  integer :: transa, transb
  integer :: m, n, k, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device, dimension(ldc, *) :: c
  complex(4), device :: alpha, beta ! device or host variable
```

2.6.3.36. csymm

If you use the cublas_v2 module, the interface for cublasCsymm is changed to the following:

```
integer(4) function cublasCsymm(h, side, uplo, m, n, alpha, a, lda, b, ldb,
beta, c, ldc)
  type(cublasHandle) :: h
  integer :: side, uplo
  integer :: m, n, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device, dimension(ldc, *) :: c
  complex(4), device :: alpha, beta ! device or host variable
```

2.6.3.37. csyrk

If you use the cublas_v2 module, the interface for cublasCsyrk is changed to the following:

```
integer(4) function cublasCsyrk(h, uplo, trans, n, k, alpha, a, lda, beta, c,
ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldc, *) :: c
  complex(4), device :: alpha, beta ! device or host variable
```

2.6.3.38. csyr2k

If you use the cublas_v2 module, the interface for cublasCsyr2k is changed to the following:

```
integer(4) function cublasCsyr2k(h, uplo, trans, n, k, alpha, a, lda, b, ldb,
beta, c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device, dimension(ldc, *) :: c
  complex(4), device :: alpha, beta ! device or host variable
```

2.6.3.39. csyrkx

If you use the cublas_v2 module, the interface for cublasCsyrkx is changed to the following:

```
integer(4) function cublasCsyrkx(h, uplo, trans, n, k, alpha, a, lda, b, ldb,
beta, c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device, dimension(ldc, *) :: c
  complex(4), device :: alpha, beta ! device or host variable
```

2.6.3.40. ctrmm

If you use the cublas_v2 module, the interface for cublasCtrmm is changed to the following:

```
integer(4) function cublasCtrmm(h, side, uplo, transa, diag, m, n, alpha, a,
lda, b, ldb, c, ldc)
  type(cublasHandle) :: h
  integer :: side, uplo, transa, diag
  integer :: m, n, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device, dimension(ldc, *) :: c
  complex(4), device :: alpha ! device or host variable
```

2.6.3.41. ctrsm

If you use the cublas_v2 module, the interface for cublasCtrsm is changed to the following:

```
integer(4) function cublasCtrsm(h, side, uplo, transa, diag, m, n, alpha, a,
lda, b, ldb)
  type(cublasHandle) :: h
  integer :: side, uplo, transa, diag
  integer :: m, n, lda, ldb
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device :: alpha ! device or host variable
```

2.6.3.42. chemm

If you use the cublas_v2 module, the interface for cublasChemm is changed to the following:

```
integer(4) function cublasChemm(h, side, uplo, m, n, alpha, a, lda, b, ldb,
beta, c, ldc)
  type(cublasHandle) :: h
  integer :: side, uplo
  integer :: m, n, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device, dimension(ldc, *) :: c
  complex(4), device :: alpha, beta ! device or host variable
```

2.6.3.43. cherk

If you use the cublas_v2 module, the interface for cublasCherk is changed to the following:

```
integer(4) function cublasCherk(h, uplo, trans, n, k, alpha, a, lda, beta, c,
ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldc, *) :: c
  real(4), device :: alpha, beta ! device or host variable
```

2.6.3.44. cher2k

If you use the cublas_v2 module, the interface for cublasCher2k is changed to the following:

```
integer(4) function cublasCher2k(h, uplo, trans, n, k, alpha, a, lda, b, ldb,
beta, c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device, dimension(ldc, *) :: c
  complex(4), device :: alpha ! device or host variable
  real(4), device :: beta ! device or host variable
```

2.6.3.45. cherkx

If you use the cublas_v2 module, the interface for cublasCherkx is changed to the following:

```
integer(4) function cublasCherkx(h, uplo, trans, n, k, alpha, a, lda, b, ldb,
beta, c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device, dimension(ldc, *) :: c
  complex(4), device :: alpha ! device or host variable
  real(4), device :: beta ! device or host variable
```

2.6.4. Double Precision Complex Functions and Subroutines

This section contains the V2 interfaces to the double precision complex BLAS and cuBLAS functions and subroutines.

2.6.4.1. izamax

If you use the cublas_v2 module, the interface for cublasIzamax is changed to the following:

```
integer(4) function cublasIzamax(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
```

```
  complex(8), device, dimension(*) :: x
  integer :: incx
  integer, device :: res ! device or host variable
```

2.6.4.2. izamin

If you use the cublas_v2 module, the interface for cublasIzamin is changed to the following:

```
integer(4) function cublasIzamin(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  complex(8), device, dimension(*) :: x
  integer :: incx
  integer, device :: res ! device or host variable
```

2.6.4.3. dzasum

If you use the cublas_v2 module, the interface for cublasDzasum is changed to the following:

```
integer(4) function cublasDzasum(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  complex(8), device, dimension(*) :: x
  integer :: incx
  real(8), device :: res ! device or host variable
```

2.6.4.4. zaxpy

If you use the cublas_v2 module, the interface for cublasZaxpy is changed to the following:

```
integer(4) function cublasZaxpy(h, n, a, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  complex(8), device :: a ! device or host variable
  complex(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

2.6.4.5. zcopy

If you use the cublas_v2 module, the interface for cublasZcopy is changed to the following:

```
integer(4) function cublasZcopy(h, n, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  complex(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

2.6.4.6. zdotc

If you use the cublas_v2 module, the interface for cublasZdotc is changed to the following:

```
integer(4) function cublasZdotc(h, n, x, incx, y, incy, res)
  type(cublasHandle) :: h
  integer :: n
  complex(8), device, dimension(*) :: x, y
  integer :: incx, incy
  complex(8), device :: res ! device or host variable
```

2.6.4.7. zdotu

If you use the cublas_v2 module, the interface for cublasZdotu is changed to the following:

```
integer(4) function cublasZdotu(h, n, x, incx, y, incy, res)
  type(cublasHandle) :: h
  integer :: n
  complex(8), device, dimension(*) :: x, y
  integer :: incx, incy
  complex(8), device :: res ! device or host variable
```

2.6.4.8. dznrm2

If you use the cublas_v2 module, the interface for cublasDznrm2 is changed to the following:

```
integer(4) function cublasDznrm2(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  complex(8), device, dimension(*) :: x
  integer :: incx
  real(8), device :: res ! device or host variable
```

2.6.4.9. zrot

If you use the cublas_v2 module, the interface for cublasZrot is changed to the following:

```
integer(4) function cublasZrot(h, n, x, incx, y, incy, sc, ss)
  type(cublasHandle) :: h
  integer :: n
  real(8), device :: sc ! device or host variable
  complex(8), device :: ss ! device or host variable
  complex(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

2.6.4.10. zsrot

If you use the cublas_v2 module, the interface for cublasZsrot is changed to the following:

```
integer(4) function cublasZsrot(h, n, x, incx, y, incy, sc, ss)
  type(cublasHandle) :: h
  integer :: n
  real(8), device :: sc, ss ! device or host variable
  complex(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

2.6.4.11. zrotg

If you use the cublas_v2 module, the interface for cublasZrotg is changed to the following:

```
integer(4) function cublasZrotg(h, sa, sb, sc, ss)
  type(cublasHandle) :: h
  complex(8), device :: sa, sb, ss ! device or host variable
  real(8), device :: sc ! device or host variable
```

2.6.4.12. zscal

If you use the cublas_v2 module, the interface for cublasZscal is changed to the following:

```
integer(4) function cublasZscal(h, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: n
  complex(8), device :: a ! device or host variable
  complex(8), device, dimension(*) :: x
  integer :: incx
```

2.6.4.13. zdscal

If you use the cublas_v2 module, the interface for cublasZdscal is changed to the following:

```
integer(4) function cublasZdscal(h, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: n
  real(8), device :: a ! device or host variable
  complex(8), device, dimension(*) :: x
  integer :: incx
```

2.6.4.14. zswap

If you use the cublas_v2 module, the interface for cublasZswap is changed to the following:

```
integer(4) function cublasZswap(h, n, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  complex(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

2.6.4.15. zgbmv

If you use the cublas_v2 module, the interface for cublasZgbmv is changed to the following:

```
integer(4) function cublasZgbmv(h, t, m, n, kl, ku, alpha, a, lda, x, incx,
beta, y, incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: m, n, kl, ku, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
  complex(8), device :: alpha, beta ! device or host variable
```

2.6.4.16. zgemv

If you use the cublas_v2 module, the interface for cublasZgemv is changed to the following:

```
integer(4) function cublasZgemv(h, t, m, n, alpha, a, lda, x, incx, beta, y,
incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: m, n, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
  complex(8), device :: alpha, beta ! device or host variable
```

2.6.4.17. zgerc

If you use the cublas_v2 module, the interface for cublasZgerc is changed to the following:

```
integer(4) function cublasZgerc(h, m, n, alpha, x, incx, y, incy, a, lda)
  type(cublasHandle) :: h
  integer :: m, n, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
  complex(8), device :: alpha ! device or host variable
```

2.6.4.18. zgeru

If you use the cublas_v2 module, the interface for cublasZgeru is changed to the following:

```
integer(4) function cublasZgeru(h, m, n, alpha, x, incx, y, incy, a, lda)
  type(cublasHandle) :: h
  integer :: m, n, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
  complex(8), device :: alpha ! device or host variable
```

2.6.4.19. zsymv

If you use the cublas_v2 module, the interface for cublasZsymv is changed to the following:

```
integer(4) function cublasZsymv(h, uplo, n, alpha, a, lda, x, incx, beta, y,
incy)
  type(cublasHandle) :: h
  integer :: uplo
  integer :: n, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
  complex(8), device :: alpha, beta ! device or host variable
```

2.6.4.20. zsyr

If you use the cublas_v2 module, the interface for cublasZsyr is changed to the following:

```
integer(4) function cublasZsyr(h, t, n, alpha, x, incx, a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, lda
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x
  complex(8), device :: alpha ! device or host variable
```

2.6.4.21. zsyr2

If you use the cublas_v2 module, the interface for cublasZsyr2 is changed to the following:

```
integer(4) function cublasZsyr2(h, t, n, alpha, x, incx, y, incy, a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy, lda
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
```

```
  complex(8), device :: alpha ! device or host variable
```

2.6.4.22. ztbmv

If you use the cublas_v2 module, the interface for cublasZtbmv is changed to the following:

```
integer(4) function cublasZtbmv(h, u, t, d, n, k, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, k, incx, lda
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x
```

2.6.4.23. ztbsv

If you use the cublas_v2 module, the interface for cublasZtbsv is changed to the following:

```
integer(4) function cublasZtbsv(h, u, t, d, n, k, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, k, incx, lda
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x
```

2.6.4.24. ztpmv

If you use the cublas_v2 module, the interface for cublasZtpmv is changed to the following:

```
integer(4) function cublasZtpmv(h, u, t, d, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx
  complex(8), device, dimension(*) :: a, x
```

2.6.4.25. ztpsv

If you use the cublas_v2 module, the interface for cublasZtpsv is changed to the following:

```
integer(4) function cublasZtpsv(h, u, t, d, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx
  complex(8), device, dimension(*) :: a, x
```

2.6.4.26. ztrmv

If you use the cublas_v2 module, the interface for cublasZtrmv is changed to the following:

```
integer(4) function cublasZtrmv(h, u, t, d, n, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx, lda
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x
```

2.6.4.27. ztrsv

If you use the cublas_v2 module, the interface for cublasZtrsv is changed to the following:

```
integer(4) function cublasZtrsv(h, u, t, d, n, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx, lda
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x
```

2.6.4.28. zhbmv

If you use the cublas_v2 module, the interface for cublasZhbmv is changed to the following:

```
integer(4) function cublasZhbmv(h, uplo, n, k, alpha, a, lda, x, incx, beta, y,
incy)
  type(cublasHandle) :: h
  integer :: uplo
  integer :: k, n, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
  complex(8), device :: alpha, beta ! device or host variable
```

2.6.4.29. zhemv

If you use the cublas_v2 module, the interface for cublasZhemv is changed to the following:

```
integer(4) function cublasZhemv(h, uplo, n, alpha, a, lda, x, incx, beta, y,
incy)
  type(cublasHandle) :: h
  integer :: uplo
  integer :: n, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
  complex(8), device :: alpha, beta ! device or host variable
```

2.6.4.30. zhpmv

If you use the cublas_v2 module, the interface for cublasZhpmv is changed to the following:

```
integer(4) function cublasZhpmv(h, uplo, n, alpha, a, x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: uplo
  integer :: n, incx, incy
  complex(8), device, dimension(*) :: a, x, y
  complex(8), device :: alpha, beta ! device or host variable
```

2.6.4.31. zher

If you use the cublas_v2 module, the interface for cublasZher is changed to the following:

```
integer(4) function cublasZher(h, t, n, alpha, x, incx, a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, lda
  complex(8), device, dimension(*) :: a, x
```

```
  real(8), device :: alpha ! device or host variable
```

2.6.4.32. zher2

If you use the cublas_v2 module, the interface for cublasZher2 is changed to the following:

```
integer(4) function cublasZher2(h, t, n, alpha, x, incx, y, incy, a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy, lda
  complex(8), device, dimension(*) :: a, x, y
  complex(8), device :: alpha ! device or host variable
```

2.6.4.33. zhpr

If you use the cublas_v2 module, the interface for cublasZhpr is changed to the following:

```
integer(4) function cublasZhpr(h, t, n, alpha, x, incx, a)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx
  complex(8), device, dimension(*) :: a, x
  real(8), device :: alpha ! device or host variable
```

2.6.4.34. zhpr2

If you use the cublas_v2 module, the interface for cublasZhpr2 is changed to the following:

```
integer(4) function cublasZhpr2(h, t, n, alpha, x, incx, y, incy, a)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy
  complex(8), device, dimension(*) :: a, x, y
  complex(8), device :: alpha ! device or host variable
```

2.6.4.35. zgemm

If you use the cublas_v2 module, the interface for cublasZgemm is changed to the following:

```
integer(4) function cublasZgemm(h, transa, transb, m, n, k, alpha, a, lda, b,
ldb, beta, c, ldc)
  type(cublasHandle) :: h
  integer :: transa, transb
  integer :: m, n, k, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device, dimension(ldc, *) :: c
  complex(8), device :: alpha, beta ! device or host variable
```

2.6.4.36. zsymm

If you use the cublas_v2 module, the interface for cublasZsymm is changed to the following:

```
integer(4) function cublasZsymm(h, side, uplo, m, n, alpha, a, lda, b, ldb,
beta, c, ldc)
  type(cublasHandle) :: h
  integer :: side, uplo
  integer :: m, n, lda, ldb, ldc
```

```
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device, dimension(ldc, *) :: c
  complex(8), device :: alpha, beta ! device or host variable
```

2.6.4.37. zsyrk

If you use the cublas_v2 module, the interface for cublasZsyrk is changed to the following:

```
integer(4) function cublasZsyrk(h, uplo, trans, n, k, alpha, a, lda, beta, c,
ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldc, *) :: c
  complex(8), device :: alpha, beta ! device or host variable
```

2.6.4.38. zsyr2k

If you use the cublas_v2 module, the interface for cublasZsyr2k is changed to the following:

```
integer(4) function cublasZsyr2k(h, uplo, trans, n, k, alpha, a, lda, b, ldb,
beta, c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device, dimension(ldc, *) :: c
  complex(8), device :: alpha, beta ! device or host variable
```

2.6.4.39. zsyrkx

If you use the cublas_v2 module, the interface for cublasZsyrkx is changed to the following:

```
integer(4) function cublasZsyrkx(h, uplo, trans, n, k, alpha, a, lda, b, ldb,
beta, c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device, dimension(ldc, *) :: c
  complex(8), device :: alpha, beta ! device or host variable
```

2.6.4.40. ztrmm

If you use the cublas_v2 module, the interface for cublasZtrmm is changed to the following:

```
integer(4) function cublasZtrmm(h, side, uplo, transa, diag, m, n, alpha, a,
lda, b, ldb, c, ldc)
  type(cublasHandle) :: h
  integer :: side, uplo, transa, diag
  integer :: m, n, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device, dimension(ldc, *) :: c
  complex(8), device :: alpha ! device or host variable
```

2.6.4.41. ztrsm

If you use the cublas_v2 module, the interface for cublasZtrsm is changed to the following:

```
integer(4) function cublasZtrsm(h, side, uplo, transa, diag, m, n, alpha, a,
lda, b, ldb)
  type(cublasHandle) :: h
  integer :: side, uplo, transa, diag
  integer :: m, n, lda, ldb
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device :: alpha ! device or host variable
```

2.6.4.42. zhemm

If you use the cublas_v2 module, the interface for cublasZhemm is changed to the following:

```
integer(4) function cublasZhemm(h, side, uplo, m, n, alpha, a, lda, b, ldb,
beta, c, ldc)
  type(cublasHandle) :: h
  integer :: side, uplo
  integer :: m, n, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device, dimension(ldc, *) :: c
  complex(8), device :: alpha, beta ! device or host variable
```

2.6.4.43. zherk

If you use the cublas_v2 module, the interface for cublasZherk is changed to the following:

```
integer(4) function cublasZherk(h, uplo, trans, n, k, alpha, a, lda, beta, c,
ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldc, *) :: c
  real(8), device :: alpha, beta ! device or host variable
```

2.6.4.44. zher2k

If you use the cublas_v2 module, the interface for cublasZher2k is changed to the following:

```
integer(4) function cublasZher2k(h, uplo, trans, n, k, alpha, a, lda, b, ldb,
beta, c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device, dimension(ldc, *) :: c
  complex(8), device :: alpha ! device or host variable
  real(8), device :: beta ! device or host variable
```

2.6.4.45. zherkx

If you use the cublas_v2 module, the interface for cublasZherkx is changed to the following:

```
integer(4) function cublasZherkx(h, uplo, trans, n, k, alpha, a, lda, b, ldb,
beta, c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device, dimension(ldc, *) :: c
  complex(8), device :: alpha ! device or host variable
  real(8), device :: beta ! device or host variable
```

2.7. CUBLAS XT Module Functions

This section contains interfaces to the cuBLAS XT Module Functions. Users can access this module by inserting the line **`use cublasXt`** into the program unit. The cublasXt library is a host-side library, which supports multiple GPUs. Here is an example:

```
subroutine testxt(n)
use cublasXt
complex*16 :: a(n,n), b(n,n), c(n,n), alpha, beta
type(cublasXtHandle) :: h
integer ndevices(1)
a = cmplx(1.0d0,0.0d0)
b = cmplx(2.0d0,0.0d0)
c = cmplx(-1.0d0,0.0d0)
alpha = cmplx(1.0d0,0.0d0)
beta = cmplx(0.0d0,0.0d0)
istat = cublasXtCreate(h)
if (istat .ne. CUBLAS_STATUS_SUCCESS) print *,istat
ndevices(1) = 0
istat = cublasXtDeviceSelect(h, 1, ndevices)
if (istat .ne. CUBLAS_STATUS_SUCCESS) print *,istat
istat = cublasXtZgemm(h, CUBLAS_OP_N, CUBLAS_OP_N, &
                      n, n, n, &
                      alpha, A, n, B, n, beta, C, n)
if (istat .ne. CUBLAS_STATUS_SUCCESS) print *,istat
istat = cublasXtDestroy(h)
if (istat .ne. CUBLAS_STATUS_SUCCESS) print *,istat
if (all(dble(c).eq.2.0d0*n)) then
    print *,"Test PASSED"
else
    print *,"Test FAILED"
endif
end
```

The **`cublasXt`** module contains all the types and definitions from the **`cublas`** module, and these additional types and enumerations:

```
TYPE cublasXtHandle
  TYPE(C_PTR)  :: handle
END TYPE
```

```
! Pinned memory mode
enum, bind(c)
    enumerator :: CUBLASXT_PINNING_DISABLED=0
    enumerator :: CUBLASXT_PINNING_ENABLED=1
```

```
end enum
```

```
! cublasXtOpType
enum, bind(c)
    enumerator :: CUBLASXT_FLOAT=0
    enumerator :: CUBLASXT_DOUBLE=1
    enumerator :: CUBLASXT_COMPLEX=2
    enumerator :: CUBLASXT_DOUBLECOMPLEX=3
end enum
```

```
! cublasXtBlasOp
enum, bind(c)
    enumerator :: CUBLASXT_GEMM=0
    enumerator :: CUBLASXT_SYRK=1
    enumerator :: CUBLASXT_HERK=2
    enumerator :: CUBLASXT_SYMM=3
    enumerator :: CUBLASXT_HEMM=4
    enumerator :: CUBLASXT_TRSM=5
    enumerator :: CUBLASXT_SYR2K=6
    enumerator :: CUBLASXT_HER2K=7
    enumerator :: CUBLASXT_SPMM=8
    enumerator :: CUBLASXT_SYRKX=9
    enumerator :: CUBLASXT_HERKX=10
    enumerator :: CUBLASXT_TRMM=11
    enumerator :: CUBLASXT_ROUTINE_MAX=12
end enum
```

2.7.1. cublasXtCreate

This function initializes the cublasXt API and creates a handle to an opaque structure holding the cublasXT library context. It allocates hardware resources on the host and device and must be called prior to making any other cublasXt API library calls.

```
integer(4) function cublasXtcreate(h)
  type(cublasXtHandle) :: h
```

2.7.2. cublasXtDestroy

This function releases hardware resources used by the cublasXt API context. This function is usually the last call with a particular handle to the cublasXt API.

```
integer(4) function cublasXtdestroy(h)
  type(cublasXtHandle) :: h
```

2.7.3. cublasXtDeviceSelect

This function allows the user to provide the number of GPU devices and their respective Ids that will participate to the subsequent cublasXt API math function calls. This function will create a cuBLAS context for every GPU provided in that list. Currently the device configuration is static and cannot be changed between math function calls. In that regard, this function should be called only once after cublasXtCreate. To be able to run multiple configurations, multiple cublasXt API contexts should be created.

```
integer(4) function cublasXtdeviceselect(h, ndevices, deviceid)
  type(cublasXtHandle) :: h
  integer :: ndevices
  integer, dimension(*) :: deviceid
```

2.7.4. cublasXtSetBlockDim

This function allows the user to set the block dimension used for the tiling of the matrices for the subsequent Math function calls. Matrices are split in square tiles of blockDim x blockDim dimension. This function can be called anytime and will take effect for the following math function calls. The block dimension should be chosen in a way to optimize the math operation and to make sure that the PCI transfers are well overlapped with the computation.

```
integer(4) function cublasXtsetblockdim(h, blockdim)
  type(cublasXtHandle) :: h
  integer :: blockdim
```

2.7.5. cublasXtGetBlockDim

This function allows the user to query the block dimension used for the tiling of the matrices.

```
integer(4) function cublasXtgetblockdim(h, blockdim)
  type(cublasXtHandle) :: h
  integer :: blockdim
```

2.7.6. cublasXtSetCpuRoutine

This function allows the user to provide a CPU implementation of the corresponding BLAS routine. This function can be used with the function cublasXtSetCpuRatio() to define an hybrid computation between the CPU and the GPUs. Currently the hybrid feature is only supported for the xGEMM routines.

```
integer(4) function cublasXtsetcpuroutine(h, blasop, blastype)
  type(cublasXtHandle) :: h
  integer :: blasop, blastype
```

2.7.7. cublasXtSetCpuRatio

This function allows the user to define the percentage of workload that should be done on a CPU in the context of an hybrid computation. This function can be used with the function cublasXtSetCpuRoutine() to define an hybrid computation between the CPU and the GPUs. Currently the hybrid feature is only supported for the xGEMM routines.

```
integer(4) function cublasXtsetcpuratio(h, blasop, blastype, ratio)
  type(cublasXtHandle) :: h
  integer :: blasop, blastype
  real(4) :: ratio
```

2.7.8. cublasXtSetPinningMemMode

This function allows the user to enable or disable the Pinning Memory mode. When enabled, the matrices passed in subsequent cublasXt API calls will be pinned/unpinned using the CUDART routine cudaHostRegister and cudaHostUnregister respectively if the matrices are not already pinned. If a matrix happened to be pinned partially, it will also not be pinned. Pinning the memory improve PCI transfer performace and allows to overlap PCI memory transfer with computation. However pinning/unpinning the memory takes some time which might not be amortized. It is advised that the user

pins the memory on its own using cudaMallocHost or cudaHostRegister and unpins it when the computation sequence is completed. By default, the Pinning Memory mode is disabled.

```
integer(4) function cublasXtsetpinningmemmode(h, mode)
  type(cublasXtHandle) :: h
  integer :: mode
```

2.7.9. cublasXtGetPinningMemMode

This function allows the user to query the Pinning Memory mode. By default, the Pinning Memory mode is disabled.

```
integer(4) function cublasXtgetpinningmemmode(h, mode)
  type(cublasXtHandle) :: h
  integer :: mode
```

2.7.10. cublasXtSgemm

SGEMM performs one of the matrix-matrix operations C := alpha*op(A)*op(B) + beta*C, where op(X) is one of op(X) = X or op(X) = X**T, alpha and beta are scalars, and A, B and C are matrices, with op(A) an m by k matrix, op(B) a k by n matrix and C an m by n matrix.

```
integer(4) function cublasXtsgemm(h, transa, transb, m, n, k, alpha, a, lda, b,
ldb, beta, c, ldc)
  type(cublasXtHandle) :: h
  integer :: transa, transb
  integer(kind=c_intptr_t) :: m, n, k, lda, ldb, ldc
  real(4), dimension(lda, *) :: a
  real(4), dimension(ldb, *) :: b
  real(4), dimension(ldc, *) :: c
  real(4) :: alpha, beta
```

2.7.11. cublasXtSsymm

SSYMM 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 scalars, A is a symmetric matrix and B and C are m by n matrices.

```
integer(4) function cublasXtssymm(h, side, uplo, m, n, alpha, a, lda, b, ldb,
beta, c, ldc)
  type(cublasXtHandle) :: h
  integer :: side, uplo
  integer(kind=c_intptr_t) :: m, n, lda, ldb, ldc
  real(4), dimension(lda, *) :: a
  real(4), dimension(ldb, *) :: b
  real(4), dimension(ldc, *) :: c
  real(4) :: alpha, beta
```

2.7.12. cublasXtSsyrk

SSYRK performs one of the symmetric rank k operations C := alpha*A*A**T + beta*C, or C := alpha*A**T*A + beta*C, where alpha and beta are scalars, C is an n by n symmetric matrix and A is an n by k matrix in the first case and a k by n matrix in the second case.

```
integer(4) function cublasXtssyrk(h, uplo, trans, n, k, alpha, a, lda, beta, c,
ldc)
  type(cublasXtHandle) :: h
  integer :: uplo, trans
```

```
  integer(kind=c_intptr_t) :: n, k, lda, ldc
  real(4), dimension(lda, *) :: a
  real(4), dimension(ldc, *) :: c
  real(4) :: alpha, beta
```

2.7.13. cublasXtSsyr2k

SSYR2K 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 scalars, C is an n by n symmetric matrix and A and B are n by k matrices in the first case and k by n matrices in the second case.

```
integer(4) function cublasXtssyr2k(h, uplo, trans, n, k, alpha, a, lda, b, ldb,
beta, c, ldc)
  type(cublasXtHandle) :: h
  integer :: uplo, trans
  integer(kind=c_intptr_t) :: n, k, lda, ldb, ldc
  real(4), dimension(lda, *) :: a
  real(4), dimension(ldb, *) :: b
  real(4), dimension(ldc, *) :: c
  real(4) :: alpha, beta
```

2.7.14. cublasXtSsyrkx

SSYRKX performs a variation of the symmetric rank k update C := alpha*A*B**T + beta*C, where alpha and beta are scalars, C is an n by n symmetric matrix stored in lower or upper mode, and A and B are n by k matrices. This routine can be used when B is in such a way that the result is guaranteed to be symmetric. See the CUBLAS documentation for more details.

```
integer(4) function cublasXtssyrkx(h, uplo, trans, n, k, alpha, a, lda, b, ldb,
beta, c, ldc)
  type(cublasXtHandle) :: h
  integer :: uplo, trans
  integer(kind=c_intptr_t) :: n, k, lda, ldb, ldc
  real(4), dimension(lda, *) :: a
  real(4), dimension(ldb, *) :: b
  real(4), dimension(ldc, *) :: c
  real(4) :: alpha, beta
```

2.7.15. cublasXtStrmm

STRMM performs one of the matrix-matrix operations B := alpha*op(A)*B, or B := alpha*B*op(A), where alpha is a scalar, B is an m by n matrix, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A**T.

```
integer(4) function cublasXtstrmm(h, side, uplo, transa, diag, m, n, alpha, a,
lda, b, ldb, c, ldc)
  type(cublasXtHandle) :: h
  integer :: side, uplo, transa, diag
  integer(kind=c_intptr_t) :: m, n, lda, ldb, ldc
  real(4), dimension(lda, *) :: a
  real(4), dimension(ldb, *) :: b
  real(4), dimension(ldc, *) :: c
  real(4) :: alpha
```

2.7.16. cublasXtStrsm

STRSM solves one of the matrix equations op(A)*X = alpha*B, or X*op(A) = alpha*B, where alpha is a scalar, X and B are m by n matrices, A is a unit, or non-unit, upper or

lower triangular matrix and op(A) is one of op(A) = A or op(A) = A**T. The matrix X is overwritten on B.

```
integer(4) function cublasXtstrsm(h, side, uplo, transa, diag, m, n, alpha, a,
lda, b, ldb)
  type(cublasXtHandle) :: h
  integer :: side, uplo, transa, diag
  integer(kind=c_intptr_t) :: m, n, lda, ldb
  real(4), dimension(lda, *) :: a
  real(4), dimension(ldb, *) :: b
  real(4) :: alpha
```

2.7.17. cublasXtSspmm

SSPMM performs one of the symmetric packed matrix-matrix operations C := alpha*A*B + beta*C, or C := alpha*B*A + beta*C, where alpha and beta are scalars, A is a n by n symmetric matrix stored in packed format, and B and C are m by n matrices.

```
integer(4) function cublasXtsspmm(h, side, uplo, m, n, alpha, ap, b, ldb, beta,
c, ldc)
  type(cublasXtHandle) :: h
  integer :: side, uplo
  integer(kind=c_intptr_t) :: m, n, ldb, ldc
  real(4), dimension(*) :: ap
  real(4), dimension(ldb, *) :: b
  real(4), dimension(ldc, *) :: c
  real(4) :: alpha, beta
```

2.7.18. cublasXtCgemm

CGEMM performs one of the matrix-matrix operations C := alpha*op(A)*op(B) + beta*C, where op(X) is one of op(X) = X or op(X) = X**T or op(X) = X**H, alpha and beta are scalars, and A, B and C are matrices, with op(A) an m by k matrix, op(B) a k by n matrix and C an m by n matrix.

```
integer(4) function cublasXtcgemm(h, transa, transb, m, n, k, alpha, a, lda, b,
ldb, beta, c, ldc)
  type(cublasXtHandle) :: h
  integer :: transa, transb
  integer(kind=c_intptr_t) :: m, n, k, lda, ldb, ldc
  complex(4), dimension(lda, *) :: a
  complex(4), dimension(ldb, *) :: b
  complex(4), dimension(ldc, *) :: c
  complex(4) :: alpha, beta
```

2.7.19. cublasXtChemm

CHEMM 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 scalars, A is an hermitian matrix and B and C are m by n matrices.

```
integer(4) function cublasXtchemm(h, side, uplo, m, n, alpha, a, lda, b, ldb,
beta, c, ldc)
  type(cublasXtHandle) :: h
  integer :: side, uplo
  integer(kind=c_intptr_t) :: m, n, lda, ldb, ldc
  complex(4), dimension(lda, *) :: a
  complex(4), dimension(ldb, *) :: b
  complex(4), dimension(ldc, *) :: c
  complex(4) :: alpha, beta
```

2.7.20. cublasXtCherk

CHERK performs one of the hermitian rank k operations C := alpha*A*A**H + beta*C, or C := alpha*A**H*A + beta*C, where alpha and beta are real scalars, C is an n by n hermitian matrix and A is an n by k matrix in the first case and a k by n matrix in the second case.

```
integer(4) function cublasXtcherk(h, uplo, trans, n, k, alpha, a, lda, beta, c,
ldc)
  type(cublasXtHandle) :: h
  integer :: uplo, trans
  integer(kind=c_intptr_t) :: n, k, lda, ldc
  complex(4), dimension(lda, *) :: a
  complex(4), dimension(ldc, *) :: c
  real(4) :: alpha, beta
```

2.7.21. cublasXtCher2k

CHER2K performs one of the hermitian rank 2k operations C := alpha*A*B**H + conjg(alpha)*B*A**H + beta*C, or C := alpha*A**H*B + conjg(alpha)*B**H*A + beta*C, where alpha and beta are scalars with beta real, C is an n by n hermitian matrix and A and B are n by k matrices in the first case and k by n matrices in the second case.

```
integer(4) function cublasXtcher2k(h, uplo, trans, n, k, alpha, a, lda, b, ldb,
beta, c, ldc)
  type(cublasXtHandle) :: h
  integer :: uplo, trans
  integer(kind=c_intptr_t) :: n, k, lda, ldb, ldc
  complex(4), dimension(lda, *) :: a
  complex(4), dimension(ldb, *) :: b
  complex(4), dimension(ldc, *) :: c
  complex(4) :: alpha
  real(4) :: beta
```

2.7.22. cublasXtCherkx

CHERKX performs a variation of the hermitian rank k operations C := alpha*A*B**H + beta*C, where alpha and beta are real scalars, C is an n by n hermitian matrix stored in lower or upper mode, and A and B are n by k matrices. See the CUBLAS documentation for more details.

```
integer(4) function cublasXtcherkx(h, uplo, trans, n, k, alpha, a, lda, b, ldb,
beta, c, ldc)
  type(cublasXtHandle) :: h
  integer :: uplo, trans
  integer(kind=c_intptr_t) :: n, k, lda, ldb, ldc
  complex(4), dimension(lda, *) :: a
  complex(4), dimension(ldb, *) :: b
  complex(4), dimension(ldc, *) :: c
  complex(4) :: alpha
  real(4) :: beta
```

2.7.23. cublasXtCsymm

CSYMM 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 scalars, A is a symmetric matrix and B and C are m by n matrices.

```
integer(4) function cublasXtcsymm(h, side, uplo, m, n, alpha, a, lda, b, ldb,
beta, c, ldc)
  type(cublasXtHandle) :: h
  integer :: side, uplo
  integer(kind=c_intptr_t) :: m, n, lda, ldb, ldc
  complex(4), dimension(lda, *) :: a
  complex(4), dimension(ldb, *) :: b
  complex(4), dimension(ldc, *) :: c
  complex(4) :: alpha, beta
```

2.7.24. cublasXtCsyrk

CSYRK performs one of the symmetric rank k operations C := alpha*A*A**T + beta*C, or C := alpha*A**T*A + beta*C, where alpha and beta are scalars, C is an n by n symmetric matrix and A is an n by k matrix in the first case and a k by n matrix in the second case.

```
integer(4) function cublasXtcsyrk(h, uplo, trans, n, k, alpha, a, lda, beta, c,
ldc)
  type(cublasXtHandle) :: h
  integer :: uplo, trans
  integer(kind=c_intptr_t) :: n, k, lda, ldc
  complex(4), dimension(lda, *) :: a
  complex(4), dimension(ldc, *) :: c
  complex(4) :: alpha, beta
```

2.7.25. cublasXtCsyr2k

CSYR2K 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 scalars, C is an n by n symmetric matrix and A and B are n by k matrices in the first case and k by n matrices in the second case.

```
integer(4) function cublasXtcsyr2k(h, uplo, trans, n, k, alpha, a, lda, b, ldb,
beta, c, ldc)
  type(cublasXtHandle) :: h
  integer :: uplo, trans
  integer(kind=c_intptr_t) :: n, k, lda, ldb, ldc
  complex(4), dimension(lda, *) :: a
  complex(4), dimension(ldb, *) :: b
  complex(4), dimension(ldc, *) :: c
  complex(4) :: alpha, beta
```

2.7.26. cublasXtCsyrkx

CSYRKX performs a variation of the symmetric rank k update C := alpha*A*B**T + beta*C, where alpha and beta are scalars, C is an n by n symmetric matrix stored in lower or upper mode, and A and B are n by k matrices. This routine can be used when B is in such a way that the result is guaranteed to be symmetric. See the CUBLAS documentation for more details.

```
integer(4) function cublasXtcsyrkx(h, uplo, trans, n, k, alpha, a, lda, b, ldb,
beta, c, ldc)
  type(cublasXtHandle) :: h
```

```
  integer :: uplo, trans
  integer(kind=c_intptr_t) :: n, k, lda, ldb, ldc
  complex(4), dimension(lda, *) :: a
  complex(4), dimension(ldb, *) :: b
  complex(4), dimension(ldc, *) :: c
  complex(4) :: alpha, beta
```

2.7.27. cublasXtCtrmm

CTRMM performs one of the matrix-matrix operations B := alpha*op(A)*B, or B := alpha*B*op(A) where alpha is a scalar, B is an m by n matrix, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A**T or op(A) = A**H.

```
integer(4) function cublasXtctrmm(h, side, uplo, transa, diag, m, n, alpha, a,
lda, b, ldb, c, ldc)
  type(cublasXtHandle) :: h
  integer :: side, uplo, transa, diag
  integer(kind=c_intptr_t) :: m, n, lda, ldb, ldc
  complex(4), dimension(lda, *) :: a
  complex(4), dimension(ldb, *) :: b
  complex(4), dimension(ldc, *) :: c
  complex(4) :: alpha
```

2.7.28. cublasXtCtrsm

CTRSM solves one of the matrix equations op(A)*X = alpha*B, or X*op(A) = alpha*B, where alpha is a scalar, X and B are m by n matrices, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A**T or op(A) = A**H. The matrix X is overwritten on B.

```
integer(4) function cublasXtctrsm(h, side, uplo, transa, diag, m, n, alpha, a,
lda, b, ldb)
  type(cublasXtHandle) :: h
  integer :: side, uplo, transa, diag
  integer(kind=c_intptr_t) :: m, n, lda, ldb
  complex(4), dimension(lda, *) :: a
  complex(4), dimension(ldb, *) :: b
  complex(4) :: alpha
```

2.7.29. cublasXtCspmm

CSPMM performs one of the symmetric packed matrix-matrix operations C := alpha*A*B + beta*C, or C := alpha*B*A + beta*C, where alpha and beta are scalars, A is a n by n symmetric matrix stored in packed format, and B and C are m by n matrices.

```
integer(4) function cublasXtcspmm(h, side, uplo, m, n, alpha, ap, b, ldb, beta,
c, ldc)
  type(cublasXtHandle) :: h
  integer :: side, uplo
  integer(kind=c_intptr_t) :: m, n, ldb, ldc
  complex(4), dimension(*) :: ap
  complex(4), dimension(ldb, *) :: b
  complex(4), dimension(ldc, *) :: c
  complex(4) :: alpha, beta
```

2.7.30. cublasXtDgemm

DGEMM performs one of the matrix-matrix operations C := alpha*op(A)*op(B) + beta*C, where op(X) is one of op(X) = X or op(X) = X**T, alpha and beta are scalars,

and A, B and C are matrices, with op(A) an m by k matrix, op(B) a k by n matrix and C an m by n matrix.

```
integer(4) function cublasXtdgemm(h, transa, transb, m, n, k, alpha, a, lda, b,
ldb, beta, c, ldc)
  type(cublasXtHandle) :: h
  integer :: transa, transb
  integer(kind=c_intptr_t) :: m, n, k, lda, ldb, ldc
  real(8), dimension(lda, *) :: a
  real(8), dimension(ldb, *) :: b
  real(8), dimension(ldc, *) :: c
  real(8) :: alpha, beta
```

2.7.31. cublasXtDsymm

DSYMM 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 scalars, A is a symmetric matrix and B and C are m by n matrices.

```
integer(4) function cublasXtdsymm(h, side, uplo, m, n, alpha, a, lda, b, ldb,
beta, c, ldc)
  type(cublasXtHandle) :: h
  integer :: side, uplo
  integer(kind=c_intptr_t) :: m, n, lda, ldb, ldc
  real(8), dimension(lda, *) :: a
  real(8), dimension(ldb, *) :: b
  real(8), dimension(ldc, *) :: c
  real(8) :: alpha, beta
```

2.7.32. cublasXtDsyrk

DSYRK performs one of the symmetric rank k operations C := alpha*A*A**T + beta*C, or C := alpha*A**T*A + beta*C, where alpha and beta are scalars, C is an n by n symmetric matrix and A is an n by k matrix in the first case and a k by n matrix in the second case.

```
integer(4) function cublasXtdsyrk(h, uplo, trans, n, k, alpha, a, lda, beta, c,
ldc)
  type(cublasXtHandle) :: h
  integer :: uplo, trans
  integer(kind=c_intptr_t) :: n, k, lda, ldc
  real(8), dimension(lda, *) :: a
  real(8), dimension(ldc, *) :: c
  real(8) :: alpha, beta
```

2.7.33. cublasXtDsyr2k

DSYR2K 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 scalars, C is an n by n symmetric matrix and A and B are n by k matrices in the first case and k by n matrices in the second case.

```
integer(4) function cublasXtdsyr2k(h, uplo, trans, n, k, alpha, a, lda, b, ldb,
beta, c, ldc)
  type(cublasXtHandle) :: h
  integer :: uplo, trans
  integer(kind=c_intptr_t) :: n, k, lda, ldb, ldc
  real(8), dimension(lda, *) :: a
  real(8), dimension(ldb, *) :: b
  real(8), dimension(ldc, *) :: c
  real(8) :: alpha, beta
```

2.7.34. cublasXtDsyrkx

DSYRKX performs a variation of the symmetric rank k update C := alpha*A*B**T + beta*C, where alpha and beta are scalars, C is an n by n symmetric matrix stored in lower or upper mode, and A and B are n by k matrices. This routine can be used when B is in such a way that the result is guaranteed to be symmetric. See the CUBLAS documentation for more details.

```
integer(4) function cublasXtdsyrkx(h, uplo, trans, n, k, alpha, a, lda, b, ldb,
beta, c, ldc)
  type(cublasXtHandle) :: h
  integer :: uplo, trans
  integer(kind=c_intptr_t) :: n, k, lda, ldb, ldc
  real(8), dimension(lda, *) :: a
  real(8), dimension(ldb, *) :: b
  real(8), dimension(ldc, *) :: c
  real(8) :: alpha, beta
```

2.7.35. cublasXtDtrmm

DTRMM performs one of the matrix-matrix operations B := alpha*op(A)*B, or B := alpha*B*op(A), where alpha is a scalar, B is an m by n matrix, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A**T.

```
integer(4) function cublasXtdtrmm(h, side, uplo, transa, diag, m, n, alpha, a,
lda, b, ldb, c, ldc)
  type(cublasXtHandle) :: h
  integer :: side, uplo, transa, diag
  integer(kind=c_intptr_t) :: m, n, lda, ldb, ldc
  real(8), dimension(lda, *) :: a
  real(8), dimension(ldb, *) :: b
  real(8), dimension(ldc, *) :: c
  real(8) :: alpha
```

2.7.36. cublasXtDtrsm

DTRSM solves one of the matrix equations op(A)*X = alpha*B, or X*op(A) = alpha*B, where alpha is a scalar, X and B are m by n matrices, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A**T. The matrix X is overwritten on B.

```
integer(4) function cublasXtdtrsm(h, side, uplo, transa, diag, m, n, alpha, a,
lda, b, ldb)
  type(cublasXtHandle) :: h
  integer :: side, uplo, transa, diag
  integer(kind=c_intptr_t) :: m, n, lda, ldb
  real(8), dimension(lda, *) :: a
  real(8), dimension(ldb, *) :: b
  real(8) :: alpha
```

2.7.37. cublasXtDspmm

DSPMM performs one of the symmetric packed matrix-matrix operations C := alpha*A*B + beta*C, or C := alpha*B*A + beta*C, where alpha and beta are scalars, A is a n by n symmetric matrix stored in packed format, and B and C are m by n matrices.

```
integer(4) function cublasXtdspmm(h, side, uplo, m, n, alpha, ap, b, ldb, beta,
c, ldc)
  type(cublasXtHandle) :: h
```

```
  integer :: side, uplo
  integer(kind=c_intptr_t) :: m, n, ldb, ldc
  real(8), dimension(*) :: ap
  real(8), dimension(ldb, *) :: b
  real(8), dimension(ldc, *) :: c
  real(8) :: alpha, beta
```

2.7.38. cublasXtZgemm

ZGEMM performs one of the matrix-matrix operations C := alpha*op(A)*op(B) + beta*C, where op(X) is one of op(X) = X or op(X) = X**T or op(X) = X**H, alpha and beta are scalars, and A, B and C are matrices, with op(A) an m by k matrix, op(B) a k by n matrix and C an m by n matrix.

```
integer(4) function cublasXtzgemm(h, transa, transb, m, n, k, alpha, a, lda, b,
ldb, beta, c, ldc)
  type(cublasXtHandle) :: h
  integer :: transa, transb
  integer(kind=c_intptr_t) :: m, n, k, lda, ldb, ldc
  complex(8), dimension(lda, *) :: a
  complex(8), dimension(ldb, *) :: b
  complex(8), dimension(ldc, *) :: c
  complex(8) :: alpha, beta
```

2.7.39. cublasXtZhemm

ZHEMM 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 scalars, A is an hermitian matrix and B and C are m by n matrices.

```
integer(4) function cublasXtzhemm(h, side, uplo, m, n, alpha, a, lda, b, ldb,
beta, c, ldc)
  type(cublasXtHandle) :: h
  integer :: side, uplo
  integer(kind=c_intptr_t) :: m, n, lda, ldb, ldc
  complex(8), dimension(lda, *) :: a
  complex(8), dimension(ldb, *) :: b
  complex(8), dimension(ldc, *) :: c
  complex(8) :: alpha, beta
```

2.7.40. cublasXtZherk

ZHERK performs one of the hermitian rank k operations C := alpha*A*A**H + beta*C, or C := alpha*A**H*A + beta*C, where alpha and beta are real scalars, C is an n by n hermitian matrix and A is an n by k matrix in the first case and a k by n matrix in the second case.

```
integer(4) function cublasXtzherk(h, uplo, trans, n, k, alpha, a, lda, beta, c,
ldc)
  type(cublasXtHandle) :: h
  integer :: uplo, trans
  integer(kind=c_intptr_t) :: n, k, lda, ldc
  complex(8), dimension(lda, *) :: a
  complex(8), dimension(ldc, *) :: c
  real(8) :: alpha, beta
```

2.7.41. cublasXtZher2k

ZHER2K performs one of the hermitian rank 2k operations C := alpha*A*B**H + conjg(alpha)*B*A**H + beta*C, or C := alpha*A**H*B + conjg(alpha)*B**H*A + beta*C,

where alpha and beta are scalars with beta real, C is an n by n hermitian matrix and A and B are n by k matrices in the first case and k by n matrices in the second case.

```
integer(4) function cublasXtzher2k(h, uplo, trans, n, k, alpha, a, lda, b, ldb,
beta, c, ldc)
  type(cublasXtHandle) :: h
  integer :: uplo, trans
  integer(kind=c_intptr_t) :: n, k, lda, ldb, ldc
  complex(8), dimension(lda, *) :: a
  complex(8), dimension(ldb, *) :: b
  complex(8), dimension(ldc, *) :: c
  complex(8) :: alpha
  real(8) :: beta
```

2.7.42. cublasXtZherkx

ZHERKX performs a variation of the hermitian rank k operations C := alpha*A*B**H + beta*C, where alpha and beta are real scalars, C is an n by n hermitian matrix stored in lower or upper mode, and A and B are n by k matrices. See the CUBLAS documentation for more details.

```
integer(4) function cublasXtzherkx(h, uplo, trans, n, k, alpha, a, lda, b, ldb,
beta, c, ldc)
  type(cublasXtHandle) :: h
  integer :: uplo, trans
  integer(kind=c_intptr_t) :: n, k, lda, ldb, ldc
  complex(8), dimension(lda, *) :: a
  complex(8), dimension(ldb, *) :: b
  complex(8), dimension(ldc, *) :: c
  complex(8) :: alpha
  real(8) :: beta
```

2.7.43. cublasXtZsymm

ZSYMM 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 scalars, A is a symmetric matrix and B and C are m by n matrices.

```
integer(4) function cublasXtzsymm(h, side, uplo, m, n, alpha, a, lda, b, ldb,
beta, c, ldc)
  type(cublasXtHandle) :: h
  integer :: side, uplo
  integer(kind=c_intptr_t) :: m, n, lda, ldb, ldc
  complex(8), dimension(lda, *) :: a
  complex(8), dimension(ldb, *) :: b
  complex(8), dimension(ldc, *) :: c
  complex(8) :: alpha, beta
```

2.7.44. cublasXtZsyrk

ZSYRK performs one of the symmetric rank k operations C := alpha*A*A**T + beta*C, or C := alpha*A**T*A + beta*C, where alpha and beta are scalars, C is an n by n symmetric matrix and A is an n by k matrix in the first case and a k by n matrix in the second case.

```
integer(4) function cublasXtzsyrk(h, uplo, trans, n, k, alpha, a, lda, beta, c,
ldc)
  type(cublasXtHandle) :: h
  integer :: uplo, trans
  integer(kind=c_intptr_t) :: n, k, lda, ldc
  complex(8), dimension(lda, *) :: a
  complex(8), dimension(ldc, *) :: c
  complex(8) :: alpha, beta
```

2.7.45. cublasXtZsyr2k

ZSYR2K 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 scalars, C is an n by n symmetric matrix and A and B are n by k matrices in the first case and k by n matrices in the second case.

```
integer(4) function cublasXtzsyr2k(h, uplo, trans, n, k, alpha, a, lda, b, ldb,
beta, c, ldc)
  type(cublasXtHandle) :: h
  integer :: uplo, trans
  integer(kind=c_intptr_t) :: n, k, lda, ldb, ldc
  complex(8), dimension(lda, *) :: a
  complex(8), dimension(ldb, *) :: b
  complex(8), dimension(ldc, *) :: c
  complex(8) :: alpha, beta
```

2.7.46. cublasXtZsyrkx

ZSYRKX performs a variation of the symmetric rank k update C := alpha*A*B**T + beta*C, where alpha and beta are scalars, C is an n by n symmetric matrix stored in lower or upper mode, and A and B are n by k matrices. This routine can be used when B is in such a way that the result is guaranteed to be symmetric. See the CUBLAS documentation for more details.

```
integer(4) function cublasXtzsyrkx(h, uplo, trans, n, k, alpha, a, lda, b, ldb,
beta, c, ldc)
  type(cublasXtHandle) :: h
  integer :: uplo, trans
  integer(kind=c_intptr_t) :: n, k, lda, ldb, ldc
  complex(8), dimension(lda, *) :: a
  complex(8), dimension(ldb, *) :: b
  complex(8), dimension(ldc, *) :: c
  complex(8) :: alpha, beta
```

2.7.47. cublasXtZtrmm

ZTRMM performs one of the matrix-matrix operations B := alpha*op(A)*B, or B := alpha*B*op(A) where alpha is a scalar, B is an m by n matrix, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A**T or op(A) = A**H.

```
integer(4) function cublasXtztrmm(h, side, uplo, transa, diag, m, n, alpha, a,
lda, b, ldb, c, ldc)
  type(cublasXtHandle) :: h
  integer :: side, uplo, transa, diag
  integer(kind=c_intptr_t) :: m, n, lda, ldb, ldc
  complex(8), dimension(lda, *) :: a
  complex(8), dimension(ldb, *) :: b
  complex(8), dimension(ldc, *) :: c
  complex(8) :: alpha
```

2.7.48. cublasXtZtrsm

ZTRSM solves one of the matrix equations op(A)*X = alpha*B, or X*op(A) = alpha*B, where alpha is a scalar, X and B are m by n matrices, A is a unit, or non-unit, upper or

lower triangular matrix and op(A) is one of op(A) = A or op(A) = A**T or op(A) = A**H. The matrix X is overwritten on B.

```
integer(4) function cublasXtztrsm(h, side, uplo, transa, diag, m, n, alpha, a,
lda, b, ldb)
  type(cublasXtHandle) :: h
  integer :: side, uplo, transa, diag
  integer(kind=c_intptr_t) :: m, n, lda, ldb
  complex(8), dimension(lda, *) :: a
  complex(8), dimension(ldb, *) :: b
  complex(8) :: alpha
```

2.7.49. cublasXtZspmm

ZSPMM performs one of the symmetric packed matrix-matrix operations C := alpha*A*B + beta*C, or C := alpha*B*A + beta*C, where alpha and beta are scalars, A is a n by n symmetric matrix stored in packed format, and B and C are m by n matrices.

```
integer(4) function cublasXtzspmm(h, side, uplo, m, n, alpha, ap, b, ldb, beta,
c, ldc)
  type(cublasXtHandle) :: h
  integer :: side, uplo
  integer(kind=c_intptr_t) :: m, n, ldb, ldc
  complex(8), dimension(*) :: ap
  complex(8), dimension(ldb, *) :: b
  complex(8), dimension(ldc, *) :: c
  complex(8) :: alpha, beta
```

2.8. CUBLAS DEVICE Module Functions

This section contains interfaces to the cuBLAS functions accessible from device code. CUDA Fortran users can access this module by inserting the line **`use cublas_device`** into the program unit. OpenACC users can access this module by inserting the line **`use openacc_cublas`** into the program unit. Examples for making cuBLAS calls from device code are included in the examples chapter.

The **`cublas_device`** module and the **`openacc_cublas`** module contain all the types and definitions from the **`cublas`** module:

```
TYPE cublasHandle
  TYPE(C_PTR)  :: handle
END TYPE
```

Each device module contains the following enumerations:

```
enum, bind(c)
    enumerator :: CUBLAS_STATUS_SUCCESS         =0
    enumerator :: CUBLAS_STATUS_NOT_INITIALIZED =1
    enumerator :: CUBLAS_STATUS_ALLOC_FAILED    =3
    enumerator :: CUBLAS_STATUS_INVALID_VALUE   =7
    enumerator :: CUBLAS_STATUS_ARCH_MISMATCH   =8
    enumerator :: CUBLAS_STATUS_MAPPING_ERROR   =11
    enumerator :: CUBLAS_STATUS_EXECUTION_FAILED=13
    enumerator :: CUBLAS_STATUS_INTERNAL_ERROR  =14
end enum
```

```
enum, bind(c)
    enumerator :: CUBLAS_FILL_MODE_LOWER=0
    enumerator :: CUBLAS_FILL_MODE_UPPER=1
end enum
```

```
enum, bind(c)
```

```
    enumerator :: CUBLAS_DIAG_NON_UNIT=0
    enumerator :: CUBLAS_DIAG_UNIT=1
end enum
```

```
enum, bind(c)
    enumerator :: CUBLAS_SIDE_LEFT =0
    enumerator :: CUBLAS_SIDE_RIGHT=1
end enum
```

```
enum, bind(c)
    enumerator :: CUBLAS_OP_N=0
    enumerator :: CUBLAS_OP_T=1
    enumerator :: CUBLAS_OP_C=2
end enum
```

```
enum, bind(c)
    enumerator :: CUBLAS_POINTER_MODE_HOST   = 0
    enumerator :: CUBLAS_POINTER_MODE_DEVICE = 1
end enum
```

2.8.1. Device Library Helper Functions

This section contains the cuBLAS interfaces to the device-side single precision BLAS and cuBLAS functions and subroutines.

2.8.1.1. cublasCreate

This function initializes the CUBLAS library and creates a handle to an opaque structure holding the CUBLAS library context. It allocates hardware resources on the host and device and must be called prior to making any other CUBLAS library calls. The CUBLAS library context is tied to the current CUDA device. To use the library on multiple devices, one CUBLAS handle needs to be created for each device. Furthermore, for a given device, multiple CUBLAS handles with different configuration can be created. Because cublasCreate allocates some internal resources and the release of those resources by calling cublasDestroy will implicitly call cublasDeviceSynchronize, it is recommended to minimize the number of cublasCreate/cublasDestroy occurences. For multi-threaded applications that use the same device from different threads, the recommended programming model is to create one CUBLAS handle per thread and use that CUBLAS handle for the entire life of the thread. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
 integer(4) function cublasCreate(handle)
   type(cublasHandle) :: handle
```

2.8.1.2. cublasDestroy

This function releases hardware resources used by the CUBLAS library. This function is usually the last call with a particular handle to the CUBLAS library. Because cublasCreate allocates some internal resources and the release of those resources by calling cublasDestroy will implicitly call cublasDeviceSynchronize, it is recommended to minimize the number of cublasCreate/cublasDestroy occurences. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
 integer(4) function cublasDestroy(handle)
   type(cublasHandle) :: handle
```

2.8.1.3. cublasGetVersion

This function returns the version number of the cuBLAS library. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasGetVersion(handle, version)
  type(cublasHandle) :: handle
  integer(4) :: version
```

2.8.1.4. cublasSetStream

This function sets the cuBLAS library stream, which will be used to execute all subsequent calls to the cuBLAS library functions. If the cuBLAS library stream is not set, all kernels use the default NULL stream. In particular, this routine can be used to change the stream between kernel launches and then to reset the cuBLAS library stream back to NULL. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasSetStream(handle, stream)
  type(cublasHandle) :: handle
  integer(kind=cuda_stream_kind()) :: stream
```

2.8.1.5. cublasGetStream

This function gets the cuBLAS library stream, which is being used to execute all calls to the cuBLAS library functions. If the cuBLAS library stream is not set, all kernels use the default NULL stream. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasGetStream(handle, stream)
  type(cublasHandle) :: handle
  integer(kind=cuda_stream_kind()) :: stream
```

2.8.2. Single Precision Functions and Subroutines

This section contains the cuBLAS interfaces to the device-side single precision BLAS and cuBLAS functions and subroutines.

2.8.2.1. cublasIsamax

ISAMAX finds the index of the first element having maximum absolute value. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasisamax(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  real(4), device, dimension(*) :: x
  integer :: incx
  integer :: res
```

2.8.2.2. cublasIsamin

ISAMIN finds the index of the first element having minimum absolute value. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasisamin(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  real(4), device, dimension(*) :: x
  integer :: incx
  integer :: res
```

2.8.2.3. cublasSasum

SASUM takes the sum of the absolute values. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublassasum(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  real(4), device, dimension(*) :: x
  integer :: incx
  real(4) :: res
```

2.8.2.4. cublasSaxpy

SAXPY constant times a vector plus a vector. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublassaxpy(h, n, a, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  real(4) :: a
  real(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

2.8.2.5. cublasScopy

SCOPY copies a vector, x, to a vector, y. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasscopy(h, n, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  real(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

2.8.2.6. cublasSdot

SDOT forms the dot product of two vectors. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublassdot(h, n, x, incx, y, incy, res)
  type(cublasHandle) :: h
  integer :: n
  real(4), device, dimension(*) :: x, y
  integer :: incx, incy
  real(4) :: res
```

2.8.2.7. cublasSnrm2

SNRM2 returns the euclidean norm of a vector via the function name, so that SNRM2 := sqrt(x'*x). Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublassnrm2(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  real(4), device, dimension(*) :: x
  integer :: incx
  real(4) :: res
```

2.8.2.8. cublasSrot

SROT applies a plane rotation. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublassrot(h, n, x, incx, y, incy, sc, ss)
  type(cublasHandle) :: h
  integer :: n
  real(4) :: sc, ss
  real(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

2.8.2.9. cublasSrotg

SROTG constructs a Givens plane rotation. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublassrotg(h, sa, sb, sc, ss)
  type(cublasHandle) :: h
  real(4) :: sa, sb, sc, ss
```

2.8.2.10. cublasSrotm

SROTM applies the modified Givens transformation, H, to the 2 by N matrix (SX**T) , where **T indicates transpose. The elements of SX are in (SX**T) SX(LX+I*INCX), I = 0 to N-1, where LX = 1 if INCX .GE. 0, ELSE LX = (-INCX)*N, and similarly for SY using LY and INCY. With SPARAM(1)=SFLAG, H has one of the following forms.. SFLAG=-1.E0 SFLAG=0.E0 SFLAG=1.E0 SFLAG=-2.E0 (SH11 SH12) (1.E0 SH12) (SH11 1.E0) (1.E0 0.E0) H=( ) ( ) ( ) ( ) (SH21 SH22), (SH21 1.E0), (-1.E0 SH22), (0.E0 1.E0). See SROTMG for a description of data storage in SPARAM. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublassrotm(h, n, x, incx, y, incy, param)
  type(cublasHandle) :: h
  integer :: n
  real(4) :: param(*)
  real(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

2.8.2.11. cublasSrotmg

SROTMG constructs the modified Givens transformation matrix H which zeros the second component of the 2-vector (SQRT(SD1)*SX1,SQRT(SD2)*SY2)**T. With SPARAM(1)=SFLAG, H has one of the following forms..

```
      SFLAG=-1.E0     SFLAG=0.E0        SFLAG=1.E0     SFLAG=-2.E0

        (SH11  SH12)    (1.E0  SH12)    (SH11  1.E0)    (1.E0  0.E0)
      H=(          )    (          )    (          )    (          )
        (SH21  SH22),   (SH21  1.E0),   (-1.E0 SH22),   (0.E0  1.E0).
```

Locations 2-4 of SPARAM contain SH11,SH21,SH12, and SH22 respectively. (Values of 1.E0, -1.E0, or 0.E0 implied by the value of SPARAM(1) are not stored in SPARAM.) Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublassrotmg(h, d1, d2, x1, y1, param)
  type(cublasHandle) :: h
  real(4) :: d1, d2, x1, y1, param(*)
```

2.8.2.12. cublasSscal

SSCAL scales a vector by a constant. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublassscal(h, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: n
  real(4) :: a
  real(4), device, dimension(*) :: x
  integer :: incx
```

2.8.2.13. cublasSswap

SSWAP interchanges two vectors. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublassswap(h, n, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  real(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

2.8.2.14. cublasSgbmv

SGBMV performs one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A**T*x + beta*y, where alpha and beta are scalars, x and y are vectors and A is an m by n band matrix, with kl sub-diagonals and ku super-diagonals. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublassgbmv(h, t, m, n, kl, ku, alpha, a, lda,
            x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: m, n, kl, ku, lda, incx, incy
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x, y
  real(4) :: alpha, beta
```

2.8.2.15. cublasSgemv

SGEMV performs one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A**T*x + beta*y, where alpha and beta are scalars, x and y are vectors and A is an

m by n matrix. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublassgemv(h, t, m, n, alpha, a, lda,
    x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: m, n, lda, incx, incy
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x, y
  real(4) :: alpha, beta
```

2.8.2.16. cublasSger

SGER performs the rank 1 operation A := alpha*x*y**T + A, where alpha is a scalar, x is an m element vector, y is an n element vector and A is an m by n matrix. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublassger(h, m, n, alpha, x, incx, y, incy,
          a, lda)
  type(cublasHandle) :: h
  integer :: m, n, lda, incx, incy
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x, y
  real(4) :: alpha
```

2.8.2.17. cublasSsbmv

SSBMV performs the matrix-vector operation y := alpha*A*x + beta*y, where alpha and beta are scalars, x and y are n element vectors and A is an n by n symmetric band matrix, with k super-diagonals. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasssbmv(h, t, n, k, alpha, a, lda,
    x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: k, n, lda, incx, incy
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x, y
  real(4) :: alpha, beta
```

2.8.2.18. cublasSspmv

SSPMV performs the matrix-vector operation y := alpha*A*x + beta*y, where alpha and beta are scalars, x and y are n element vectors and A is an n by n symmetric matrix, supplied in packed form. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublassspmv(h, t, n, alpha, a, x, incx,
     beta, y, incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy
  real(4), device, dimension(*) :: a, x, y
  real(4) :: alpha, beta
```

2.8.2.19. cublasSspr

SSPR performs the symmetric rank 1 operation A := alpha*x*x**T + A, where alpha is a real scalar, x is an n element vector and A is an n by n symmetric matrix, supplied in packed form. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublassspr(h, t, n, alpha, x, incx, a)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx
  real(4), device, dimension(*) :: a, x
  real(4) :: alpha
```

2.8.2.20. cublasSspr2

SSPR2 performs the symmetric rank 2 operation A := alpha*x*y**T + alpha*y*x**T + A, where alpha is a scalar, x and y are n element vectors and A is an n by n symmetric matrix, supplied in packed form. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublassspr2(h, t, n, alpha, x, incx,
   y, incy, a)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy
  real(4), device, dimension(*) :: a, x, y
  real(4) :: alpha
```

2.8.2.21. cublasSsymv

SSYMV performs the matrix-vector operation y := alpha*A*x + beta*y, where alpha and beta are scalars, x and y are n element vectors and A is an n by n symmetric matrix. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasssymv(h, t, n, alpha, a, lda,
 x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, lda, incx, incy
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x, y
  real(4) :: alpha, beta
```

2.8.2.22. cublasSsyr

SSYR performs the symmetric rank 1 operation A := alpha*x*x**T + A, where alpha is a real scalar, x is an n element vector and A is an n by n symmetric matrix. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasssyr(h, t, n, alpha, x, incx, a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, lda
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x
  real(4) :: alpha
```

2.8.2.23. cublasSsyr2

SSYR2 performs the symmetric rank 2 operation A := alpha*x*y**T + alpha*y*x**T + A, where alpha is a scalar, x and y are n element vectors and A is an n by n symmetric matrix. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasssyr2(h, t, n, alpha, x, incx,
  y, incy, a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy, lda
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x, y
  real(4) :: alpha
```

2.8.2.24. cublasStbmv

STBMV performs one of the matrix-vector operations x := A*x, or x := A**T*x, where x is an n element vector and A is an n by n unit, or non-unit, upper or lower triangular band matrix, with (k + 1) diagonals. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasstbmv(h, u, t, d, n, k, a, lda,
    x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, k, incx, lda
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x
```

2.8.2.25. cublasStbsv

STBSV solves one of the systems of equations A*x = b, or A**T*x = b, where b and x are n element vectors and A is an n by 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 routine. Such tests must be performed before calling this routine. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasstbsv(h, u, t, d, n, k, a, lda,
    x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, k, incx, lda
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x
```

2.8.2.26. cublasStpmv

STPMV performs one of the matrix-vector operations x := A*x, or x := A**T*x, where x is an n element vector and A is an n by n unit, or non-unit, upper or lower triangular matrix, supplied in packed form. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasstpmv(h, u, t, d, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx
```

```
  real(4), device, dimension(*) :: a, x
```

2.8.2.27. cublasStpsv

STPSV solves one of the systems of equations A*x = b, or A**T*x = b, where b and x are n element vectors and A is an n by n unit, or non-unit, upper or lower triangular matrix, supplied in packed form. No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasstpsv(h, u, t, d, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx
  real(4), device, dimension(*) :: a, x
```

2.8.2.28. cublasStrmv

STRMV performs one of the matrix-vector operations x := A*x, or x := A**T*x, where x is an n element vector and A is an n by n unit, or non-unit, upper or lower triangular matrix. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasstrmv(h, u, t, d, n, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx, lda
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x
```

2.8.2.29. cublasStrsv

STRSV solves one of the systems of equations A*x = b, or A**T*x = b, where b and x are n element vectors and A is an n by n unit, or non-unit, upper or lower triangular matrix. No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasstrsv(h, u, t, d, n, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx, lda
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x
```

2.8.2.30. cublasSgemm

SGEMM performs one of the matrix-matrix operations C := alpha*op(A)*op(B) + beta*C, where op(X) is one of op(X) = X or op(X) = X**T, alpha and beta are scalars, and A, B and C are matrices, with op(A) an m by k matrix, op(B) a k by n matrix and C an m by n matrix. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublassgemm(h, transa, transb, m, n,
               k, alpha, a, lda, b, ldb, beta, c, ldc)
  type(cublasHandle) :: h
  integer :: transa, transb
  integer :: m, n, k, lda, ldb, ldc
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(ldb, *) :: b
```

```
  real(4), device, dimension(ldc, *) :: c
  real(4) :: alpha, beta
```

2.8.2.31. cublasSsymm

SSYMM 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 scalars, A is a symmetric matrix and B and C are m by n matrices. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasssymm(h, side, uplo, m, n,
          alpha, a, lda, b, ldb, beta, c, ldc)
  type(cublasHandle) :: h
  integer :: side, uplo
  integer :: m, n, lda, ldb, ldc
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(ldb, *) :: b
  real(4), device, dimension(ldc, *) :: c
  real(4) :: alpha, beta
```

2.8.2.32. cublasSsyrk

SSYRK performs one of the symmetric rank k operations C := alpha*A*A**T + beta*C, or C := alpha*A**T*A + beta*C, where alpha and beta are scalars, C is an n by n symmetric matrix and A is an n by k matrix in the first case and a k by n matrix in the second case. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasssyrk(h, uplo, trans, n, k,
                    alpha, a, lda, beta, c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldc
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(ldc, *) :: c
  real(4) :: alpha, beta
```

2.8.2.33. cublasSsyr2k

SSYR2K 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 scalars, C is an n by n symmetric matrix and A and B are n by k matrices in the first case and k by n matrices in the second case. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasssyr2k(h, uplo, trans, n, k,
              alpha, a, lda, b, ldb, beta, c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(ldb, *) :: b
  real(4), device, dimension(ldc, *) :: c
  real(4) :: alpha, beta
```

2.8.2.34. cublasStrmm

STRMM performs one of the matrix-matrix operations B := alpha*op(A)*B, or B := alpha*B*op(A), where alpha is a scalar, B is an m by n matrix, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A**T.

Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasstrmm(h, side, uplo, transa,
        diag, m, n, alpha, a, lda, b, ldb, c, ldc)
  type(cublasHandle) :: h
  integer :: side, uplo, transa, diag
  integer :: m, n, lda, ldb, ldc
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(ldb, *) :: b
  real(4), device, dimension(ldc, *) :: c
  real(4) :: alpha
```

2.8.2.35. cublasStrsm

STRSM solves one of the matrix equations op(A)*X = alpha*B, or X*op(A) = alpha*B, where alpha is a scalar, X and B are m by n matrices, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A**T. The matrix X is overwritten on B. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasstrsm(h, side, uplo, transa,
                diag, m, n, alpha, a, lda, b, ldb)
  type(cublasHandle) :: h
  integer :: side, uplo, transa, diag
  integer :: m, n, lda, ldb
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(ldb, *) :: b
  real(4) :: alpha
```

2.8.2.36. cublasSgemmBatched

SGEMM performs one of the matrix-matrix operations C := alpha*op(A)*op(B) + beta*C, where op(X) is one of op(X) = X or op(X) = X**T, alpha and beta are scalars, and A, B and C are matrices, with op(A) an m by k matrix, op(B) a k by n matrix and C an m by n matrix. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasSgemmBatched(h, transa, transb,                 m, n,
k, alpha, Aarray, lda, Barray, ldb, beta, Carray, ldc, batchCount)
  type(cublasHandle) :: h
  integer :: transa
  integer :: transb
  integer :: m, n, k
  real(4) :: alpha
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: Barray(*)
  integer :: ldb
  real(4) :: beta
  type(c_devptr), device :: Carray(*)
  integer :: ldc
  integer :: batchCount
```

2.8.2.37. cublasSgetrfBatched

SGETRF computes an LU factorization of a general M-by-N matrix A using partial pivoting with row interchanges. The factorization has the form A = P * L * U where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if m > n), and U is upper triangular (upper trapezoidal if m < n). This is the

right-looking Level 3 BLAS version of the algorithm. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasSgetrfBatched(h, n, Aarray, lda,
                                        ipvt, info, batchCount)
  type(cublasHandle) :: h
  integer :: n
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  integer, device :: ipvt(*)
  integer, device :: info(*)
  integer :: batchCount
```

2.8.2.38. cublasSgetriBatched

SGETRI computes the inverse of a matrix using the LU factorization computed by SGETRF. This method inverts U and then computes inv(A) by solving the system inv(A)*L = inv(U) for inv(A). Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasSgetriBatched(h, n, Aarray, lda,
                   ipvt, Carray, ldc, info, batchCount)
  type(cublasHandle) :: h
  integer :: n
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  integer, device :: ipvt(*)
  type(c_devptr), device :: Carray(*)
  integer :: ldc
  integer, device :: info(*)
  integer :: batchCount
```

2.8.2.39. cublasSgetrsBatched

SGETRS solves a system of linear equations A * X = B or A**T * X = B with a general N-by-N matrix A using the LU factorization computed by SGETRF. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasSgetrsBatched(h, trans, n, nrhs,
                A, lda, ipvt, B, ldb, info, batchCount)
  type(cublasHandle) :: h
  integer :: trans
  integer :: n, nrhs
  type(c_devptr), device :: A(*)
  integer :: lda
  integer, device :: ipvt(*)
  type(c_devptr), device :: B(*)
  integer :: ldb
  integer, device :: info(*)
  integer :: batchCount
```

2.8.2.40. cublasStrsmBatched

STRSM solves one of the matrix equations op(A)*X = alpha*B, or X*op(A) = alpha*B, where alpha is a scalar, X and B are m by n matrices, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A**T. The matrix X is overwritten on B. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasStrsmBatched(h, side, uplo,
    trans, diag, m, n, alpha, A, lda, B, ldb, batchCount)
  type(cublasHandle) :: h
```

```
  integer :: side
  integer :: uplo
  integer :: trans
  integer :: diag
  integer :: m, n
  real(4) :: alpha
  type(c_devptr), device :: A(*)
  integer :: lda
  type(c_devptr), device :: B(*)
  integer :: ldb
  integer :: batchCount
```

2.8.2.41. cublasSmatinvBatched

cublasSmatinvBatched is a short cut of cublasSgetrfBatched plus cublasSgetriBatched. However it only works if n is less than 32. If not, the user has to go through cublasSgetrfBatched and cublasSgetriBatched. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasSmatinvBatched(h, n, A, lda,
               Ainv, lda_inv, info, batchCount)
  type(cublasHandle) :: h
  integer :: n
  type(c_devptr), device :: A(*)
  integer :: lda
  type(c_devptr), device :: Ainv(*)
  integer :: lda_inv
  integer, device :: info(*)
  integer :: batchCount
```

2.8.2.42. cublasSgeqrfBatched

SGEQRF computes a QR factorization of a real M-by-N matrix A: A = Q * R. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasSgeqrfBatched(h, m, n,
     A, lda, Tau, info, batchCount)
  type(cublasHandle) :: h
  integer :: m, n
  type(c_devptr), device :: A(*)
  integer :: lda
  type(c_devptr), device :: Tau(*)
  integer, device :: info(*)
  integer :: batchCount
```

2.8.2.43. cublasSgelsBatched

SGELS solves overdetermined or underdetermined real linear systems involving an M-by-N matrix A, or its transpose, using a QR or LQ factorization of A. It is assumed that A has full rank. The following options are provided: 1. If TRANS = 'N' and m >= n: find the least squares solution of an overdetermined system, i.e., solve the least squares problem minimize || B - A*X ||. 2. If TRANS = 'N' and m < n: find the minimum norm solution of an underdetermined system A * X = B. 3. If TRANS = 'T' and m >= n: find the minimum norm solution of an undetermined system A**T * X = B. 4. If TRANS = 'T' and m < n: find the least squares solution of an overdetermined system, i.e., solve the least squares problem minimize || B - A**T * X ||. Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the M-by-NRHS right hand side matrix B and the N-by-NRHS solution matrix X. Device

Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasSgelsBatched(h, trans, m, n,
 nrhs, A, lda, C, ldc, info, dinfo, batchCount)
  type(cublasHandle) :: h
  integer :: trans
  integer :: m, n, nrhs
  type(c_devptr), device :: A(*)
  integer :: lda
  type(c_devptr), device :: C(*)
  integer :: ldc
  integer, device :: info(*)
  integer, device :: dinfo(*)
  integer :: batchCount
```

2.8.3. Single Precision Complex Functions and Subroutines

This section contains the cuBLAS interfaces to the device-side single precision complex BLAS and cuBLAS functions and subroutines.

2.8.3.1. cublasCaxpy

CAXPY constant times a vector plus a vector. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublascaxpy(h, n, a, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  complex(4) :: a
  complex(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

2.8.3.2. cublasCcopy

CCOPY copies a vector x to a vector y. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasccopy(h, n, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  complex(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

2.8.3.3. cublasCdotc

forms the dot product of two vectors, conjugating the first vector. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublascdotc(h, n, x, incx, y, incy, res)
  type(cublasHandle) :: h
  integer :: n
  complex(4), device, dimension(*) :: x, y
  integer :: incx, incy
  complex(4) :: res
```

2.8.3.4. cublasCdotu

CDOTU forms the dot product of two vectors. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublascdotu(h, n, x, incx, y, incy, res)
  type(cublasHandle) :: h
  integer :: n
  complex(4), device, dimension(*) :: x, y
  integer :: incx, incy
  complex(4) :: res
```

2.8.3.5. cublasCrot

CROT applies a plane rotation, where the cos (C) is real and the sin (S) is complex, and the vectors CX and CY are complex. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublascrot(h, n, x, incx, y, incy, sc, cs)
  type(cublasHandle) :: h
  integer :: n
  real(4) :: sc
  complex(4) :: cs
  complex(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

2.8.3.6. cublasCscal

CSCAL scales a vector by a constant. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublascscal(h, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: n
  complex(4) :: a
  complex(4), device, dimension(*) :: x
  integer :: incx
```

2.8.3.7. cublasCsscal

CSSCAL scales a complex vector by a real constant. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublascsscal(h, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: n
  real(4) :: a
  complex(4), device, dimension(*) :: x
  integer :: incx
```

2.8.3.8. cublasCswap

CSWAP interchanges two vectors. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublascswap(h, n, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  complex(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

2.8.3.9. cublasIcamax

ICAMAX finds the index of the first element having maximum absolute value. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasicamax(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  complex(4), device, dimension(*) :: x
  integer :: incx
  integer :: res
```

2.8.3.10. cublasIcamin

ICAMIN finds the index of the first element having minimum absolute value. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasicamin(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  complex(4), device, dimension(*) :: x
  integer :: incx
  integer :: res
```

2.8.3.11. cublasScasum

SCASUM takes the sum of the absolute values of a complex vector and returns a single precision result. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasscasum(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  complex(4), device, dimension(*) :: x
  integer :: incx
  real(4) :: res
```

2.8.3.12. cublasScnrm2

SCNRM2 returns the euclidean norm of a vector via the function name, so that SCNRM2 := sqrt(x**H*x) Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasscnrm2(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  complex(4), device, dimension(*) :: x
  integer :: incx
  real(4) :: res
```

2.8.3.13. cublasCgbmv

CGBMV performs one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A**T*x + beta*y, or y := alpha*A**H*x + beta*y, where alpha and beta are scalars, x and y are vectors and A is an m by n band matrix, with kl sub-diagonals and ku super-

diagonals. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublascgbmv(h, t, m, n, kl, ku, alpha, a, lda,
           x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: m, n, kl, ku, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x, y
  complex(4) :: alpha, beta
```

2.8.3.14. cublasCgemv

CGEMV performs one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A**T*x + beta*y, or y := alpha*A**H*x + beta*y, where alpha and beta are scalars, x and y are vectors and A is an m by n matrix. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublascgemv(h, t, m, n, alpha, a, lda,
    x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: m, n, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x, y
  complex(4) :: alpha, beta
```

2.8.3.15. cublasCgerc

CGERC performs the rank 1 operation A := alpha*x*y**H + A, where alpha is a scalar, x is an m element vector, y is an n element vector and A is an m by n matrix. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublascgerc(h, m, n, alpha, x, incx, y, incy,
           a, lda)
  type(cublasHandle) :: h
  integer :: m, n, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x, y
  complex(4) :: alpha
```

2.8.3.16. cublasCgeru

CGERU performs the rank 1 operation A := alpha*x*y**T + A, where alpha is a scalar, x is an m element vector, y is an n element vector and A is an m by n matrix. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublascgeru(h, m, n, alpha, x, incx, y, incy,
           a, lda)
  type(cublasHandle) :: h
  integer :: m, n, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x, y
  complex(4) :: alpha
```

2.8.3.17. cublasChbmv

CHBMV performs the matrix-vector operation y := alpha*A*x + beta*y, where alpha and beta are scalars, x and y are n element vectors and A is an n by n hermitian band matrix, with k super-diagonals. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublaschbmv(h, t, n, k, alpha, a, lda,
    x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: k, n, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x, y
  complex(4) :: alpha, beta
```

2.8.3.18. cublasChemv

CHEMV performs the matrix-vector operation y := alpha*A*x + beta*y, where alpha and beta are scalars, x and y are n element vectors and A is an n by n hermitian matrix. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublaschemv(h, t, n, alpha, a, lda,
 x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x, y
  complex(4) :: alpha, beta
```

2.8.3.19. cublasCher

CHER performs the hermitian rank 1 operation A := alpha*x*x**H + A, where alpha is a real scalar, x is an n element vector and A is an n by n hermitian matrix. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublascher(h, t, n, alpha, x, incx, a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, lda
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x
  real(4) :: alpha
```

2.8.3.20. cublasCher2

CHER2 performs the hermitian rank 2 operation A := alpha*x*y**H + conjg(alpha)*y*x**H + A, where alpha is a scalar, x and y are n element vectors and A is an n by n hermitian matrix. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublascher2(h, t, n, alpha, x, incx, y, incy,
                      a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy, lda
  complex(4), device, dimension(lda, *) :: a
```

```
  complex(4), device, dimension(*) :: x, y
  complex(4) :: alpha
```

2.8.3.21. cublasChpmv

CHPMV performs the matrix-vector operation y := alpha*A*x + beta*y, where alpha and beta are scalars, x and y are n element vectors and A is an n by n hermitian matrix, supplied in packed form. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublaschpmv(h, t, n, alpha, a, x, incx,
     beta, y, incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy
  complex(4), device, dimension(*) :: a, x, y
  complex(4) :: alpha, beta
```

2.8.3.22. cublasChpr

CHPR performs the hermitian rank 1 operation A := alpha*x*x**H + A, where alpha is a real scalar, x is an n element vector and A is an n by n hermitian matrix, supplied in packed form. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublaschpr(h, t, n, alpha, x, incx, a)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx
  complex(4), device, dimension(*) :: a, x
  real(4) :: alpha
```

2.8.3.23. cublasChpr2

CHPR2 performs the hermitian rank 2 operation A := alpha*x*y**H + conjg(alpha)*y*x**H + A, where alpha is a scalar, x and y are n element vectors and A is an n by n hermitian matrix, supplied in packed form. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublaschpr2(h, t, n, alpha, x, incx,
   y, incy, a)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy
  complex(4), device, dimension(*) :: a, x, y
  complex(4) :: alpha
```

2.8.3.24. cublasCtbmv

CTBMV performs one of the matrix-vector operations x := A*x, or x := A**T*x, or x := A**H*x, where x is an n element vector and A is an n by n unit, or non-unit, upper or lower triangular band matrix, with (k + 1) diagonals. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasctbmv(h, u, t, d, n, k, a, lda,
    x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, k, incx, lda
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x
```

2.8.3.25. cublasCtbsv

CTBSV solves one of the systems of equations A*x = b, or A**T*x = b, or A**H*x = b, where b and x are n element vectors and A is an n by 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 routine. Such tests must be performed before calling this routine. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasctbsv(h, u, t, d, n, k, a, lda,
    x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, k, incx, lda
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x
```

2.8.3.26. cublasCtpmv

CTPMV performs one of the matrix-vector operations x := A*x, or x := A**T*x, or x := A**H*x, where x is an n element vector and A is an n by n unit, or non-unit, upper or lower triangular matrix, supplied in packed form. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasctpmv(h, u, t, d, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx
  complex(4), device, dimension(*) :: a, x
```

2.8.3.27. cublasCtpsv

CTPSV solves one of the systems of equations A*x = b, or A**T*x = b, or A**H*x = b, where b and x are n element vectors and A is an n by n unit, or non-unit, upper or lower triangular matrix, supplied in packed form. No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasctpsv(h, u, t, d, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx
  complex(4), device, dimension(*) :: a, x
```

2.8.3.28. cublasCtrmv

CTRMV performs one of the matrix-vector operations x := A*x, or x := A**T*x, or x := A**H*x, where x is an n element vector and A is an n by n unit, or non-unit, upper or lower triangular matrix. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasctrmv(h, u, t, d, n, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx, lda
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x
```

2.8.3.29. cublasCtrsv

CTRSV solves one of the systems of equations A*x = b, or A**T*x = b, or A**H*x = b, where b and x are n element vectors and A is an n by n unit, or non-unit, upper or lower triangular matrix. No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasctrsv(h, u, t, d, n, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx, lda
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x
```

2.8.3.30. cublasCgemm

CGEMM performs one of the matrix-matrix operations C := alpha*op(A)*op(B) + beta*C, where op(X) is one of op(X) = X or op(X) = X**T or op(X) = X**H, alpha and beta are scalars, and A, B and C are matrices, with op(A) an m by k matrix, op(B) a k by n matrix and C an m by n matrix. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublascgemm(h, transa, transb, m, n,
               k, alpha, a, lda, b, ldb, beta, c, ldc)
  type(cublasHandle) :: h
  integer :: transa, transb
  integer :: m, n, k, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device, dimension(ldc, *) :: c
  complex(4) :: alpha, beta
```

2.8.3.31. cublasChemm

CHEMM 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 scalars, A is an hermitian matrix and B and C are m by n matrices. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublaschemm(h, side, uplo, m, n,
          alpha, a, lda, b, ldb, beta, c, ldc)
  type(cublasHandle) :: h
  integer :: side, uplo
  integer :: m, n, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device, dimension(ldc, *) :: c
  complex(4) :: alpha, beta
```

2.8.3.32. cublasCherk

CHERK performs one of the hermitian rank k operations C := alpha*A*A**H + beta*C, or C := alpha*A**H*A + beta*C, where alpha and beta are real scalars, C is an n by n hermitian matrix and A is an n by k matrix in the first case and a k by n matrix in the

second case. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublascherk(h, uplo, trans, n, k,
                    alpha, a, lda, beta, c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldc, *) :: c
  real(4) :: alpha, beta
```

2.8.3.33. cublasCher2k

CHER2K performs one of the hermitian rank 2k operations C := alpha*A*B**H + conjg(alpha)*B*A**H + beta*C, or C := alpha*A**H*B + conjg(alpha)*B**H*A + beta*C, where alpha and beta are scalars with beta real, C is an n by n hermitian matrix and A and B are n by k matrices in the first case and k by n matrices in the second case. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublascher2k(h, uplo, trans, n, k,
               alpha, a, lda, b, ldb, beta, c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device, dimension(ldc, *) :: c
  complex(4) :: alpha
  real(4) :: beta
```

2.8.3.34. cublasCsymm

CSYMM 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 scalars, A is a symmetric matrix and B and C are m by n matrices. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublascsymm(h, side, uplo, m, n,
          alpha, a, lda, b, ldb, beta, c, ldc)
  type(cublasHandle) :: h
  integer :: side, uplo
  integer :: m, n, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device, dimension(ldc, *) :: c
  complex(4) :: alpha, beta
```

2.8.3.35. cublasCsyrk

CSYRK performs one of the symmetric rank k operations C := alpha*A*A**T + beta*C, or C := alpha*A**T*A + beta*C, where alpha and beta are scalars, C is an n by n symmetric matrix and A is an n by k matrix in the first case and a k by n matrix in the second case. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublascsyrk(h, uplo, trans, n, k,
                    alpha, a, lda, beta, c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
```

```
  integer :: n, k, lda, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldc, *) :: c
  complex(4) :: alpha, beta
```

2.8.3.36. cublasCsyr2k

CSYR2K 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 scalars, C is an n by n symmetric matrix and A and B are n by k matrices in the first case and k by n matrices in the second case. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublascsyr2k(h, uplo, trans, n, k,
              alpha, a, lda, b, ldb, beta, c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device, dimension(ldc, *) :: c
  complex(4) :: alpha, beta
```

2.8.3.37. cublasCtrmm

CTRMM performs one of the matrix-matrix operations B := alpha*op(A)*B, or B := alpha*B*op(A) where alpha is a scalar, B is an m by n matrix, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A**T or op(A) = A**H. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasctrmm(h, side, uplo, transa,
        diag, m, n, alpha, a, lda, b, ldb, c, ldc)
  type(cublasHandle) :: h
  integer :: side, uplo, transa, diag
  integer :: m, n, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device, dimension(ldc, *) :: c
  complex(4) :: alpha
```

2.8.3.38. cublasCtrsm

CTRSM solves one of the matrix equations op(A)*X = alpha*B, or X*op(A) = alpha*B, where alpha is a scalar, X and B are m by n matrices, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A**T or op(A) = A**H. The matrix X is overwritten on B. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasctrsm(h, side, uplo, transa,
               diag, m, n, alpha, a, lda, b, ldb)
  type(cublasHandle) :: h
  integer :: side, uplo, transa, diag
  integer :: m, n, lda, ldb
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4) :: alpha
```

2.8.3.39. cublasCgemmBatched

CGEMM performs one of the matrix-matrix operations C := alpha*op(A)*op(B) + beta*C, where op(X) is one of op(X) = X or op(X) = X**T or op(X) = X**H, alpha and beta are scalars, and A, B and C are matrices, with op(A) an m by k matrix, op(B) a k by n matrix and C an m by n matrix. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasCgemmBatched(h, transa, transb,                m, n,
k, alpha, Aarray, lda, Barray, ldb, beta, Carray, ldc, batchCount)
  type(cublasHandle) :: h
  integer :: transa
  integer :: transb
  integer :: m, n, k
  complex(4) :: alpha
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: Barray(*)
  integer :: ldb
  complex(4) :: beta
  type(c_devptr), device :: Carray(*)
  integer :: ldc
  integer :: batchCount
```

2.8.3.40. cublasCgetrfBatched

CGETRF computes an LU factorization of a general M-by-N matrix A using partial pivoting with row interchanges. The factorization has the form A = P * L * U where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if m > n), and U is upper triangular (upper trapezoidal if m < n). This is the right-looking Level 3 BLAS version of the algorithm. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasCgetrfBatched(h, n, Aarray, lda,
                                ipvt, info, batchCount)
  type(cublasHandle) :: h
  integer :: n
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  integer, device :: ipvt(*)
  integer, device :: info(*)
  integer :: batchCount
```

2.8.3.41. cublasCgetriBatched

CGETRI computes the inverse of a matrix using the LU factorization computed by CGETRF. This method inverts U and then computes inv(A) by solving the system inv(A)*L = inv(U) for inv(A). Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasCgetriBatched(h, n, Aarray, lda,
                   ipvt, Carray, ldc, info, batchCount)
  type(cublasHandle) :: h
  integer :: n
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  integer, device :: ipvt(*)
  type(c_devptr), device :: Carray(*)
  integer :: ldc
  integer, device :: info(*)
```

```
  integer :: batchCount
```

2.8.3.42. cublasCgetrsBatched

CGETRS solves a system of linear equations A * X = B, A**T * X = B, or A**H * X = B with a general N-by-N matrix A using the LU factorization computed by CGETRF. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasCgetrsBatched(h, trans, n, nrhs,
                A, lda, ipvt, B, ldb, info, batchCount)
  type(cublasHandle) :: h
  integer :: trans
  integer :: n, nrhs
  type(c_devptr), device :: A(*)
  integer :: lda
  integer, device :: ipvt(*)
  type(c_devptr), device :: B(*)
  integer :: ldb
  integer, device :: info(*)
  integer :: batchCount
```

2.8.3.43. cublasCtrsmBatched

CTRSM solves one of the matrix equations op(A)*X = alpha*B, or X*op(A) = alpha*B, where alpha is a scalar, X and B are m by n matrices, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A**T or op(A) = A**H. The matrix X is overwritten on B. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasCtrsmBatched(h, side, uplo,
    trans, diag, m, n, alpha, A, lda, B, ldb, batchCount)
  type(cublasHandle) :: h
  integer :: side
  integer :: uplo
  integer :: trans
  integer :: diag
  integer :: m, n
  complex(4) :: alpha
  type(c_devptr), device :: A(*)
  integer :: lda
  type(c_devptr), device :: B(*)
  integer :: ldb
  integer :: batchCount
```

2.8.3.44. cublasCmatinvBatched

cublasCmatinvBatched is a short cut of cublasCgetrfBatched plus cublasCgetriBatched. However it only works if n is less than 32. If not, the user has to go through cublasCgetrfBatched and cublasCgetriBatched. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasCmatinvBatched(h, n, A, lda,
               Ainv, lda_inv, info, batchCount)
  type(cublasHandle) :: h
  integer :: n
  type(c_devptr), device :: A(*)
  integer :: lda
  type(c_devptr), device :: Ainv(*)
  integer :: lda_inv
  integer, device :: info(*)
  integer :: batchCount
```

2.8.3.45. cublasCgeqrfBatched

CGEQRF computes a QR factorization of a complex M-by-N matrix A: A = Q * R. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasCgeqrfBatched(h, m, n,
     A, lda, Tau, info, batchCount)
  type(cublasHandle) :: h
  integer :: m, n
  type(c_devptr), device :: A(*)
  integer :: lda
  type(c_devptr), device :: Tau(*)
  integer, device :: info(*)
  integer :: batchCount
```

2.8.3.46. cublasCgelsBatched

CGELS solves overdetermined or underdetermined complex linear systems involving an M-by-N matrix A, or its conjugate-transpose, using a QR or LQ factorization of A. It is assumed that A has full rank. The following options are provided: 1. If TRANS = 'N' and m >= n: find the least squares solution of an overdetermined system, i.e., solve the least squares problem minimize || B - A*X ||. 2. If TRANS = 'N' and m < n: find the minimum norm solution of an underdetermined system A * X = B. 3. If TRANS = 'C' and m >= n: find the minimum norm solution of an undetermined system A**H * X = B. 4. If TRANS = 'C' and m < n: find the least squares solution of an overdetermined system, i.e., solve the least squares problem minimize || B - A**H * X ||. Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the M-by-NRHS right hand side matrix B and the N-by-NRHS solution matrix X. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasCgelsBatched(h, trans, m, n,
 nrhs, A, lda, C, ldc, info, dinfo, batchCount)
  type(cublasHandle) :: h
  integer :: trans
  integer :: m, n, nrhs
  type(c_devptr), device :: A(*)
  integer :: lda
  type(c_devptr), device :: C(*)
  integer :: ldc
  integer, device :: info(*)
  integer, device :: dinfo(*)
  integer :: batchCount
```

2.8.4. Double Precision Functions and Subroutines

This section contains the cuBLAS interfaces to the device-side double precision BLAS and cuBLAS functions and subroutines.

2.8.4.1. cublasIdamax

IDAMAX finds the the index of the first element having maximum absolute value. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasidamax(h, n, x, incx, res)
```

```
  type(cublasHandle) :: h
  integer :: n
  real(8), device, dimension(*) :: x
  integer :: incx
  integer :: res
```

2.8.4.2. cublasIdamin

IDAMIN finds the index of the first element having minimum absolute value. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasidamin(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  real(8), device, dimension(*) :: x
  integer :: incx
  integer :: res
```

2.8.4.3. cublasDasum

DASUM takes the sum of the absolute values. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasdasum(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  real(8), device, dimension(*) :: x
  integer :: incx
  real(8) :: res
```

2.8.4.4. cublasDaxpy

DAXPY constant times a vector plus a vector. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasdaxpy(h, n, a, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  real(8) :: a
  real(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

2.8.4.5. cublasDcopy

DCOPY copies a vector, x, to a vector, y. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasdcopy(h, n, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  real(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

2.8.4.6. cublasDdot

DDOT forms the dot product of two vectors. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasddot(h, n, x, incx, y, incy, res)
  type(cublasHandle) :: h
  integer :: n
  real(8), device, dimension(*) :: x, y
```

```
  integer :: incx, incy
  real(8) :: res
```

2.8.4.7. cublasDnrm2

DNRM2 returns the euclidean norm of a vector via the function name, so that DNRM2 := sqrt(x'*x) Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasdnrm2(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  real(8), device, dimension(*) :: x
  integer :: incx
  real(8) :: res
```

2.8.4.8. cublasDrot

DROT applies a plane rotation. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasdrot(h, n, x, incx, y, incy, dc, ds)
  type(cublasHandle) :: h
  integer :: n
  real(8) :: dc, ds
  real(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

2.8.4.9. cublasDrotg

DROTG constructs a Givens plane rotation. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasdrotg(h, sa, sb, sc, ss)
  type(cublasHandle) :: h
  real(8) :: sa, sb, sc, ss
```

2.8.4.10. cublasDrotm

DROTM applies the modified Givens transformation, H, to the 2 by N matrix (DX**T) , where **T indicates transpose. The elements of DX are in (DX**T) DX(LX+I*INCX), I = 0 to N-1, where LX = 1 if INCX .GE. 0, ELSE LX = (-INCX)*N, and similarly for DY using LY and INCY. With DPARAM(1)=DFLAG, H has one of the following forms.. DFLAG=-1.D0 DFLAG=0.D0 DFLAG=1.D0 DFLAG=-2.D0 (DH11 DH12) (1.D0 DH12) (DH11 1.D0) (1.D0 0.D0) H=( ) ( ) ( ) ( ) (DH21 DH22), (DH21 1.D0), (-1.D0 DH22), (0.D0 1.D0). See DROTMG for a description of data storage in DPARAM. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasdrotm(h, n, x, incx, y, incy, param)
  type(cublasHandle) :: h
  integer :: n
  real(8) :: param(*)
  real(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

2.8.4.11. cublasDrotmg

DROTMG constructs the modified Givens transformation matrix H which zeros the second component of the 2-vector (SQRT(DD1)*DX1,SQRT(DD2)*DY2)**T. With DPARAM(1)=DFLAG, H has one of the following forms..

```
      DFLAG=-1.D0     DFLAG=0.D0        DFLAG=1.D0     DFLAG=-2.D0

        (DH11  DH12)    (1.D0  DH12)    (DH11  1.D0)    (1.D0  0.D0)
      H=(          )    (         )    (          )    (         )
        (DH21  DH22),   (DH21  1.D0),   (-1.D0 DH22),   (0.D0  1.D0).
```

Locations 2-4 of DPARAM contain DH11, DH21, DH12, and DH22 respectively. (Values of 1.D0, -1.D0, of 0.D0 implied by the value of DPARAM(1) are not stored in DPARAM.) Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasdrotmg(h, d1, d2, x1, y1, param)
  type(cublasHandle) :: h
  real(8) :: d1, d2, x1, y1, param(*)
```

2.8.4.12. cublasDscal

DSCAL scales a vector by a constant. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasdscal(h, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: n
  real(8) :: a
  real(8), device, dimension(*) :: x
  integer :: incx
```

2.8.4.13. cublasDswap

interchanges two vectors. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasdswap(h, n, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  real(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

2.8.4.14. cublasDgbmv

DGBMV performs one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A**T*x + beta*y, where alpha and beta are scalars, x and y are vectors and A is an m by n band matrix, with kl sub-diagonals and ku super-diagonals. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasdgbmv(h, t, m, n, kl, ku, alpha, a, lda,
            x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: m, n, kl, ku, lda, incx, incy
  real(8), device, dimension(lda, *) :: a
```

```
  real(8), device, dimension(*) :: x, y
  real(8) :: alpha, beta
```

2.8.4.15. cublasDgemv

DGEMV performs one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A**T*x + beta*y, where alpha and beta are scalars, x and y are vectors and A is an m by n matrix. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasdgemv(h, t, m, n, alpha, a, lda,
    x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: m, n, lda, incx, incy
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x, y
  real(8) :: alpha, beta
```

2.8.4.16. cublasDger

DGER performs the rank 1 operation A := alpha*x*y**T + A, where alpha is a scalar, x is an m element vector, y is an n element vector and A is an m by n matrix. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasdger(h, m, n, alpha, x, incx, y, incy,
          a, lda)
  type(cublasHandle) :: h
  integer :: m, n, lda, incx, incy
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x, y
  real(8) :: alpha
```

2.8.4.17. cublasDsbmv

DSBMV performs the matrix-vector operation y := alpha*A*x + beta*y, where alpha and beta are scalars, x and y are n element vectors and A is an n by n symmetric band matrix, with k super-diagonals. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasdsbmv(h, t, n, k, alpha, a, lda,
    x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: k, n, lda, incx, incy
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x, y
  real(8) :: alpha, beta
```

2.8.4.18. cublasDspmv

DSPMV performs the matrix-vector operation y := alpha*A*x + beta*y, where alpha and beta are scalars, x and y are n element vectors and A is an n by n symmetric matrix, supplied in packed form. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasdspmv(h, t, n, alpha, a, x, incx,
     beta, y, incy)
  type(cublasHandle) :: h
  integer :: t
```

```
  integer :: n, incx, incy
  real(8), device, dimension(*) :: a, x, y
  real(8) :: alpha, beta
```

2.8.4.19. cublasDspr

DSPR performs the symmetric rank 1 operation A := alpha*x*x**T + A, where alpha is a real scalar, x is an n element vector and A is an n by n symmetric matrix, supplied in packed form. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasdspr(h, t, n, alpha, x, incx, a)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx
  real(8), device, dimension(*) :: a, x
  real(8) :: alpha
```

2.8.4.20. cublasDspr2

DSPR2 performs the symmetric rank 2 operation A := alpha*x*y**T + alpha*y*x**T + A, where alpha is a scalar, x and y are n element vectors and A is an n by n symmetric matrix, supplied in packed form. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasdspr2(h, t, n, alpha, x, incx,
   y, incy, a)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy
  real(8), device, dimension(*) :: a, x, y
  real(8) :: alpha
```

2.8.4.21. cublasDsymv

DSYMV performs the matrix-vector operation y := alpha*A*x + beta*y, where alpha and beta are scalars, x and y are n element vectors and A is an n by n symmetric matrix. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasdsymv(h, t, n, alpha, a, lda,
 x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, lda, incx, incy
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x, y
  real(8) :: alpha, beta
```

2.8.4.22. cublasDsyr

DSYR performs the symmetric rank 1 operation A := alpha*x*x**T + A, where alpha is a real scalar, x is an n element vector and A is an n by n symmetric matrix. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasdsyr(h, t, n, alpha, x, incx, a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, lda
  real(8), device, dimension(lda, *) :: a
```

```
  real(8), device, dimension(*) :: x
  real(8) :: alpha
```

2.8.4.23. cublasDsyr2

DSYR2 performs the symmetric rank 2 operation A := alpha*x*y**T + alpha*y*x**T + A, where alpha is a scalar, x and y are n element vectors and A is an n by n symmetric matrix. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasdsyr2(h, t, n, alpha, x, incx,
  y, incy, a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy, lda
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x, y
  real(8) :: alpha
```

2.8.4.24. cublasDtbmv

DTBMV performs one of the matrix-vector operations x := A*x, or x := A**T*x, where x is an n element vector and A is an n by n unit, or non-unit, upper or lower triangular band matrix, with (k + 1) diagonals. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasdtbmv(h, u, t, d, n, k, a, lda,
    x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, k, incx, lda
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x
```

2.8.4.25. cublasDtbsv

DTBSV solves one of the systems of equations A*x = b, or A**T*x = b, where b and x are n element vectors and A is an n by 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 routine. Such tests must be performed before calling this routine. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasdtbsv(h, u, t, d, n, k, a, lda,
    x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, k, incx, lda
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x
```

2.8.4.26. cublasDtpmv

DTPMV performs one of the matrix-vector operations x := A*x, or x := A**T*x, where x is an n element vector and A is an n by n unit, or non-unit, upper or lower triangular matrix, supplied in packed form. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasdtpmv(h, u, t, d, n, a, x, incx)
  type(cublasHandle) :: h
```

```
  integer :: u, t, d
  integer :: n, incx
  real(8), device, dimension(*) :: a, x
```

2.8.4.27. cublasDtpsv

DTPSV solves one of the systems of equations A*x = b, or A**T*x = b, where b and x are n element vectors and A is an n by n unit, or non-unit, upper or lower triangular matrix, supplied in packed form. No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasdtpsv(h, u, t, d, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx
  real(8), device, dimension(*) :: a, x
```

2.8.4.28. cublasDtrmv

DTRMV performs one of the matrix-vector operations x := A*x, or x := A**T*x, where x is an n element vector and A is an n by n unit, or non-unit, upper or lower triangular matrix. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasdtrmv(h, u, t, d, n, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx, lda
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x
```

2.8.4.29. cublasDtrsv

DTRSV solves one of the systems of equations A*x = b, or A**T*x = b, where b and x are n element vectors and A is an n by n unit, or non-unit, upper or lower triangular matrix. No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasdtrsv(h, u, t, d, n, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx, lda
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x
```

2.8.4.30. cublasDgemm

DGEMM performs one of the matrix-matrix operations C := alpha*op(A)*op(B) + beta*C, where op(X) is one of op(X) = X or op(X) = X**T, alpha and beta are scalars, and A, B and C are matrices, with op(A) an m by k matrix, op(B) a k by n matrix and C an m by n matrix. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasdgemm(h, transa, transb, m, n,
                k, alpha, a, lda, b, ldb, beta, c, ldc)
  type(cublasHandle) :: h
  integer :: transa, transb
  integer :: m, n, k, lda, ldb, ldc
```

```
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(ldb, *) :: b
  real(8), device, dimension(ldc, *) :: c
  real(8) :: alpha, beta
```

2.8.4.31. cublasDsymm

DSYMM 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 scalars, A is a symmetric matrix and B and C are m by n matrices. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasdsymm(h, side, uplo, m, n,
          alpha, a, lda, b, ldb, beta, c, ldc)
  type(cublasHandle) :: h
  integer :: side, uplo
  integer :: m, n, lda, ldb, ldc
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(ldb, *) :: b
  real(8), device, dimension(ldc, *) :: c
  real(8) :: alpha, beta
```

2.8.4.32. cublasDsyrk

DSYRK performs one of the symmetric rank k operations C := alpha*A*A**T + beta*C, or C := alpha*A**T*A + beta*C, where alpha and beta are scalars, C is an n by n symmetric matrix and A is an n by k matrix in the first case and a k by n matrix in the second case. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasdsyrk(h, uplo, trans, n, k,
                    alpha, a, lda, beta, c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldc
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(ldc, *) :: c
  real(8) :: alpha, beta
```

2.8.4.33. cublasDsyr2k

DSYR2K 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 scalars, C is an n by n symmetric matrix and A and B are n by k matrices in the first case and k by n matrices in the second case. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasdsyr2k(h, uplo, trans, n, k,
              alpha, a, lda, b, ldb, beta, c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(ldb, *) :: b
  real(8), device, dimension(ldc, *) :: c
  real(8) :: alpha, beta
```

2.8.4.34. cublasDtrmm

DTRMM performs one of the matrix-matrix operations B := alpha*op(A)*B, or B := alpha*B*op(A), where alpha is a scalar, B is an m by n matrix, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A**T. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasdtrmm(h, side, uplo, transa,
        diag, m, n, alpha, a, lda, b, ldb, c, ldc)
  type(cublasHandle) :: h
  integer :: side, uplo, transa, diag
  integer :: m, n, lda, ldb, ldc
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(ldb, *) :: b
  real(8), device, dimension(ldc, *) :: c
  real(8) :: alpha
```

2.8.4.35. cublasDtrsm

DTRSM solves one of the matrix equations op(A)*X = alpha*B, or X*op(A) = alpha*B, where alpha is a scalar, X and B are m by n matrices, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A**T. The matrix X is overwritten on B. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasdtrsm(h, side, uplo, transa,
                diag, m, n, alpha, a, lda, b, ldb)
  type(cublasHandle) :: h
  integer :: side, uplo, transa, diag
  integer :: m, n, lda, ldb
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(ldb, *) :: b
  real(8) :: alpha
```

2.8.4.36. cublasDgemmBatched

DGEMM performs one of the matrix-matrix operations C := alpha*op(A)*op(B) + beta*C, where op(X) is one of op(X) = X or op(X) = X**T, alpha and beta are scalars, and A, B and C are matrices, with op(A) an m by k matrix, op(B) a k by n matrix and C an m by n matrix. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasDgemmBatched(h, transa, transb,                 m, n,
k, alpha, Aarray, lda, Barray, ldb, beta, Carray, ldc, batchCount)
  type(cublasHandle) :: h
  integer :: transa
  integer :: transb
  integer :: m, n, k
  real(8) :: alpha
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: Barray(*)
  integer :: ldb
  real(8) :: beta
  type(c_devptr), device :: Carray(*)
  integer :: ldc
  integer :: batchCount
```

2.8.4.37. cublasDgetrfBatched

DGETRF computes an LU factorization of a general M-by-N matrix A using partial pivoting with row interchanges. The factorization has the form A = P * L * U where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if m > n), and U is upper triangular (upper trapezoidal if m < n). This is the right-looking Level 3 BLAS version of the algorithm. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasDgetrfBatched(h, n, Aarray, lda,
                                  ipvt, info, batchCount)
  type(cublasHandle) :: h
  integer :: n
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  integer, device :: ipvt(*)
  integer, device :: info(*)
  integer :: batchCount
```

2.8.4.38. cublasDgetriBatched

DGETRI computes the inverse of a matrix using the LU factorization computed by DGETRF. This method inverts U and then computes inv(A) by solving the system inv(A)*L = inv(U) for inv(A). Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasDgetriBatched(h, n, Aarray, lda,
                   ipvt, Carray, ldc, info, batchCount)
  type(cublasHandle) :: h
  integer :: n
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  integer, device :: ipvt(*)
  type(c_devptr), device :: Carray(*)
  integer :: ldc
  integer, device :: info(*)
  integer :: batchCount
```

2.8.4.39. cublasDgetrsBatched

DGETRS solves a system of linear equations A * X = B or A**T * X = B with a general N-by-N matrix A using the LU factorization computed by DGETRF. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasDgetrsBatched(h, trans, n, nrhs,
                A, lda, ipvt, B, ldb, info, batchCount)
  type(cublasHandle) :: h
  integer :: trans
  integer :: n, nrhs
  type(c_devptr), device :: A(*)
  integer :: lda
  integer, device :: ipvt(*)
  type(c_devptr), device :: B(*)
  integer :: ldb
  integer, device :: info(*)
  integer :: batchCount
```

2.8.4.40. cublasDtrsmBatched

DTRSM solves one of the matrix equations op(A)*X = alpha*B, or X*op(A) = alpha*B, where alpha is a scalar, X and B are m by n matrices, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A**T. The matrix X is overwritten on B. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasDtrsmBatched(h, side, uplo,
    trans, diag, m, n, alpha, A, lda, B, ldb, batchCount)
  type(cublasHandle) :: h
  integer :: side
  integer :: uplo
  integer :: trans
  integer :: diag
  integer :: m, n
  real(8) :: alpha
  type(c_devptr), device :: A(*)
  integer :: lda
  type(c_devptr), device :: B(*)
  integer :: ldb
  integer :: batchCount
```

2.8.4.41. cublasDmatinvBatched

cublasDmatinvBatched is a short cut of cublasDgetrfBatched plus cublasDgetriBatched. However it only works if n is less than 32. If not, the user has to go through cublasDgetrfBatched and cublasDgetriBatched. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasDmatinvBatched(h, n, A, lda,
               Ainv, lda_inv, info, batchCount)
  type(cublasHandle) :: h
  integer :: n
  type(c_devptr), device :: A(*)
  integer :: lda
  type(c_devptr), device :: Ainv(*)
  integer :: lda_inv
  integer, device :: info(*)
  integer :: batchCount
```

2.8.4.42. cublasDgeqrfBatched

DGEQRF computes a QR factorization of a real M-by-N matrix A: A = Q * R. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasDgeqrfBatched(h, m, n,
     A, lda, Tau, info, batchCount)
  type(cublasHandle) :: h
  integer :: m, n
  type(c_devptr), device :: A(*)
  integer :: lda
  type(c_devptr), device :: Tau(*)
  integer, device :: info(*)
  integer :: batchCount
```

2.8.4.43. cublasDgelsBatched

DGELS solves overdetermined or underdetermined real linear systems involving an M-by-N matrix A, or its transpose, using a QR or LQ factorization of A. It is assumed that A has full rank. The following options are provided: 1. If TRANS = 'N' and m >= n: find the least squares solution of an overdetermined system, i.e., solve the least squares problem minimize || B - A*X ||. 2. If TRANS = 'N' and m < n: find the minimum norm solution of an underdetermined system A * X = B. 3. If TRANS = 'T' and m >= n: find the minimum norm solution of an undetermined system A**T * X = B. 4. If TRANS = 'T' and m < n: find the least squares solution of an overdetermined system, i.e., solve the least squares problem minimize || B - A**T * X ||. Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the M-by-NRHS right hand side matrix B and the N-by-NRHS solution matrix X. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasDgelsBatched(h, trans, m, n,
 nrhs, A, lda, C, ldc, info, dinfo, batchCount)
  type(cublasHandle) :: h
  integer :: trans
  integer :: m, n, nrhs
  type(c_devptr), device :: A(*)
  integer :: lda
  type(c_devptr), device :: C(*)
  integer :: ldc
  integer, device :: info(*)
  integer, device :: dinfo(*)
  integer :: batchCount
```

2.8.5. Double Precision Complex Functions and Subroutines

This section contains the cuBLAS interfaces to the device-side double precision complex BLAS and cuBLAS functions and subroutines.

2.8.5.1. cublasDzasum

DZASUM takes the sum of the absolute values. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasdzasum(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  complex(8), device, dimension(*) :: x
  integer :: incx
  real(8) :: res
```

2.8.5.2. cublasDznrm2

DZNRM2 returns the euclidean norm of a vector via the function name, so that DZNRM2 := sqrt(x**H*x) Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasdznrm2(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
```

```
  complex(8), device, dimension(*) :: x
  integer :: incx
  real(8) :: res
```

2.8.5.3. cublasIzamax

IZAMAX finds the index of the first element having maximum absolute value. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasizamax(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  complex(8), device, dimension(*) :: x
  integer :: incx
  integer :: res
```

2.8.5.4. cublasIzamin

IZAMIN finds the index of the first element having minimum absolute value. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasizamin(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  complex(8), device, dimension(*) :: x
  integer :: incx
  integer :: res
```

2.8.5.5. cublasZaxpy

ZAXPY constant times a vector plus a vector. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublaszaxpy(h, n, a, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  complex(8) :: a
  complex(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

2.8.5.6. cublasZcopy

ZCOPY copies a vector, x, to a vector, y. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublaszcopy(h, n, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  complex(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

2.8.5.7. cublasZdotc

ZDOTC forms the dot product of a vector. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublaszdotc(h, n, x, incx, y, incy, res)
  type(cublasHandle) :: h
  integer :: n
  complex(8), device, dimension(*) :: x, y
```

```
  integer :: incx, incy
  complex(8) :: res
```

2.8.5.8. cublasZdotu

ZDOTU forms the dot product of two vectors. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublaszdotu(h, n, x, incx, y, incy, res)
  type(cublasHandle) :: h
  integer :: n
  complex(8), device, dimension(*) :: x, y
  integer :: incx, incy
  complex(8) :: res
```

2.8.5.9. cublasZdrot

Applies a plane rotation, where the cos and sin (c and s) are real and the vectors cx and cy are complex. jack dongarra, linpack, 3/11/78. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublaszdrot(h, n, x, incx, y, incy, sc, cs)
  type(cublasHandle) :: h
  integer :: n
  complex(8), device, dimension(*) :: x, y
  integer :: incx, incy
  real(8) :: sc, cs
```

2.8.5.10. cublasZrot

ZROT applies a plane rotation, where the cos (C) is real and the sin (S) is complex, and the vectors CX and CY are complex. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublaszrot(h, n, x, incx, y, incy, sc, cs)
  type(cublasHandle) :: h
  integer :: n
  complex(8), device, dimension(*) :: x, y
  integer :: incx, incy
  real(8) :: sc
  complex(4) :: cs
```

2.8.5.11. cublasZscal

ZSCAL scales a vector by a constant. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublaszscal(h, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: n
  complex(8) :: a
  complex(8), device, dimension(*) :: x
  integer :: incx
```

2.8.5.12. cublasZdscal

ZDSCAL scales a vector by a constant. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublaszdscal(h, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: n
```

```
  real(8) :: a
  complex(8), device, dimension(*) :: x
  integer :: incx
```

2.8.5.13. cublasZswap

ZSWAP interchanges two vectors. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublaszswap(h, n, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  complex(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

2.8.5.14. cublasZgbmv

ZGBMV performs one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A**T*x + beta*y, or y := alpha*A**H*x + beta*y, where alpha and beta are scalars, x and y are vectors and A is an m by n band matrix, with kl sub-diagonals and ku super-diagonals. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublaszgbmv(h, t, m, n, kl, ku, alpha, a, lda,
            x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: m, n, kl, ku, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
  complex(8) :: alpha, beta
```

2.8.5.15. cublasZgemv

ZGEMV performs one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A**T*x + beta*y, or y := alpha*A**H*x + beta*y, where alpha and beta are scalars, x and y are vectors and A is an m by n matrix. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublaszgemv(h, t, m, n, alpha, a, lda,
    x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: m, n, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
  complex(8) :: alpha, beta
```

2.8.5.16. cublasZgerc

ZGERC performs the rank 1 operation A := alpha*x*y**H + A, where alpha is a scalar, x is an m element vector, y is an n element vector and A is an m by n matrix. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublaszgerc(h, m, n, alpha, x, incx, y, incy,
           a, lda)
  type(cublasHandle) :: h
  integer :: m, n, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
```

```
  complex(8) :: alpha
```

2.8.5.17. cublasZgeru

ZGERU performs the rank 1 operation A := alpha*x*y**T + A, where alpha is a scalar, x is an m element vector, y is an n element vector and A is an m by n matrix. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublaszgeru(h, m, n, alpha, x, incx, y, incy,
           a, lda)
  type(cublasHandle) :: h
  integer :: m, n, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
  complex(8) :: alpha
```

2.8.5.18. cublasZhbmv

ZHBMV performs the matrix-vector operation y := alpha*A*x + beta*y, where alpha and beta are scalars, x and y are n element vectors and A is an n by n hermitian band matrix, with k super-diagonals. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublaszhbmv(h, t, n, k, alpha, a, lda,
    x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: k, n, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
  complex(8) :: alpha, beta
```

2.8.5.19. cublasZhemv

ZHEMV performs the matrix-vector operation y := alpha*A*x + beta*y, where alpha and beta are scalars, x and y are n element vectors and A is an n by n hermitian matrix. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublaszhemv(h, t, n, alpha, a, lda,
 x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
  complex(8) :: alpha, beta
```

2.8.5.20. cublasZher

ZHER performs the hermitian rank 1 operation A := alpha*x*x**H + A, where alpha is a real scalar, x is an n element vector and A is an n by n hermitian matrix. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublaszher(h, t, n, alpha, x, incx, a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, lda
  complex(8), device, dimension(lda, *) :: a
```

```
  complex(8), device, dimension(*) :: x
  real(8) :: alpha
```

2.8.5.21. cublasZher2

ZHER2 performs the hermitian rank 2 operation A := alpha*x*y**H + conjg(alpha)*y*x**H + A, where alpha is a scalar, x and y are n element vectors and A is an n by n hermitian matrix. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublaszher2(h, t, n, alpha, x, incx, y, incy,
                        a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy, lda
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
  complex(8) :: alpha
```

2.8.5.22. cublasZhpmv

ZHPMV performs the matrix-vector operation y := alpha*A*x + beta*y, where alpha and beta are scalars, x and y are n element vectors and A is an n by n hermitian matrix, supplied in packed form. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublaszhpmv(h, t, n, alpha, a, x, incx,
     beta, y, incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy
  complex(8), device, dimension(*) :: a, x, y
  complex(8) :: alpha, beta
```

2.8.5.23. cublasZhpr

ZHPR performs the hermitian rank 1 operation A := alpha*x*x**H + A, where alpha is a real scalar, x is an n element vector and A is an n by n hermitian matrix, supplied in packed form. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublaszhpr(h, t, n, alpha, x, incx, a)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx
  complex(8), device, dimension(*) :: a, x
  real(8) :: alpha
```

2.8.5.24. cublasZhpr2

ZHPR2 performs the hermitian rank 2 operation A := alpha*x*y**H + conjg(alpha)*y*x**H + A, where alpha is a scalar, x and y are n element vectors and A is an n by n hermitian matrix, supplied in packed form. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublaszhpr2(h, t, n, alpha, x, incx,
   y, incy, a)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy
  complex(8), device, dimension(*) :: a, x, y
```

```
  complex(8) :: alpha
```

2.8.5.25. cublasZtbmv

ZTBMV performs one of the matrix-vector operations x := A*x, or x := A**T*x, or x := A**H*x, where x is an n element vector and A is an n by n unit, or non-unit, upper or lower triangular band matrix, with (k + 1) diagonals. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasztbmv(h, u, t, d, n, k, a, lda,
    x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, k, incx, lda
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x
```

2.8.5.26. cublasZtbsv

ZTBSV solves one of the systems of equations A*x = b, or A**T*x = b, or A**H*x = b, where b and x are n element vectors and A is an n by 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 routine. Such tests must be performed before calling this routine. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasztbsv(h, u, t, d, n, k, a, lda,
    x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, k, incx, lda
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x
```

2.8.5.27. cublasZtpmv

ZTPMV performs one of the matrix-vector operations x := A*x, or x := A**T*x, or x := A**H*x, where x is an n element vector and A is an n by n unit, or non-unit, upper or lower triangular matrix, supplied in packed form. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasztpmv(h, u, t, d, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx
  complex(8), device, dimension(*) :: a, x
```

2.8.5.28. cublasZtpsv

ZTPSV solves one of the systems of equations A*x = b, or A**T*x = b, or A**H*x = b, where b and x are n element vectors and A is an n by n unit, or non-unit, upper or lower triangular matrix, supplied in packed form. No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasztpsv(h, u, t, d, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
```

```
  integer :: n, incx
  complex(8), device, dimension(*) :: a, x
```

2.8.5.29. cublasZtrmv

ZTRMV performs one of the matrix-vector operations x := A*x, or x := A**T*x, or x := A**H*x, where x is an n element vector and A is an n by n unit, or non-unit, upper or lower triangular matrix. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasztrmv(h, u, t, d, n, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx, lda
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x
```

2.8.5.30. cublasZtrsv

ZTRSV solves one of the systems of equations A*x = b, or A**T*x = b, or A**H*x = b, where b and x are n element vectors and A is an n by n unit, or non-unit, upper or lower triangular matrix. No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasztrsv(h, u, t, d, n, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx, lda
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x
```

2.8.5.31. cublasZgemm

ZGEMM performs one of the matrix-matrix operations C := alpha*op(A)*op(B) + beta*C, where op(X) is one of op(X) = X or op(X) = X**T or op(X) = X**H, alpha and beta are scalars, and A, B and C are matrices, with op(A) an m by k matrix, op(B) a k by n matrix and C an m by n matrix. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublaszgemm(h, transa, transb, m, n,
               k, alpha, a, lda, b, ldb, beta, c, ldc)
  type(cublasHandle) :: h
  integer :: transa, transb
  integer :: m, n, k, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device, dimension(ldc, *) :: c
  complex(8) :: alpha, beta
```

2.8.5.32. cublasZhemm

ZHEMM 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 scalars, A is an hermitian matrix and B and C are m by n matrices. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublaszhemm(h, side, uplo, m, n,
          alpha, a, lda, b, ldb, beta, c, ldc)
  type(cublasHandle) :: h
```

```
  integer :: side, uplo
  integer :: m, n, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device, dimension(ldc, *) :: c
  complex(8) :: alpha, beta
```

2.8.5.33. cublasZherk

ZHERK performs one of the hermitian rank k operations C := alpha*A*A**H + beta*C, or C := alpha*A**H*A + beta*C, where alpha and beta are real scalars, C is an n by n hermitian matrix and A is an n by k matrix in the first case and a k by n matrix in the second case. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublaszherk(h, uplo, trans, n, k,
                   alpha, a, lda, beta, c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldc, *) :: c
  real(8) :: alpha, beta
```

2.8.5.34. cublasZher2k

ZHER2K performs one of the hermitian rank 2k operations C := alpha*A*B**H + conjg(alpha)*B*A**H + beta*C, or C := alpha*A**H*B + conjg(alpha)*B**H*A + beta*C, where alpha and beta are scalars with beta real, C is an n by n hermitian matrix and A and B are n by k matrices in the first case and k by n matrices in the second case. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublaszher2k(h, uplo, trans, n, k,
              alpha, a, lda, b, ldb, beta, c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device, dimension(ldc, *) :: c
  complex(8) :: alpha
  real(8) :: beta
```

2.8.5.35. cublasZsymm

ZSYMM 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 scalars, A is a symmetric matrix and B and C are m by n matrices. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublaszsymm(h, side, uplo, m, n,
          alpha, a, lda, b, ldb, beta, c, ldc)
  type(cublasHandle) :: h
  integer :: side, uplo
  integer :: m, n, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device, dimension(ldc, *) :: c
  complex(8) :: alpha, beta
```

2.8.5.36. cublasZsyrk

ZSYRK performs one of the symmetric rank k operations C := alpha*A*A**T + beta*C, or C := alpha*A**T*A + beta*C, where alpha and beta are scalars, C is an n by n symmetric matrix and A is an n by k matrix in the first case and a k by n matrix in the second case. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublaszsyrk(h, uplo, trans, n, k,
                    alpha, a, lda, beta, c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldc, *) :: c
  complex(8) :: alpha, beta
```

2.8.5.37. cublasZsyr2k

ZSYR2K 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 scalars, C is an n by n symmetric matrix and A and B are n by k matrices in the first case and k by n matrices in the second case. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublaszsyr2k(h, uplo, trans, n, k,
               alpha, a, lda, b, ldb, beta, c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device, dimension(ldc, *) :: c
  complex(8) :: alpha, beta
```

2.8.5.38. cublasZtrmm

ZTRMM performs one of the matrix-matrix operations B := alpha*op(A)*B, or B := alpha*B*op(A) where alpha is a scalar, B is an m by n matrix, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A**T or op(A) = A**H. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasztrmm(h, side, uplo, transa,
                        diag, m, n, alpha, a, lda,
                               b, ldb, c, ldc)
  type(cublasHandle) :: h
  integer :: side, uplo, transa, diag
  integer :: m, n, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device, dimension(ldc, *) :: c
  complex(8) :: alpha
```

2.8.5.39. cublasZtrsm

ZTRSM solves one of the matrix equations op(A)*X = alpha*B, or X*op(A) = alpha*B, where alpha is a scalar, X and B are m by n matrices, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A**T or

op(A) = A**H. The matrix X is overwritten on B. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasztrsm(h, side, uplo, transa,
                diag, m, n, alpha, a, lda, b, ldb)
  type(cublasHandle) :: h
  integer :: side, uplo, transa, diag
  integer :: m, n, lda, ldb
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8) :: alpha
```

2.8.5.40. cublasZgemmBatched

ZGEMM performs one of the matrix-matrix operations C := alpha*op(A)*op(B) + beta*C, where op(X) is one of op(X) = X or op(X) = X**T or op(X) = X**H, alpha and beta are scalars, and A, B and C are matrices, with op(A) an m by k matrix, op(B) a k by n matrix and C an m by n matrix. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasZgemmBatched(h, transa, transb,                 m, n,
k, alpha, Aarray, lda, Barray, ldb, beta, Carray, ldc, batchCount)
  type(cublasHandle) :: h
  integer :: transa
  integer :: transb
  integer :: m, n, k
  complex(8) :: alpha
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: Barray(*)
  integer :: ldb
  complex(8) :: beta
  type(c_devptr), device :: Carray(*)
  integer :: ldc
  integer :: batchCount
```

2.8.5.41. cublasZgetrfBatched

ZGETRF computes an LU factorization of a general M-by-N matrix A using partial pivoting with row interchanges. The factorization has the form A = P * L * U where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if m > n), and U is upper triangular (upper trapezoidal if m < n). This is the right-looking Level 3 BLAS version of the algorithm. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasZgetrfBatched(h, n, Aarray, lda,
                                ipvt, info, batchCount)
  type(cublasHandle) :: h
  integer :: n
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  integer, device :: ipvt(*)
  integer, device :: info(*)
  integer :: batchCount
```

2.8.5.42. cublasZgetriBatched

ZGETRI computes the inverse of a matrix using the LU factorization computed by ZGETRF. This method inverts U and then computes inv(A) by solving the system

inv(A)*L = inv(U) for inv(A). Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasZgetriBatched(h, n, Aarray, lda,
                    ipvt, Carray, ldc, info, batchCount)
  type(cublasHandle) :: h
  integer :: n
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  integer, device :: ipvt(*)
  type(c_devptr), device :: Carray(*)
  integer :: ldc
  integer, device :: info(*)
  integer :: batchCount
```

2.8.5.43. cublasZgetrsBatched

ZGETRS solves a system of linear equations A * X = B, A**T * X = B, or A**H * X = B with a general N-by-N matrix A using the LU factorization computed by ZGETRF. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasZgetrsBatched(h, trans, n, nrhs,
                A, lda, ipvt, B, ldb, info, batchCount)
  type(cublasHandle) :: h
  integer :: trans
  integer :: n, nrhs
  type(c_devptr), device :: A(*)
  integer :: lda
  integer, device :: ipvt(*)
  type(c_devptr), device :: B(*)
  integer :: ldb
  integer, device :: info(*)
  integer :: batchCount
```

2.8.5.44. cublasZtrsmBatched

ZTRSM solves one of the matrix equations op(A)*X = alpha*B, or X*op(A) = alpha*B, where alpha is a scalar, X and B are m by n matrices, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A**T or op(A) = A**H. The matrix X is overwritten on B. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasZtrsmBatched(h, side, uplo,
    trans, diag, m, n, alpha, A, lda, B, ldb, batchCount)
  type(cublasHandle) :: h
  integer :: side
  integer :: uplo
  integer :: trans
  integer :: diag
  integer :: m, n
  complex(8) :: alpha
  type(c_devptr), device :: A(*)
  integer :: lda
  type(c_devptr), device :: B(*)
  integer :: ldb
  integer :: batchCount
```

2.8.5.45. cublasZmatinvBatched

cublasZmatinvBatched is a short cut of cublasZgetrfBatched plus cublasZgetriBatched. However it only works if n is less than 32. If not, the user has to go through

cublasZgetrfBatched and cublasZgetriBatched. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasZmatinvBatched(h, n, A, lda,
                Ainv, lda_inv, info, batchCount)
  type(cublasHandle) :: h
  integer :: n
  type(c_devptr), device :: A(*)
  integer :: lda
  type(c_devptr), device :: Ainv(*)
  integer :: lda_inv
  integer, device :: info(*)
  integer :: batchCount
```

2.8.5.46. cublasZgeqrfBatched

ZGEQRF computes a QR factorization of a complex M-by-N matrix A: A = Q * R. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasZgeqrfBatched(h, m, n,
     A, lda, Tau, info, batchCount)
  type(cublasHandle) :: h
  integer :: m, n
  type(c_devptr), device :: A(*)
  integer :: lda
  type(c_devptr), device :: Tau(*)
  integer, device :: info(*)
  integer :: batchCount
```

2.8.5.47. cublasZgelsBatched

ZGELS solves overdetermined or underdetermined complex linear systems involving an M-by-N matrix A, or its conjugate-transpose, using a QR or LQ factorization of A. It is assumed that A has full rank. The following options are provided: 1. If TRANS = 'N' and m >= n: find the least squares solution of an overdetermined system, i.e., solve the least squares problem minimize || B - A*X ||. 2. If TRANS = 'N' and m < n: find the minimum norm solution of an underdetermined system A * X = B. 3. If TRANS = 'C' and m >= n: find the minimum norm solution of an undetermined system A**H * X = B. 4. If TRANS = 'C' and m < n: find the least squares solution of an overdetermined system, i.e., solve the least squares problem minimize || B - A**H * X ||. Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the M-by-NRHS right hand side matrix B and the N-by-NRHS solution matrix X. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function cublasZgelsBatched(h, trans, m, n,
 nrhs, A, lda, C, ldc, info, dinfo, batchCount)
  type(cublasHandle) :: h
  integer :: trans
  integer :: m, n, nrhs
  type(c_devptr), device :: A(*)
  integer :: lda
  type(c_devptr), device :: C(*)
  integer :: ldc
  integer, device :: info(*)
  integer, device :: dinfo(*)
  integer :: batchCount
```

Chapter 3.
FFT RUNTIME LIBRARY APIS

This section describes the Fortran interfaces to the cuFFT library. The FFT functions are only accessible from host code. All of the runtime API routines are integer functions that return an error code; they return a value of CUFFT_SUCCESS if the call was successful, or another cuFFT status return value if there was an error.

Unless a specific kind is provided in the following interfaces, the plain integer type implies integer(4) and the plain real type implies real(4).

3.1. CUFFT Definitions and Helper Functions

This section contains definitions and data types used in the cuFFT library and interfaces to the cuFFT helper functions.

The cuFFT module contains the following constants and enumerations:

```
integer, parameter :: CUFFT_FORWARD = -1
integer, parameter :: CUFFT_INVERSE = 1
```

```
! CUFFT Status
enum, bind(C)
    enumerator :: CUFFT_SUCCESS        = 0
    enumerator :: CUFFT_INVALID_PLAN   = 1
    enumerator :: CUFFT_ALLOC_FAILED   = 2
    enumerator :: CUFFT_INVALID_TYPE   = 3
    enumerator :: CUFFT_INVALID_VALUE  = 4
    enumerator :: CUFFT_INTERNAL_ERROR = 5
    enumerator :: CUFFT_EXEC_FAILED    = 6
    enumerator :: CUFFT_SETUP_FAILED   = 7
    enumerator :: CUFFT_INVALID_SIZE   = 8
    enumerator :: CUFFT_UNALIGNED_DATA = 9
end enum
```

```
! CUFFT Transform Types
enum, bind(C)
    enumerator :: CUFFT_R2C = z'2a'     ! Real to Complex (interleaved)
    enumerator :: CUFFT_C2R = z'2c'     ! Complex (interleaved) to Real
    enumerator :: CUFFT_C2C = z'29'     ! Complex to Complex, interleaved
    enumerator :: CUFFT_D2Z = z'6a'     ! Double to Double-Complex
    enumerator :: CUFFT_Z2D = z'6c'     ! Double-Complex to Double
    enumerator :: CUFFT_Z2Z = z'69'     ! Double-Complex to Double-Complex
end enum
```

```
! CUFFT Data Layouts
```

```
enum, bind(C)
    enumerator :: CUFFT_COMPATIBILITY_NATIVE          = 0
    enumerator :: CUFFT_COMPATIBILITY_FFTW_PADDING    = 1
    enumerator :: CUFFT_COMPATIBILITY_FFTW_ASYMMETRIC = 2
    enumerator :: CUFFT_COMPATIBILITY_FFTW_ALL        = 3
end enum
```

```
integer, parameter :: CUFFT_COMPATIBILITY_DEFAULT =
 CUFFT_COMPATIBILITY_FFTW_PADDING
```

3.1.1. cufftSetCompatibilityMode

This function configures the layout of cuFFT output in FFTW-compatible modes.

```
integer(4) function cufftSetCompatibilityMode( plan, mode )
  integer :: plan
  integer :: mode
```

3.1.2. cufftSetStream

This function sets the stream to be used by the cuFFT library to execute its routines.

```
integer(4) function cufftSetStream(plan, stream)
  integer :: plan
  integer(kind=cuda_stream_kind()) :: stream
```

3.1.3. cufftGetVersion

This function returns the version number of cuFFT.

```
integer(4) function cufftGetVersion( version )
  integer :: version
```

3.1.4. cufftSetAutoAllocation

This function indicates that the caller intends to allocate and manage work areas for plans that have been generated. cuFFT default behavior is to allocate the work area at plan generation time. If cufftSetAutoAllocation() has been called with autoAllocate set to 0 prior to one of the cufftMakePlan*() calls, cuFFT does not allocate the work area. This is the preferred sequence for callers wishing to manage work area allocation.

```
integer(4) function cufftSetAutoAllocation(plan, autoAllocate)
  integer(4) :: plan, autoallocate
```

3.1.5. cufftSetWorkArea

This function overrides the work area pointer associated with a plan. If the work area was auto-allocated, cuFFT frees the auto-allocated space. The cufftExecute*() calls assume that the work area pointer is valid and that it points to a contiguous region in device memory that does not overlap with any other work area. If this is not the case, results are indeterminate.

```
integer(4) function cufftSetWorkArea(plan, workArea)
  integer(4) :: plan
  integer, device :: workArea(*)
```

3.1.6. cufftDestroy

This function frees all GPU resources associated with a cuFFT plan and destroys the internal plan data structure.

```
integer(4) function cufftDestroy( plan )
  integer :: plan
```

3.2. CUFFT Plans and Estimated Size Functions

This section contains functions from the cuFFT library used to create plans and estimate work buffer size.

3.2.1. cufftPlan1d

This function creates a 1D FFT plan configuration for a specified signal size and data type. Nx is the size of the transform; batch is the number of transforms of size nx.

```
integer(4) function cufftPlan1d(plan, nx, ffttype, batch)
  integer :: plan
  integer :: nx
  integer :: ffttype
  integer :: batch
```

3.2.2. cufftPlan2d

This function creates a 2D FFT plan configuration according to a specified signal size and data type. Nx is the size of the of the 1st dimension in the transform; ny is the size of the 2nd dimension.

```
integer(4) function cufftPlan2d( plan, nx, ny, ffttype )
  integer :: plan
  integer :: nx, ny
  integer :: ffttype
```

3.2.3. cufftPlan3d

This function creates a 3D FFT plan configuration according to a specified signal size and data type. Nx is the size of the of the 1st dimension in the transform; ny is the size of the 2nd dimension; nz is the size of the 3rd dimension.

```
integer(4) function cufftPlan3d( plan, nx, ny, nz, ffttype )
  integer :: plan
  integer :: nx, ny, nz
  integer :: ffttype
```

3.2.4. cufftPlanMany

This function creates an FFT plan configuration of dimension rank, with sizes specified in the array n. Batch is the number of transforms to configure. This function supports more complicated input and output data layouts using the arguments inembed, istride, idist, onembed, ostride, and odist. In the C function, if inembed and onembed are set to NULL, all other stride information is ignored. Fortran programmers can pass NULL when using the NVIDIA cufft module by setting an F90 pointer to null(), either through

direct assignment, using c_f_pointer() with c_null_ptr as the first argument, or the nullify statement, then passing the nullified F90 pointer as the actual argument for the inembed and onembed dummies.

```
integer(4) function cufftPlanMany(plan, rank, n, inembed, istride, idist,
onembed, ostride, odist, ffttype, batch )
  integer :: plan
  integer :: rank
  integer :: n
  integer :: inembed, onembed
  integer :: istride, idist, ostride, odist
  integer :: ffttype, batch
```

3.2.5. cufftCreate

This function creates an opaque handle for further cuFFT calls and allocates some small data structures on the host. In C, the handle type is currently typedef'ed to an int, so in Fortran we use an integer*4 to hold the plan.

```
integer(4) function cufftCreate(plan)
  integer(4) :: plan
```

3.2.6. cufftMakePlan1d

Following a call to cufftCreate(), this function creates a 1D FFT plan configuration for a specified signal size and data type. Nx is the size of the transform; batch is the number of transforms of size nx.

```
integer(4) function cufftMakePlan1d(plan, nx, ffttype, batch, worksize)
  integer(4) :: plan
  integer(4) :: nx
  integer(4) :: ffttype
  integer(4) :: batch
  integer(kind=int_ptr_kind()) :: workSize
```

3.2.7. cufftMakePlan2d

Following a call to cufftCreate(), this function creates a 2D FFT plan configuration according to a specified signal size and data type. Nx is the size of the of the 1st dimension in the transform; ny is the size of the 2nd dimension.

```
integer(4) function cufftMakePlan2d(plan, nx, ny, ffttype, workSize)
  integer(4) :: plan
  integer(4) :: nx, ny
  integer(4) :: ffttype
  integer(kind=int_ptr_kind()) :: workSize
```

3.2.8. cufftMakePlan3d

Following a call to cufftCreate(), this function creates a 3D FFT plan configuration according to a specified signal size and data type. Nx is the size of the of the 1st dimension in the transform; ny is the size of the 2nd dimension; nz is the size of the 3rd dimension.

```
integer(4) function cufftMakePlan3d(plan, nx, ny, nz, ffttype, workSize)
  integer(4) :: plan
  integer(4) :: nx, ny, nz
  integer(4) :: ffttype
  integer(kind=int_ptr_kind()) :: workSize
```

3.2.9. cufftMakePlanMany

Following a call to cufftCreate(), this function creates an FFT plan configuration of dimension rank, with sizes specified in the array n. Batch is the number of transforms to configure. This function supports more complicated input and output data layouts using the arguments inembed, istride, idist, onembed, ostride, and odist. In the C function, if inembed and onembed are set to NULL, all other stride information is ignored. Fortran programmers can pass NULL when using the NVIDIA cufft module by setting an F90 pointer to null(), either through direct assignment, using c_f_pointer() with c_null_ptr as the first argument, or the nullify statement, then passing the nullified F90 pointer as the actual argument for the inembed and onembed dummies.

```
integer(4) function cufftMakePlanMany(plan, rank, n, inembed, istride, idist,
onembed, ostride, odist, ffttype, batch, workSize)
  integer(4) :: plan
  integer(4) :: rank
  integer :: n
  integer :: inembed, onembed
  integer(4) :: istride, idist, ostride, odist
  integer(4) :: ffttype, batch
  integer(kind=int_ptr_kind()) :: workSize
```

3.2.10. cufftEstimate1d

This function returns an estimate for the size of the work area required, in bytes, given the specified size and data type, and assuming default plan settings.

```
integer(4) function cufftEstimate1d(nx, ffttype, batch, workSize)
  integer(4) :: nx
  integer(4) :: ffttype
  integer(4) :: batch
  integer(kind=int_ptr_kind()) :: workSize
```

3.2.11. cufftEstimate2d

This function returns an estimate for the size of the work area required, in bytes, given the specified size and data type, and assuming default plan settings.

```
integer(4) function cufftEstimate2d(nx, ny, ffttype, workSize)
  integer(4) :: nx, ny
  integer(4) :: ffttype
  integer(kind=int_ptr_kind()) :: workSize
```

3.2.12. cufftEstimate3d

This function returns an estimate for the size of the work area required, in bytes, given the specified size and data type, and assuming default plan settings.

```
integer(4) function cufftEstimate3d(nx, ny, nz, ffttype, workSize)
  integer(4) :: nx, ny, nz
  integer(4) :: ffttype
  integer(kind=int_ptr_kind()) :: workSize
```

3.2.13. cufftEstimateMany

This function returns an estimate for the size of the work area required, in bytes, given the specified size and data type, and assuming default plan settings.

```
integer(4) function cufftEstimateMany(rank, n, inembed, istride, idist,
onembed, ostride, odist, ffttype, batch, workSize)
  integer(4) :: rank, istride, idist, ostride, odist
  integer :: n, inembed, onembed
  integer(4) :: ffttype
  integer(4) :: batch
  integer(kind=int_ptr_kind()) :: workSize
```

3.2.14. cufftGetSize1d

This function gives a more accurate estimate than cufftEstimate1d() of the size of the work area required, in bytes, given the specified plan parameters and taking into account any plan settings which may have been made.

```
integer(4) function cufftGetSize1d(plan, nx, ffttype, batch, workSize)
  integer(4) :: plan, nx, ffttype, batch
  integer(kind=int_ptr_kind()) :: workSize
```

3.2.15. cufftGetSize2d

This function gives a more accurate estimate than cufftEstimate2d() of the size of the work area required, in bytes, given the specified plan parameters and taking into account any plan settings which may have been made.

```
integer(4) function cufftGetSize2d(plan, nx, ny, ffttype, workSize)
  integer(4) :: plan, nx, ny, ffttype
  integer(kind=int_ptr_kind()) :: workSize
```

3.2.16. cufftGetSize3d

This function gives a more accurate estimate than cufftEstimate3d() of the size of the work area required, in bytes, given the specified plan parameters and taking into account any plan settings which may have been made.

```
integer(4) function cufftGetSize3d(plan, nx, ny, nz, ffttype, workSize)
  integer(4) :: plan, nx, ny, nz, ffttype
  integer(kind=int_ptr_kind()) :: workSize
```

3.2.17. cufftGetSizeMany

This function gives a more accurate estimate than cufftEstimateMany() of the size of the work area required, in bytes, given the specified plan parameters and taking into account any plan settings which may have been made.

```
integer(4) function cufftGetSizeMany(plan, rank, n, inembed, istride, idist,
onembed, ostride, odist, ffttype, batch, workSize)
  integer(4) :: plan, rank, istride, idist, ostride, odist
  integer :: n, inembed, onembed
  integer(4) :: ffttype
  integer(4) :: batch
  integer(kind=int_ptr_kind()) :: workSize
```

3.2.18. cufftGetSize

Once plan generation has been done, either with the original API or the extensible API, this call returns the actual size of the work area required, in bytes, to support the plan. Callers who choose to manage work area allocation within their application must use this call after plan generation, and after any cufftSet*() calls subsequent to plan generation, if those calls might alter the required work space size.

```
integer(4) function cufftGetSize(plan, workSize)
  integer(4) :: plan
  integer(kind=int_ptr_kind()) :: workSize
```

3.3. CUFFT Execution Functions

This section contains definitions and data types used in the cuFFT library and interfaces to the cuFFT helper functions.

3.3.1. cufftExecC2C

This function executes a single precision complex-to-complex transform plan in the transform direction as specified by the direction parameter. If idata and odata are the same, this function does an in-place transform.

```
integer(4) function cufftExecC2C( plan, idata, odata, direction )
  integer :: plan
  complex(4), device, dimension(*) :: idata, odata
  integer :: direction
```

3.3.2. cufftExecR2C

This function executes a single precision real-to-complex, implicity forward, cuFFT transform plan. If idata and odata are the same, this function does an in-place transform, but note there are data layout differences between in-place and out-of-place transforms for real-to- complex FFTs in cuFFT.

```
integer(4) function cufftExecR2C( plan, idata, odata )
  integer :: plan
  real(4), device, dimension(*) :: idata
  complex(4), device, dimension(*) :: odata
```

3.3.3. cufftExecC2R

This function executes a single precision complex-to-real, implicity inverse, cuFFT transform plan. If idata and odata are the same, this function does an in-place transform.

```
integer(4) function cufftExecC2R( plan, idata, odata )
  integer :: plan
  complex(4), device, dimension(*) :: idata
  real(4), device, dimension(*) :: odata
```

3.3.4. cufftExecZ2Z

This function executes a double precision complex-to-complex transform plan in the transform direction as specified by the direction parameter. If idata and odata are the same, this function does an in-place transform.

```
integer(4) function cufftExecZ2Z( plan, idata, odata, direction )
  integer :: plan
  complex(8), device, dimension(*) :: idata, odata
  integer :: direction
```

3.3.5. cufftExecD2Z

This function executes a double precision real-to-complex, implicity forward, cuFFT transform plan. If idata and odata are the same, this function does an in-place transform, but note there are data layout differences between in-place and out-of-place transforms for real-to- complex FFTs in cuFFT.

```
integer(4) function cufftExecD2Z( plan, idata, odata )
  integer :: plan
  real(8), device, dimension(*) :: idata
  complex(8), device, dimension(*) :: odata
```

3.3.6. cufftExecZ2D

This function executes a double precision complex-to-real, implicity inverse, cuFFT transform plan. If idata and odata are the same, this function does an in-place transform.

```
integer(4) function cufftExecZ2D( plan, idata, odata )
  integer :: plan
  complex(8), device, dimension(*) :: idata
  real(8), device, dimension(*) :: odata
```

Chapter 4. RANDOM NUMBER RUNTIME LIBRARY APIS

This section describes the Fortran interfaces to the CUDA cuRAND library. The cuRAND functionality is accessible from both host and device code. In the host library, all of the runtime API routines are integer functions that return an error code; they return a value of CURAND_STATUS_SUCCESS if the call was successful, or other cuRAND return status value if there was an error. The host library routines are meant to produce a series or array of random numbers. In the device library, the init routines are subroutines and the generator functions return the type of the value being generated. The device library routines are meant for producing a single value per thread per call.

Unless a specific kind is provided, the plain integer type implies integer(4) and the plain real type implies real(4).

4.1. CURAND Definitions and Helper Functions

This section contains definitions and data types used in the cuRAND library and interfaces to the cuRAND helper functions.

The curand module contains the following derived type definitions:

```
TYPE curandGenerator
  TYPE(C_PTR)  :: handle
END TYPE
```

The curand module contains the following enumerations:

```
! CURAND Status
enum, bind(c)
    enumerator :: CURAND_STATUS_SUCCESS                   = 0
    enumerator :: CURAND_STATUS_VERSION_MISMATCH          = 100
    enumerator :: CURAND_STATUS_NOT_INITIALIZED           = 101
    enumerator :: CURAND_STATUS_ALLOCATION_FAILED         = 102
    enumerator :: CURAND_STATUS_TYPE_ERROR                = 103
    enumerator :: CURAND_STATUS_OUT_OF_RANGE              = 104
    enumerator :: CURAND_STATUS_LENGTH_NOT_MULTIPLE       = 105
    enumerator :: CURAND_STATUS_DOUBLE_PRECISION_REQUIRED = 106
    enumerator :: CURAND_STATUS_LAUNCH_FAILURE            = 201
    enumerator :: CURAND_STATUS_PREEXISTING_FAILURE       = 202
    enumerator :: CURAND_STATUS_INITIALIZATION_FAILED     = 203
    enumerator :: CURAND_STATUS_ARCH_MISMATCH             = 204
```

```
    enumerator :: CURAND_STATUS_INTERNAL_ERROR            = 999
end enum
```

```
! CURAND Generator Types
enum, bind(c)
    enumerator :: CURAND_RNG_TEST                    = 0
    enumerator :: CURAND_RNG_PSEUDO_DEFAULT          = 100
    enumerator :: CURAND_RNG_PSEUDO_XORWOW           = 101
    enumerator :: CURAND_RNG_PSEUDO_MRG32K3A         = 121
    enumerator :: CURAND_RNG_PSEUDO_MTGP32           = 141
    enumerator :: CURAND_RNG_PSEUDO_MT19937          = 142
    enumerator :: CURAND_RNG_PSEUDO_PHILOX4_32_10    = 161
    enumerator :: CURAND_RNG_QUASI_DEFAULT           = 200
    enumerator :: CURAND_RNG_QUASI_SOBOL32           = 201
    enumerator :: CURAND_RNG_QUASI_SCRAMBLED_SOBOL32 = 202
    enumerator :: CURAND_RNG_QUASI_SOBOL64           = 203
    enumerator :: CURAND_RNG_QUASI_SCRAMBLED_SOBOL64 = 204
end enum
```

```
! CURAND Memory Ordering
enum, bind(c)
    enumerator :: CURAND_ORDERING_PSEUDO_BEST    = 100
    enumerator :: CURAND_ORDERING_PSEUDO_DEFAULT = 101
    enumerator :: CURAND_ORDERING_PSEUDO_SEEDED  = 102
    enumerator :: CURAND_ORDERING_QUASI_DEFAULT  = 201
end enum
```

```
! CURAND Direction Vectors
enum, bind(c)
    enumerator :: CURAND_DIRECTION_VECTORS_32_JOEKUO6           = 101
    enumerator :: CURAND_SCRAMBLED_DIRECTION_VECTORS_32_JOEKUO6 = 102
    enumerator :: CURAND_DIRECTION_VECTORS_64_JOEKUO6           = 103
    enumerator :: CURAND_SCRAMBLED_DIRECTION_VECTORS_64_JOEKUO6 = 104
end enum
```

```
! CURAND Methods
enum, bind(c)
    enumerator :: CURAND_CHOOSE_BEST    = 0
    enumerator :: CURAND_ITR            = 1
    enumerator :: CURAND_KNUTH          = 2
    enumerator :: CURAND_HITR           = 3
    enumerator :: CURAND_M1             = 4
    enumerator :: CURAND_M2             = 5
    enumerator :: CURAND_BINARY_SEARCH  = 6
    enumerator :: CURAND_DISCRETE_GAUSS = 7
    enumerator :: CURAND_REJECTION      = 8
    enumerator :: CURAND_DEVICE_API     = 9
    enumerator :: CURAND_FAST_REJECTION = 10
    enumerator :: CURAND_3RD            = 11
    enumerator :: CURAND_DEFINITION     = 12
    enumerator :: CURAND_POISSON        = 13
end enum
```

4.1.1. curandCreateGenerator

This function creates a new random number generator of type rng. See the beginning of this section for valid values of rng.

```
 integer(4) function curandCreateGenerator(generator, rng)
   type(curandGenerator) :: generator
   integer :: rng
```

4.1.2. curandCreateGeneratorHost

This function creates a new host CPU random number generator of type rng. See the beginning of this section for valid values of rng.

```
integer(4) function curandCreateGeneratorHost(generator, rng)
  type(curandGenerator) :: generator
  integer :: rng
```

4.1.3. curandDestroyGenerator

This function destroys an existing random number generator.

```
integer(4) function curandDestroyGenerator(generator)
  type(curandGenerator) :: generator
```

4.1.4. curandGetVersion

This function returns the version number of the cuRAND library.

```
integer(4) function curandGetVersion(version)
  integer(4) :: version
```

4.1.5. curandSetStream

This function sets the current stream for the cuRAND kernel launches.

```
integer(4) function curandSetStream(generator, stream)
  type(curandGenerator) :: generator
  integer(kind=c_intptr_t) :: stream
```

4.1.6. curandSetPseudoRandomGeneratorSeed

This function sets the seed value of the pseudo-random number generator.

```
integer(4) function curandSetPseudoRandomGeneratorSeed(generator, seed)
  type(curandGenerator) :: generator
  integer(8) :: seed
```

4.1.7. curandSetGeneratorOffset

This function sets the absolute offset of the pseudo or quasirandom number generator.

```
integer(4) function curandSetGeneratorOffset(generator, offset)
  type(curandGenerator) :: generator
  integer(8) :: offset
```

4.1.8. curandSetGeneratorOrdering

This function sets the ordering of results of the pseudo or quasirandom number generator.

```
integer(4) function curandSetGeneratorOrdering(generator, order)
  type(curandGenerator) :: generator
  integer(4) :: order
```

4.1.9. curandSetQuasiRandomGeneratorDimensions

This function sets number of dimensions of the quasirandom number generator.

```
integer(4) function curandSetQuasiRandomGeneratorDimensions(generator, num)
  type(curandGenerator) :: generator
  integer(4) :: num
```

4.2. CURAND Generator Functions

This section contains interfaces for the cuRAND generator functions.

4.2.1. curandGenerate

This function generates 32-bit pseudo or quasirandom numbers.

```
integer(4) function curandGenerate(generator, array, num )
  type(curandGenerator) :: generator
  integer(4), device :: array(*) ! Host or device depending on the generator
  integer(kind=c_intptr_t) :: num
```

4.2.2. curandGenerateLongLong

This function generates 64-bit integer quasirandom numbers. The function curandGenerate() has also been overloaded to accept these function arguments.

```
integer(4) function curandGenerateLongLong(generator, array, num )
  type(curandGenerator) :: generator
  integer(8), device :: array(*) ! Host or device depending on the generator
  integer(kind=c_intptr_t) :: num
```

4.2.3. curandGenerateUniform

This function generates 32-bit floating point uniformly distributed random numbers. The function curandGenerate() has also been overloaded to accept these function arguments.

```
integer(4) function curandGenerateUniform(generator, array, num )
  type(curandGenerator) :: generator
  real(4), device :: array(*) ! Host or device depending on the generator
  integer(kind=c_intptr_t) :: num
```

4.2.4. curandGenerateUniformDouble

This function generates 64-bit floating point uniformly distributed random numbers. The function curandGenerate() has also been overloaded to accept these function arguments.

```
integer(4) function curandGenerateUniformDouble(generator, array, num )
  type(curandGenerator) :: generator
  real(8), device :: array(*) ! Host or device depending on the generator
  integer(kind=c_intptr_t) :: num
```

4.2.5. curandGenerateNormal

This function generates 32-bit floating point normally distributed random numbers. The function curandGenerate() has also been overloaded to accept these function arguments.

```
integer(4) function curandGenerateNormal(generator, array, num, mean, stddev )
  type(curandGenerator) :: generator
  real(4), device :: array(*) ! Host or device depending on the generator
  integer(kind=c_intptr_t) :: num
  real(4) :: mean, stddev
```

4.2.6. curandGenerateNormalDouble

This function generates 64-bit floating point normally distributed random numbers. The function curandGenerate() has also been overloaded to accept these function arguments.

```
integer(4) function curandGenerateNormalDouble(generator, array, num, mean,
stddev )
  type(curandGenerator) :: generator
  real(8), device :: array(*) ! Host or device depending on the generator
  integer(kind=c_intptr_t) :: num
  real(8) :: mean, stddev
```

4.2.7. curandGeneratePoisson

This function generates Poisson-distributed random numbers. The function curandGenerate() has also been overloaded to accept these function arguments.

```
integer(4) function curandGeneratePoisson(generator, array, num, lambda )
  type(curandGenerator) :: generator
  real(8), device :: array(*) ! Host or device depending on the generator
  integer(kind=c_intptr_t) :: num
  real(8) :: lambda
```

4.2.8. curandGenerateSeeds

This function sets the starting state of the generator.

```
integer(4) function curandGenerateSeeds(generator)
  type(curandGenerator) :: generator
```

4.2.9. curandGenerateLogNormal

This function generates 32-bit floating point log-normally distributed random numbers.

```
integer(4) function curandGenerateLogNormal(generator, array, num, mean,
stddev )
  type(curandGenerator) :: generator
  real(4), device :: array(*) ! Host or device depending on the generator
  integer(kind=c_intptr_t) :: num
  real(4) :: mean, stddev
```

4.2.10. curandGenerateLogNormalDouble

This function generates 64-bit floating point log-normally distributed random numbers.

```
integer(4) function curandGenerateLogNormalDouble(generator, array, num, mean,
stddev )
  type(curandGenerator) :: generator
  real(8), device :: array(*) ! Host or device depending on the generator
```

```
   integer(kind=c_intptr_t) :: num
   real(8) :: mean, stddev
```

4.3. CURAND Device Definitions and Functions

This section contains definitions and data types used in the cuRAND device library and interfaces to the cuRAND functions.

The curand device module contains the following derived type definitions:

```
TYPE curandStateXORWOW
    integer(4) :: d
    integer(4) :: v(5)
    integer(4) :: boxmuller_flag
    integer(4) :: boxmuller_flag_double
    real(4)    :: boxmuller_extra
    real(8)    :: boxmuller_extra_double
END TYPE curandStateXORWOW
```

```
TYPE curandStateMRG32k3a
    real(8)    :: s1(3)
    real(8)    :: s2(3)
    integer(4) :: boxmuller_flag
    integer(4) :: boxmuller_flag_double
    real(4)    :: boxmuller_extra
    real(8)    :: boxmuller_extra_double
END TYPE curandStateMRG32k3a
```

```
TYPE curandStateSobol32
    integer(4) :: d
    integer(4) :: x
    integer(4) :: c
    integer(4) :: direction_vectors(32)
END TYPE curandStateSobol32
```

```
TYPE curandStateScrambledSobol32
    integer(4) :: d
    integer(4) :: x
    integer(4) :: c
    integer(4) :: direction_vectors(32)
END TYPE curandStateScrambledSobol32
```

```
TYPE curandStateSobol64
    integer(8) :: d
    integer(8) :: x
    integer(8) :: c
    integer(8) :: direction_vectors(32)
END TYPE curandStateSobol64
```

```
TYPE curandStateScrambledSobol64
    integer(8) :: d
    integer(8) :: x
    integer(8) :: c
    integer(8) :: direction_vectors(32)
END TYPE curandStateScrambledSobol64
```

```
TYPE curandStateMtgp32
    integer(4) :: s(MTGP32_STATE_SIZE)
    integer(4) :: offset
    integer(4) :: pIdx
    integer(kind=int_ptr_kind()) :: k
    integer(4) :: precise_double_flag
END TYPE curandStateMtgp32
```

```
TYPE curandStatePhilox4_32_10
    integer(4) :: ctr
```

```
    integer(4) :: output
    integer(2) :: key
    integer(4) :: state
    integer(4) :: boxmuller_flag
    integer(4) :: boxmuller_flag_double
    real(4)    :: boxmuller_extra
    real(8)    :: boxmuller_extra_double
END TYPE curandStatePhilox4_32_10
```

4.3.1. curand_Init

This overloaded device subroutine initializes the state for the random number generator. These device subroutines are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

4.3.1.1. curandInitXORWOW

This function initializes the state for the XORWOW random number generator. The function curand_init() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
subroutine curandInitXORWOW(seed, sequence, offset, state)
  integer(8) :: seed
  integer(8) :: sequence
  integer(8) :: offset
  TYPE(curandStateXORWOW) :: state
```

4.3.1.2. curandInitMRG32k3a

This function initializes the state for the MRG32k3a random number generator. The function curand_init() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
subroutine curandInitMRG32k3a(seed, sequence, offset, state)
  integer(8) :: seed
  integer(8) :: sequence
  integer(8) :: offset
  TYPE(curandStateMRG32k3a) :: state
```

4.3.1.3. curandInitPhilox4_32_10

This function initializes the state for the Philox4_32_10 random number generator. The function curand_init() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
subroutine curandInitPhilox4_32_10(seed, sequence, offset, state)
  integer(8) :: seed
  integer(8) :: sequence
  integer(8) :: offset
  TYPE(curandStatePhilox4_32_10) :: state
```

4.3.1.4. curandInitSobol32

This function initializes the state for the Sobol32 random number generator. The function curand_init() has also been overloaded to accept these function arguments, as in CUDA

C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
subroutine curandInitSobol32(direction_vectors, offset, state)
  integer :: direction_vectors(*)
  integer(4) :: offset
  TYPE(curandStateSobol32) :: state
```

4.3.1.5. curandInitScrambledSobol32

This function initializes the state for the scrambled Sobol32 random number generator. The function curand_init() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
subroutine curandInitScrambledSobol32(direction_vectors, scramble, offset,
state)
  integer :: direction_vectors(*)
  integer(4) :: scramble
  integer(4) :: offset
  TYPE(curandStateScrambledSobol32) :: state
```

4.3.1.6. curandInitSobol64

This function initializes the state for the Sobol64 random number generator. The function curand_init() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
subroutine curandInitSobol64(direction_vectors, offset, state)
  integer :: direction_vectors(*)
  integer(8) :: offset
  TYPE(curandStateSobol64) :: state
```

4.3.1.7. curandInitScrambledSobol64

This function initializes the state for the scrambled Sobol64 random number generator. The function curand_init() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
subroutine curandInitScrambledSobol64(direction_vectors, scramble, offset,
state)
  integer :: direction_vectors(*)
  integer(8) :: scramble
  integer(8) :: offset
  TYPE(curandStateScrambledSobol64) :: state
```

4.3.2. curand

This overloaded device function returns 32 or 64 bits or random data based on the state argument. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

4.3.2.1. curandGetXORWOW

This function returns 32 bits of pseudorandomness from the XORWOW random number generator. The function curand() has also been overloaded to accept these function

arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function curandGetXORWOW(state)
  TYPE(curandStateXORWOW) :: state
```

4.3.2.2. curandGetMRG32k3a

This function returns 32 bits of pseudorandomness from the MRG32k3a random number generator. The function curand() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function curandGetMRG32k3a(state)
  TYPE(curandStateMRG32k3a) :: state
```

4.3.2.3. curandGetPhilox4_32_10

This function returns 32 bits of pseudorandomness from the Philox4_32_10 random number generator. The function curand() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function curandGetPhilox4_32_10(state)
  TYPE(curandStatePhilox4_32_10) :: state
```

4.3.2.4. curandGetSobol32

This function returns 32 bits of quasirandomness from the Sobol32 random number generator. The function curand() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function curandGetSobol32(state)
  TYPE(curandStateSobol32) :: state
```

4.3.2.5. curandGetScrambledSobol32

This function returns 32 bits of quasirandomness from the scrambled Sobol32 random number generator. The function curand() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function curandGetScrambledSobol32(state)
  TYPE(curandStateScrambledSobol32) :: state
```

4.3.2.6. curandGetSobol64

This function returns 64 bits of quasirandomness from the Sobol64 random number generator. The function curand() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function curandGetSobol64(state)
  TYPE(curandStateSobol64) :: state
```

4.3.2.7. curandGetScrambledSobol64

This function returns 64 bits of quasirandomness from the scrambled Sobol64 random number generator. The function curand() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
integer(4) function curandGetScrambledSobol64(state)
  TYPE(curandStateScrambledSobol64) :: state
```

4.3.3. Curand_Normal

This overloaded device function returns a 32-bit floating point normally distributed random number. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

4.3.3.1. curandNormalXORWOW

This function returns a 32-bit floating point normally distributed random number from an XORWOW generator. The function curand_normal() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
real(4) function curandNormalXORWOW(state)
  TYPE(curandStateXORWOW) :: state
```

4.3.3.2. curandNormalMRG32k3a

This function returns a 32-bit floating point normally distributed random number from an MRG32k3a generator. The function curand_normal() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
real(4) function curandNormalMRG32k3a(state)
  TYPE(curandStateMRG32k3a) :: state
```

4.3.3.3. curandNormalPhilox4_32_10

This function returns a 32-bit floating point normally distributed random number from a Philox4_32_10 generator. The function curand_normal() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
real(4) function curandNormalPhilox4_32_10(state)
  TYPE(curandStatePhilox4_32_10) :: state
```

4.3.3.4. curandNormalSobol32

This function returns a 32-bit floating point normally distributed random number from an Sobol32 generator. The function curand_normal() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
real(4) function curandNormalSobol32(state)
  TYPE(curandStateSobol32) :: state
```

4.3.3.5. curandNormalScrambledSobol32

This function returns a 32-bit floating point normally distributed random number from an scrambled Sobol32 generator. The function curand_normal() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
real(4) function curandNormalScrambledSobol32(state)
  TYPE(curandStateScrambledSobol32) :: state
```

4.3.3.6. curandNormalSobol64

This function returns a 32-bit floating point normally distributed random number from an Sobol64 generator. The function curand_normal() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
real(4) function curandNormalSobol64(state)
  TYPE(curandStateSobol64) :: state
```

4.3.3.7. curandNormalScrambledSobol64

This function returns a 32-bit floating point normally distributed random number from an scrambled Sobol64 generator. The function curand_normal() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
real(4) function curandNormalScrambledSobol64(state)
  TYPE(curandStateScrambledSobol64) :: state
```

4.3.4. Curand_Normal_Double

This overloaded device function returns a 64-bit floating point normally distributed random number. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

4.3.4.1. curandNormalDoubleXORWOW

This function returns a 64-bit floating point normally distributed random number from an XORWOW generator. The function curand_normal_double() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
real(8) function curandNormalDoubleXORWOW(state)
  TYPE(curandStateXORWOW) :: state
```

4.3.4.2. curandNormalDoubleMRG32k3a

This function returns a 64-bit floating point normally distributed random number from an MRG32k3a generator. The function curand_normal_double() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
real(8) function curandNormalDoubleMRG32k3a(state)
  TYPE(curandStateMRG32k3a) :: state
```

4.3.4.3. curandNormalDoublePhilox4_32_10

This function returns a 64-bit floating point normally distributed random number from a Philox4_32_10 generator. The function curand_normal_double() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
real(8) function curandNormalDoublePhilox4_32_10(state)
  TYPE(curandStatePhilox4_32_10) :: state
```

4.3.4.4. curandNormalDoubleSobol32

This function returns a 64-bit floating point normally distributed random number from an Sobol32 generator. The function curand_normal_double() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
real(8) function curandNormalDoubleSobol32(state)
  TYPE(curandStateSobol32) :: state
```

4.3.4.5. curandNormalDoubleScrambledSobol32

This function returns a 64-bit floating point normally distributed random number from an scrambled Sobol32 generator. The function curand_normal_double() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
real(8) function curandNormalDoubleScrambledSobol32(state)
  TYPE(curandStateScrambledSobol32) :: state
```

4.3.4.6. curandNormalDoubleSobol64

This function returns a 64-bit floating point normally distributed random number from an Sobol64 generator. The function curand_normal_double() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
real(8) function curandNormalDoubleSobol64(state)
  TYPE(curandStateSobol64) :: state
```

4.3.4.7. curandNormalDoubleScrambledSobol64

This function returns a 64-bit floating point normally distributed random number from an scrambled Sobol64 generator. The function curand_normal_double() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
real(8) function curandNormalDoubleScrambledSobol64(state)
  TYPE(curandStateScrambledSobol64) :: state
```

4.3.5. Curand_Log_Normal

This overloaded device function returns a 32-bit floating point log-normally distributed random number. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

4.3.5.1. curandLogNormalXORWOW

This function returns a 32-bit floating point log-normally distributed random number from an XORWOW generator. The function curand_log_normal() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
real(4) function curandLogNormalXORWOW(state)
  TYPE(curandStateXORWOW) :: state
```

4.3.5.2. curandLogNormalMRG32k3a

This function returns a 32-bit floating point log-normally distributed random number from an MRG32k3a generator. The function curand_log_normal() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
real(4) function curandLogNormalMRG32k3a(state)
  TYPE(curandStateMRG32k3a) :: state
```

4.3.5.3. curandLogNormalPhilox4_32_10

This function returns a 32-bit floating point log-normally distributed random number from a Philox4_32_10 generator. The function curand_log_normal() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
real(4) function curandLogNormalPhilox4_32_10(state)
  TYPE(curandStatePhilox4_32_10) :: state
```

4.3.5.4. curandLogNormalSobol32

This function returns a 32-bit floating point log-normally distributed random number from an Sobol32 generator. The function curand_log_normal() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
real(4) function curandLogNormalSobol32(state)
  TYPE(curandStateSobol32) :: state
```

4.3.5.5. curandLogNormalScrambledSobol32

This function returns a 32-bit floating point log-normally distributed random number from an scrambled Sobol32 generator. The function curand_log_normal() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
real(4) function curandLogNormalScrambledSobol32(state)
  TYPE(curandStateScrambledSobol32) :: state
```

4.3.5.6. curandLogNormalSobol64

This function returns a 32-bit floating point log-normally distributed random number from an Sobol64 generator. The function curand_log_normal() has also been overloaded

to accept these function arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
real(4) function curandLogNormalSobol64(state)
  TYPE(curandStateSobol64) :: state
```

4.3.5.7. curandLogNormalScrambledSobol64

This function returns a 32-bit floating point log-normally distributed random number from an scrambled Sobol64 generator. The function curand_log_normal() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
real(4) function curandLogNormalScrambledSobol64(state)
  TYPE(curandStateScrambledSobol64) :: state
```

4.3.6. Curand_Log_Normal_Double

This overloaded device function returns a 64-bit floating point log-normally distributed random number. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

4.3.6.1. curandLogNormalDoubleXORWOW

This function returns a 64-bit floating point log-normally distributed random number from an XORWOW generator. The function curand_log_normal_double() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
real(8) function curandLogNormalDoubleXORWOW(state)
  TYPE(curandStateXORWOW) :: state
```

4.3.6.2. curandLogNormalDoubleMRG32k3a

This function returns a 64-bit floating point log-normally distributed random number from an MRG32k3a generator. The function curand_log_normal_double() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
real(8) function curandLogNormalDoubleMRG32k3a(state)
  TYPE(curandStateMRG32k3a) :: state
```

4.3.6.3. curandLogNormalDoublePhilox4_32_10

This function returns a 64-bit floating point log-normally distributed random number from a Philox4_32_10 generator. The function curand_log_normal_double() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
real(8) function curandLogNormalDoublePhilox4_32_10(state)
  TYPE(curandStatePhilox4_32_10) :: state
```

4.3.6.4. curandLogNormalDoubleSobol32

This function returns a 64-bit floating point log-normally distributed random number from an Sobol32 generator. The function curand_log_normal_double() has also been

overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
real(8) function curandLogNormalDoubleSobol32(state)
  TYPE(curandStateSobol32) :: state
```

4.3.6.5. curandLogNormalDoubleScrambledSobol32

This function returns a 64-bit floating point log-normally distributed random number from an scrambled Sobol32 generator. The function curand_log_normal_double() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
real(8) function curandLogNormalDoubleScrambledSobol32(state)
  TYPE(curandStateScrambledSobol32) :: state
```

4.3.6.6. curandLogNormalDoubleSobol64

This function returns a 64-bit floating point log-normally distributed random number from an Sobol64 generator. The function curand_log_normal_double() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
real(8) function curandLogNormalDoubleSobol64(state)
  TYPE(curandStateSobol64) :: state
```

4.3.6.7. curandLogNormalDoubleScrambledSobol64

This function returns a 64-bit floating point log-normally distributed random number from an scrambled Sobol64 generator. The function curand_log_normal_double() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
real(8) function curandLogNormalDoubleScrambledSobol64(state)
  TYPE(curandStateScrambledSobol64) :: state
```

4.3.7. Curand_Uniform

This overloaded device function returns a 32-bit floating point uniformly distributed random number. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

4.3.7.1. curandUniformXORWOW

This function returns a 32-bit floating point uniformly distributed random number from an XORWOW generator. The function curand_uniform() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
real(4) function curandUniformXORWOW(state)
  TYPE(curandStateXORWOW) :: state
```

4.3.7.2. curandUniformMRG32k3a

This function returns a 32-bit floating point uniformly distributed random number from an MRG32k3a generator. The function curand_uniform() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
real(4) function curandUniformMRG32k3a(state)
  TYPE(curandStateMRG32k3a) :: state
```

4.3.7.3. curandUniformPhilox4_32_10

This function returns a 32-bit floating point uniformly distributed random number from a Philox4_32_10 generator. The function curand_uniform() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
real(4) function curandUniformPhilox4_32_10(state)
  TYPE(curandStatePhilox4_32_10) :: state
```

4.3.7.4. curandUniformSobol32

This function returns a 32-bit floating point uniformly distributed random number from an Sobol32 generator. The function curand_uniform() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
real(4) function curandUniformSobol32(state)
  TYPE(curandStateSobol32) :: state
```

4.3.7.5. curandUniformScrambledSobol32

This function returns a 32-bit floating point uniformly distributed random number from an scrambled Sobol32 generator. The function curand_uniform() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
real(4) function curandUniformScrambledSobol32(state)
  TYPE(curandStateScrambledSobol32) :: state
```

4.3.7.6. curandUniformSobol64

This function returns a 32-bit floating point uniformly distributed random number from an Sobol64 generator. The function curand_uniform() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
real(4) function curandUniformSobol64(state)
  TYPE(curandStateSobol64) :: state
```

4.3.7.7. curandUniformScrambledSobol64

This function returns a 32-bit floating point uniformly distributed random number from an scrambled Sobol64 generator. The function curand_uniform() has also been

overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
real(4) function curandUniformScrambledSobol64(state)
  TYPE(curandStateScrambledSobol64) :: state
```

4.3.8. Curand_Uniform_Double

This overloaded device function returns a 64-bit floating point uniformly distributed random number. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

4.3.8.1. curandUniformDoubleXORWOW

This function returns a 64-bit floating point uniformly distributed random number from an XORWOW generator. The function curand_uniform_double() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
real(8) function curandUniformDoubleXORWOW(state)
  TYPE(curandStateXORWOW) :: state
```

4.3.8.2. curandUniformDoubleMRG32k3a

This function returns a 64-bit floating point uniformly distributed random number from an MRG32k3a generator. The function curand_uniform_double() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
real(8) function curandUniformDoubleMRG32k3a(state)
  TYPE(curandStateMRG32k3a) :: state
```

4.3.8.3. curandUniformDoublePhilox4_32_10

This function returns a 64-bit floating point uniformly distributed random number from a Philox4_32_10 generator. The function curand_uniform_double() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
real(8) function curandUniformDoublePhilox4_32_10(state)
  TYPE(curandStatePhilox4_32_10) :: state
```

4.3.8.4. curandUniformDoubleSobol32

This function returns a 64-bit floating point uniformly distributed random number from an Sobol32 generator. The function curand_uniform_double() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
real(8) function curandUniformDoubleSobol32(state)
  TYPE(curandStateSobol32) :: state
```

4.3.8.5. curandUniformDoubleScrambledSobol32

This function returns a 64-bit floating point uniformly distributed random number from an scrambled Sobol32 generator. The function curand_uniform_double() has also been

overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
real(8) function curandUniformDoubleScrambledSobol32(state)
  TYPE(curandStateScrambledSobol32) :: state
```

4.3.8.6. curandUniformDoubleSobol64

This function returns a 64-bit floating point uniformly distributed random number from an Sobol64 generator. The function curand_uniform_double() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
real(8) function curandUniformDoubleSobol64(state)
  TYPE(curandStateSobol64) :: state
```

4.3.8.7. curandUniformDoubleScrambledSobol64

This function returns a 64-bit floating point uniformly distributed random number from an scrambled Sobol64 generator. The function curand_uniform_double() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared "attributes(device)" in CUDA Fortran and "!$acc routine() seq" in OpenACC.

```
real(8) function curandUniformDoubleScrambledSobol64(state)
  TYPE(curandStateScrambledSobol64) :: state
```

Chapter 5.
SPARSE MATRIX RUNTIME LIBRARY APIS

This section describes the Fortran interfaces to the CUDA cuSPARSE library. The cuSPARSE functions are only accessible from host code. All of the runtime API routines are integer functions that return an error code; they return a value of CUSPARSE_STATUS_SUCCESS if the call was successful, or another cuSPARSE status return value if there was an error.

Chapter 8 contains examples of accessing the cuSPARSE library routines from OpenACC and CUDA Fortran. In both cases, the interfaces to the library can be exposed by adding the line

```
use cusparse
```

to your program unit.

A number of the function interfaces listed in this chapter can take host or device scalar arguments. Those functions have an additional v2 interface, which does not implicitly manage the pointer mode for these calls. See section 1.6 for further discussion on the handling of pointer modes.

Unless a specific kind is provided, the plain integer type used in the interfaces implies integer(4) and the plain real type implies real(4).

5.1. CUSPARSE Definitions and Helper Functions

This section contains definitions and data types used in the cuSPARSE library and interfaces to the cuSPARSE helper functions.

The cuSPARSE module contains the following derived type definitions:

```
type cusparseHandle
  type(c_ptr) :: handle
end type cusparseHandle
```

```
type :: cusparseMatDescr
     type(c_ptr) :: descr
end type cusparseMatDescr
```

```
type cusparseSolveAnalysisInfo
     type(c_ptr) :: info
```

```
end type cusparseSolveAnalysisInfo

type cusparseHybMat
     type(c_ptr) :: mat
end type cusparseHybMat

type cusparseCsrsv2Info
     type(c_ptr) :: info
end type cusparseCsrsv2Info

type cusparseCsric02Info
     type(c_ptr) :: info
end type cusparseCsric02Info

type cusparseCsrilu02Info
     type(c_ptr) :: info
end type cusparseCsrilu02Info

type cusparseBsrsv2Info
     type(c_ptr) :: info
end type cusparseBsrsv2Info

type cusparseBsric02Info
     type(c_ptr) :: info
end type cusparseBsric02Info

type cusparseBsrilu02Info
     type(c_ptr) :: info
end type cusparseBsrilu02Info

type cusparseBsrsm2Info
     type(c_ptr) :: info
end type cusparseBsrsm2Info

type cusparseCsrgemm2Info
     type(c_ptr) :: info
end type cusparseCsrgemm2Info

type cusparseColorInfo
     type(c_ptr) :: info
end type cusparseColorInfo

type cusparseCsru2csrInfo
     type(c_ptr) :: info
end type cusparseCsru2csrInfo

type cusparseSpVecDescr
     type(c_ptr) :: descr
end type cusparseSpVecDescr

type cusparseDnVecDescr
       type(c_ptr) :: descr
end type cusparseDnVecDescr

type cusparseSpMatDescr
       type(c_ptr) :: descr
end type cusparseSpMatDescr

type cusparseDnMatDescr
       type(c_ptr) :: descr
end type cusparseDnMatDescr
```

The cuSPARSE module contains the following enumerations:

```
! cuSPARSE status return values
enum, bind(C) ! cusparseStatus_t
    enumerator :: CUSPARSE_STATUS_SUCCESS=0
    enumerator :: CUSPARSE_STATUS_NOT_INITIALIZED=1
    enumerator :: CUSPARSE_STATUS_ALLOC_FAILED=2
    enumerator :: CUSPARSE_STATUS_INVALID_VALUE=3
    enumerator :: CUSPARSE_STATUS_ARCH_MISMATCH=4
    enumerator :: CUSPARSE_STATUS_MAPPING_ERROR=5
```

```
    enumerator :: CUSPARSE_STATUS_EXECUTION_FAILED=6
    enumerator :: CUSPARSE_STATUS_INTERNAL_ERROR=7
    enumerator :: CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED=8
    enumerator :: CUSPARSE_STATUS_ZERO_PIVOT=9
    enumerator :: CUSPARSE_STATUS_NOT_SUPPORTED=10
end enum

enum, bind(c) ! cusparsePointerMode_t
    enumerator :: CUSPARSE_POINTER_MODE_HOST = 0
    enumerator :: CUSPARSE_POINTER_MODE_DEVICE = 1
end enum

enum, bind(c) ! cusparseAction_t
    enumerator :: CUSPARSE_ACTION_SYMBOLIC = 0
    enumerator :: CUSPARSE_ACTION_NUMERIC = 1
end enum

enum, bind(C) ! cusparseMatrixType_t
    enumerator :: CUSPARSE_MATRIX_TYPE_GENERAL = 0
    enumerator :: CUSPARSE_MATRIX_TYPE_SYMMETRIC = 1
    enumerator :: CUSPARSE_MATRIX_TYPE_HERMITIAN = 2
    enumerator :: CUSPARSE_MATRIX_TYPE_TRIANGULAR = 3
end enum

enum, bind(C) ! cusparseFillMode_t
    enumerator :: CUSPARSE_FILL_MODE_LOWER = 0
    enumerator :: CUSPARSE_FILL_MODE_UPPER = 1
end enum

enum, bind(C) ! cusparseDiagType_t
    enumerator :: CUSPARSE_DIAG_TYPE_NON_UNIT = 0
    enumerator :: CUSPARSE_DIAG_TYPE_UNIT = 1
end enum

enum, bind(C) ! cusparseIndexBase_t
    enumerator :: CUSPARSE_INDEX_BASE_ZERO = 0
    enumerator :: CUSPARSE_INDEX_BASE_ONE = 1
end enum

enum, bind(C) ! cusparseOperation_t
    enumerator :: CUSPARSE_OPERATION_NON_TRANSPOSE = 0
    enumerator :: CUSPARSE_OPERATION_TRANSPOSE = 1
    enumerator :: CUSPARSE_OPERATION_CONJUGATE_TRANSPOSE = 2
end enum

enum, bind(C) ! cusparseDirection_t
    enumerator :: CUSPARSE_DIRECTION_ROW = 0
    enumerator :: CUSPARSE_DIRECTION_COLUMN = 1
end enum

enum, bind(C) ! cusparseHybPartition_t
    enumerator :: CUSPARSE_HYB_PARTITION_AUTO = 0
    enumerator :: CUSPARSE_HYB_PARTITION_USER = 1
    enumerator :: CUSPARSE_HYB_PARTITION_MAX = 2
end enum

enum, bind(C) ! cusparseSolvePolicy_t
    enumerator :: CUSPARSE_SOLVE_POLICY_NO_LEVEL = 0
    enumerator :: CUSPARSE_SOLVE_POLICY_USE_LEVEL = 1
end enum

enum, bind(C) ! cusparseSideMode_t
    enumerator :: CUSPARSE_SIDE_LEFT  = 0
    enumerator :: CUSPARSE_SIDE_RIGHT = 1
end enum

enum, bind(C) ! cusparseColorAlg_t
    enumerator :: CUSPARSE_COLOR_ALG0 = 0
    enumerator :: CUSPARSE_COLOR_ALG1 = 1
```

```
end enum
```

```
enum, bind(C) ! cusparseAlgMode_t;
    enumerator :: CUSPARSE_ALG0           = 0
    enumerator :: CUSPARSE_ALG1           = 1
    enumerator :: CUSPARSE_ALG_NAIVE      = 0
    enumerator :: CUSPARSE_ALG_MERGE_PATH = 1
end enum
```

```
enum, bind(C) ! cusparseCsr2CscAlg_t;
    enumerator :: CUSPARSE_CSR2CSC_ALG1 = 1
    enumerator :: CUSPARSE_CSR2CSC_ALG2 = 2
end enum
```

```
enum, bind(C) ! cusparseFormat_t;
    enumerator :: CUSPARSE_FORMAT_CSR     = 1
    enumerator :: CUSPARSE_FORMAT_CSC     = 2
    enumerator :: CUSPARSE_FORMAT_COO     = 3
    enumerator :: CUSPARSE_FORMAT_COO_AOS = 4
end enum
```

```
enum, bind(C) ! cusparseOrder_t;
    enumerator :: CUSPARSE_ORDER_COL = 1
    enumerator :: CUSPARSE_ORDER_ROW = 2
end enum
```

```
enum, bind(C) ! cusparseSpMVAlg_t;
    enumerator :: CUSPARSE_MV_ALG_DEFAULT = 0
    enumerator :: CUSPARSE_COOMV_ALG      = 1
    enumerator :: CUSPARSE_CSRMV_ALG1     = 2
    enumerator :: CUSPARSE_CSRMV_ALG2     = 3
end enum
```

```
enum, bind(C) ! cusparseSpMMAlg_t;
    enumerator :: CUSPARSE_MM_ALG_DEFAULT = 0
    enumerator :: CUSPARSE_COOMM_ALG1 = 1
    enumerator :: CUSPARSE_COOMM_ALG2 = 2
    enumerator :: CUSPARSE_COOMM_ALG3 = 3
    enumerator :: CUSPARSE_CSRMM_ALG1 = 4
end enum
```

```
enum, bind(C) ! cusparseIndexType_t;
    enumerator :: CUSPARSE_INDEX_16U = 1
    enumerator :: CUSPARSE_INDEX_32I = 2
    enumerator :: CUSPARSE_INDEX_64I = 3
end enum
```

5.1.1. cusparseCreate

This function initializes the cuSPARSE library and creates a handle on the cuSPARSE context. It must be called before any other cuSPARSE API function is invoked. It allocates hardware resources necessary for accessing the GPU.

```
integer(4) function cusparseCreate(handle)
  type(cusparseHandle) :: handle
```

5.1.2. cusparseDestroy

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

```
integer(4) function cusparseDestroy(handle)
  type(cusparseHandle) :: handle
```

5.1.3. cusparseGetVersion

This function returns the version number of the cuSPARSE library.

```
integer(4) function cusparseGetVersion(handle, version)
  type(cusparseHandle) :: handle
  integer(c_int) :: version
```

5.1.4. cusparseSetStream

This function sets the stream to be used by the cuSPARSE library to execute its routines.

```
integer(4) function cusparseSetStream(handle, stream)
  type(cusparseHandle) :: handle
  integer(cuda_stream_kind) :: stream
```

5.1.5. cusparseGetStream

This function gets the stream used by the cuSPARSE library to execute its routines. If the cuSPARSE library stream is not set, all kernels use the default NULL stream.

```
integer(4) function cusparseGetStream(handle, stream)
  type(cusparseHandle) :: handle
  integer(cuda_stream_kind) :: stream
```

5.1.6. cusparseGetPointerMode

This function obtains the pointer mode used by the cuSPARSE library. Please see section 1.6 for more details on pointer modes.

```
integer(4) function cusparseGetPointerMode(handle, mode)
  type(cusparseHandle) :: handle
  integer(c_int) :: mode
```

5.1.7. cusparseSetPointerMode

This function sets the pointer mode used by the cuSPARSE library. In these Fortran interfaces, this only has an effect when using the *_v2 interfaces. The default is for the values to be passed by reference on the host. Please see section 1.6 for more details on pointer modes.

```
integer(4) function cusparseSetPointerMode(handle, mode)
  type(cusparseHandle) :: handle
  integer(4) :: mode
```

5.1.8. cusparseCreateMatDescr

This function initializes the matrix descriptor. It sets the fields MatrixType and IndexBase to the default values CUSPARSE_MATRIX_TYPE_GENERAL and CUSPARSE_INDEX_BASE_ZERO , respectively, while leaving other fields uninitialized.

```
integer(4) function cusparseCreateMatDescr(descrA)
  type(cusparseMatDescr) :: descrA
```

5.1.9. cusparseDestroyMatDescr

This function releases the memory allocated for the matrix descriptor.

```
integer(4) function cusparseDestroyMatDescr(descrA)
  type(cusparseMatDescr) :: descrA
```

5.1.10. cusparseSetMatType

This function sets the MatrixType of the matrix descriptor descrA.

```
integer(4) function cusparseSetMatType(descrA, type)
  type(cusparseMatDescr) :: descrA
  integer(4) :: type
```

5.1.11. cusparseGetMatType

This function returns the MatrixType of the matrix descriptor descrA.

```
integer(4) function cusparseGetMatType(descrA)
  type(cusparseMatDescr) :: descrA
```

5.1.12. cusparseSetMatFillMode

This function sets the FillMode field of the matrix descriptor descrA.

```
integer(4) function cusparseSetMatFillMode(descrA, mode)
  type(cusparseMatDescr) :: descrA
  integer(4) :: mode
```

5.1.13. cusparseGetMatFillMode

This function returns the FillMode field of the matrix descriptor descrA.

```
integer(4) function cusparseGetMatFillMode(descrA)
  type(cusparseMatDescr) :: descrA
```

5.1.14. cusparseSetMatDiagType

This function sets the DiagType of the matrix descriptor descrA.

```
integer(4) function cusparseSetMatDiagType(descrA, type)
  type(cusparseMatDescr) :: descrA
  integer(4) :: type
```

5.1.15. cusparseGetMatDiagType

This function returns the DiagType of the matrix descriptor descrA.

```
integer(4) function cusparseGetMatDiagType(descrA)
  type(cusparseMatDescr) :: descrA
```

5.1.16. cusparseSetMatIndexBase

This function sets the IndexBase field of the matrix descriptor descrA.

```
integer(4) function cusparseSetMatIndexBase(descrA, base)
  type(cusparseMatDescr) :: descrA
  integer(4) :: base
```

5.1.17. cusparseGetMatIndexBase

This function returns the IndexBase field of the matrix descriptor descrA.

```
integer(4) function cusparseGetMatIndexBase(descrA)
  type(cusparseMatDescr) :: descrA
```

5.1.18. cusparseCreateSolveAnalysisInfo

This function creates and initializes the solve and analysis structure to default values.

```
integer(4) function cusparseCreateSolveAnalysisInfo(info)
  type(cusparseSolveAnalysisinfo) :: info
```

5.1.19. cusparseDestroySolveAnalysisInfo

This function destroys and releases any memory required by the structure.

```
integer(4) function cusparseDestroySolveAnalysisInfo(info)
  type(cusparseSolveAnalysisinfo) :: info
```

5.1.20. cusparseGetLevelInfo

This function returns the number of levels and the assignment of rows into the levels computed by either the csrsv_analysis, csrsm_analysis or hybsv_analysis routines.

```
integer(4) function cusparseGetLevelInfo(handle, info, nlevels, levelPtr,
levelInd)
  type(cusparseHandle) :: handle
  type(cusparseSolveAnalysisinfo) :: info
  integer(c_int) :: nlevels
  type(c_ptr) :: levelPtr
  type(c_ptr) :: levelInd
```

5.1.21. cusparseCreateHybMat

This function creates and initializes the hybA opaque data structure.

```
integer(4) function cusparseCreateHybMat(hybA)
  type(cusparseHybMat) :: hybA
```

5.1.22. cusparseDestroyHybMat

This function destroys and releases any memory required by the hybA structure.

```
integer(4) function cusparseDestroyHybMat(hybA)
  type(cusparseHybMat) :: hybA
```

5.1.23. cusparseCreateCsrsv2Info

This function creates and initializes the solve and analysis structure of csrsv2 to default values.

```
integer(4) function cusparseCreateCsrsv2Info(info)
  type(cusparseCsrsv2Info) :: info
```

5.1.24. cusparseDestroyCsrsv2Info

This function destroys and releases any memory required by the structure.

```
integer(4) function cusparseDestroyCsrsv2Info(info)
  type(cusparseCsrsv2Info) :: info
```

5.1.25. cusparseCreateCsric02Info

This function creates and initializes the solve and analysis structure of incomplete Cholesky to default values.

```
integer(4) function cusparseCreateCsric02Info(info)
  type(cusparseCsric02Info) :: info
```

5.1.26. cusparseDestroyCsric02Info

This function destroys and releases any memory required by the structure.

```
integer(4) function cusparseDestroyCsric02Info(info)
  type(cusparseCsric02Info) :: info
```

5.1.27. cusparseCreateCsrilu02Info

This function creates and initializes the solve and analysis structure of incomplete LU to default values.

```
integer(4) function cusparseCreateCsrilu02Info(info)
  type(cusparseCsrilu02Info) :: info
```

5.1.28. cusparseDestroyCsrilu02Info

This function destroys and releases any memory required by the structure.

```
integer(4) function cusparseDestroyCsrilu02Info(info)
  type(cusparseCsrilu02Info) :: info
```

5.1.29. cusparseCreateBsrsv2Info

This function creates and initializes the solve and analysis structure of bsrsv2 to default values.

```
integer(4) function cusparseCreateBsrsv2Info(info)
  type(cusparseBsrsv2Info) :: info
```

5.1.30. cusparseDestroyBsrsv2Info

This function destroys and releases any memory required by the structure.

```
integer(4) function cusparseDestroyBsrsv2Info(info)
  type(cusparseBsrsv2Info) :: info
```

5.1.31. cusparseCreateBsric02Info

This function creates and initializes the solve and analysis structure of block incomplete Cholesky to default values.

```
integer(4) function cusparseCreateBsric02Info(info)
```

```
  type(cusparseBsric02Info) :: info
```

5.1.32. cusparseDestroyBsric02Info

This function destroys and releases any memory required by the structure.

```
integer(4) function cusparseDestroyBsric02Info(info)
  type(cusparseBsric02Info) :: info
```

5.1.33. cusparseCreateBsrilu02Info

This function creates and initializes the solve and analysis structure of block incomplete LU to default values.

```
integer(4) function cusparseCreateBsrilu02Info(info)
  type(cusparseBsrilu02Info) :: info
```

5.1.34. cusparseDestroyBsrilu02Info

This function destroys and releases any memory required by the structure.

```
integer(4) function cusparseDestroyBsrilu02Info(info)
  type(cusparseBsrilu02Info) :: info
```

5.1.35. cusparseCreateBsrsm2Info

This function creates and initializes the solve and analysis structure of bsrsm2 to default values.

```
integer(4) function cusparseCreateBsrsm2Info(info)
  type(cusparseBsrsm2Info) :: info
```

5.1.36. cusparseDestroyBsrsm2Info

This function destroys and releases any memory required by the structure.

```
integer(4) function cusparseDestroyBsrsm2Info(info)
  type(cusparseBsrsm2Info) :: info
```

5.1.37. cusparseCreateCsrgemm2Info

This function creates and initializes the analysis structure of general sparse matrix-matrix multiplication.

```
integer(4) function cusparseCreateCsrgemm2Info(info)
  type(cusparseCsrgemm2Info) :: info
```

5.1.38. cusparseDestroyCsrgemm2Info

This function destroys and releases any memory required by the structure.

```
integer(4) function cusparseDestroyCsrgemm2Info(info)
  type(cusparseCsrgemm2Info) :: info
```

5.1.39. cusparseCreateColorInfo

This function creates coloring information used in calls like CSRCOLOR.

```
integer(4) function cusparseCreateColorInfo(info)
```

```
  type(cusparseColorInfo) :: info
```

5.1.40. cusparseDestroyColorInfo

This function destroys coloring information used in calls like CSRCOLOR.

```
integer(4) function cusparseDestroyColorInfo(info)
  type(cusparseColorInfo) :: info
```

5.1.41. cusparseCreateCsru2csrInfo

This function creates sorting information used in calls like CSRU2CSR.

```
integer(4) function cusparseCreateCsru2csrInfo(info)
  type(cusparseCsru2csrInfo) :: info
```

5.1.42. cusparseDestroyCsru2csrInfo

This function creates sorting information used in calls like CSRU2CSR.

```
integer(4) function cusparseDestroyCsru2csrInfo(info)
  type(cusparseCsru2csrInfo) :: info
```

5.2. CUSPARSE Level 1 Functions

This section contains interfaces for the level 1 sparse linear algebra functions that perform operations between dense and sparse vectors.

5.2.1. cusparseSaxpyi

SAXPY performs constant times a vector plus a vector. This function multiplies the vector x in sparse format by the constant alpha and adds the result to the vector y in dense format, i.e. y = y + alpha * xVal(xInd)

```
integer(4) function cusparseSaxpyi(handle, nnz, alpha, xVal, xInd, y, idxBase)
  type(cusparseHandle) :: handle
  integer :: nnz
  real(4), device :: alpha ! device or host variable
  real(4), device :: xVal(*)
  integer(4), device :: xInd(*)
  real(4), device :: y(*)
  integer :: idxBase
```

5.2.2. cusparseDaxpyi

DAXPY performs constant times a vector plus a vector. This function multiplies the vector x in sparse format by the constant alpha and adds the result to the vector y in dense format, i.e. y = y + alpha * xVal(xInd)

```
integer(4) function cusparseDaxpyi(handle, nnz, alpha, xVal, xInd, y, idxBase)
  type(cusparseHandle) :: handle
  integer :: nnz
  real(8), device :: alpha ! device or host variable
  real(8), device :: xVal(*)
  integer(4), device :: xInd(*)
  real(8), device :: y(*)
  integer :: idxBase
```

5.2.3. cusparseCaxpyi

CAXPY performs constant times a vector plus a vector. This function multiplies the vector x in sparse format by the constant alpha and adds the result to the vector y in dense format, i.e. y = y + alpha * xVal(xInd)

```
integer(4) function cusparseCaxpyi(handle, nnz, alpha, xVal, xInd, y, idxBase)
  type(cusparseHandle) :: handle
  integer :: nnz
  complex(4), device :: alpha ! device or host variable
  complex(4), device :: xVal(*)
  integer(4), device :: xInd(*)
  complex(4), device :: y(*)
  integer :: idxBase
```

5.2.4. cusparseZaxpyi

ZAXPY performs constant times a vector plus a vector. This function multiplies the vector x in sparse format by the constant alpha and adds the result to the vector y in dense format, i.e. y = y + alpha * xVal(xInd)

```
integer(4) function cusparseZaxpyi(handle, nnz, alpha, xVal, xInd, y, idxBase)
  type(cusparseHandle) :: handle
  integer :: nnz
  complex(8), device :: alpha ! device or host variable
  complex(8), device :: xVal(*)
  integer(4), device :: xInd(*)
  complex(8), device :: y(*)
  integer :: idxBase
```

5.2.5. cusparseSdoti

SDOT forms the dot product of two vectors. This function returns the dot product of a vector x in sparse format and vector y in dense format, i.e. res = sum(y * xVal(xInd))

```
integer(4) function cusparseSdoti(handle, nnz, xVal, xInd, y, res, idxBase)
  type(cusparseHandle) :: handle
  integer :: nnz
  real(4), device :: xVal(*)
  integer(4), device :: xInd(*)
  real(4), device :: y(*)
  real(4), device :: res ! device or host variable
  integer :: idxBase
```

5.2.6. cusparseDdoti

DDOT forms the dot product of two vectors. This function returns the dot product of a vector x in sparse format and vector y in dense format, i.e. res = sum(y * xVal(xInd))

```
integer(4) function cusparseDdoti(handle, nnz, xVal, xInd, y, res, idxBase)
  type(cusparseHandle) :: handle
  integer :: nnz
  real(8), device :: xVal(*)
  integer(4), device :: xInd(*)
  real(8), device :: y(*)
  real(8), device :: res ! device or host variable
  integer :: idxBase
```

5.2.7. cusparseCdoti

CDOT forms the dot product of two vectors. This function returns the dot product of a vector x in sparse format and vector y in dense format, i.e. res = sum(y * xVal(xInd))

```
integer(4) function cusparseCdoti(handle, nnz, xVal, xInd, y, res, idxBase)
  type(cusparseHandle) :: handle
  integer :: nnz
  complex(4), device :: xVal(*)
  integer(4), device :: xInd(*)
  complex(4), device :: y(*)
  complex(4), device :: res ! device or host variable
  integer :: idxBase
```

5.2.8. cusparseZdoti

ZDOT forms the dot product of two vectors. This function returns the dot product of a vector x in sparse format and vector y in dense format, i.e. res = sum(y * xVal(xInd))

```
integer(4) function cusparseZdoti(handle, nnz, xVal, xInd, y, res, idxBase)
  type(cusparseHandle) :: handle
  integer :: nnz
  complex(8), device :: xVal(*)
  integer(4), device :: xInd(*)
  complex(8), device :: y(*)
  complex(8), device :: res ! device or host variable
  integer :: idxBase
```

5.2.9. cusparseCdotci

CDOTC forms the dot product of two vectors, conjugating the first vector. This function returns the dot product of a vector x in sparse format and vector y in dense format, i.e. res = sum(y * conjg(xVal(xInd))

```
integer(4) function cusparseCdotci(handle, nnz, xVal, xInd, y, res, idxBase)
  type(cusparseHandle) :: handle
  integer :: nnz
  complex(4), device :: xVal(*)
  integer(4), device :: xInd(*)
  complex(4), device :: y(*)
  complex(4), device :: res ! device or host variable
  integer :: idxBase
```

5.2.10. cusparseZdotci

ZDOTC forms the dot product of two vectors, conjugating the first vector. This function returns the dot product of a vector x in sparse format and vector y in dense format, i.e. res = sum(y * conjg(xVal(xInd))

```
integer(4) function cusparseZdotci(handle, nnz, xVal, xInd, y, res, idxBase)
  type(cusparseHandle) :: handle
  integer :: nnz
  complex(8), device :: xVal(*)
  integer(4), device :: xInd(*)
  complex(8), device :: y(*)
  complex(8), device :: res ! device or host variable
  integer :: idxBase
```

5.2.11. cusparseSgthr

This function gathers the elements of the vector y listed in the index array xInd into data array xVal, i.e. xVal = y(xInd) Fortran programmers should normally use idxBase == 1.

```
integer(4) function cusparseSgthr(handle, nnz, y, xVal, xInd, idxBase)
  type(cusparseHandle) :: handle
  integer(4) :: nnz
  real(4), device :: y(*)
  real(4), device :: xVal(*)
  integer(4), device :: xInd(*)
  integer(4) :: idxBase
```

5.2.12. cusparseDgthr

This function gathers the elements of the vector y listed in the index array xInd into data array xVal, i.e. xVal = y(xInd) Fortran programmers should normally use idxBase == 1.

```
integer(4) function cusparseDgthr(handle, nnz, y, xVal, xInd, idxBase)
  type(cusparseHandle) :: handle
  integer(4) :: nnz
  real(8), device :: y(*)
  real(8), device :: xVal(*)
  integer(4), device :: xInd(*)
  integer(4) :: idxBase
```

5.2.13. cusparseCgthr

This function gathers the elements of the vector y listed in the index array xInd into data array xVal, i.e. xVal = y(xInd) Fortran programmers should normally use idxBase == 1.

```
integer(4) function cusparseCgthr(handle, nnz, y, xVal, xInd, idxBase)
  type(cusparseHandle) :: handle
  integer(4) :: nnz
  complex(4), device :: y(*)
  complex(4), device :: xVal(*)
  integer(4), device :: xInd(*)
  integer(4) :: idxBase
```

5.2.14. cusparseZgthr

This function gathers the elements of the vector y listed in the index array xInd into data array xVal, i.e. xVal = y(xInd) Fortran programmers should normally use idxBase == 1.

```
integer(4) function cusparseZgthr(handle, nnz, y, xVal, xInd, idxBase)
  type(cusparseHandle) :: handle
  integer(4) :: nnz
  complex(8), device :: y(*)
  complex(8), device :: xVal(*)
  integer(4), device :: xInd(*)
  integer(4) :: idxBase
```

5.2.15. cusparseSgthrz

This function gathers the elements of the vector y listed in the index array xInd into data array xVal, i.e. xVal = y(xInd); y(xInd) = 0.0 Fortran programmers should normally use idxBase == 1.

```
integer(4) function cusparseSgthrz(handle, nnz, y, xVal, xInd, idxBase)
  type(cusparseHandle) :: handle
```

```
  integer(4) :: nnz
  real(4), device :: y(*)
  real(4), device :: xVal(*)
  integer(4), device :: xInd(*)
  integer(4) :: idxBase
```

5.2.16. cusparseDgthrz

This function gathers the elements of the vector y listed in the index array xInd into data array xVal, i.e. xVal = y(xInd); y(xInd) = 0.0 Fortran programmers should normally use idxBase == 1.

```
integer(4) function cusparseDgthrz(handle, nnz, y, xVal, xInd, idxBase)
  type(cusparseHandle) :: handle
  integer(4) :: nnz
  real(8), device :: y(*)
  real(8), device :: xVal(*)
  integer(4), device :: xInd(*)
  integer(4) :: idxBase
```

5.2.17. cusparseCgthrz

This function gathers the elements of the vector y listed in the index array xInd into data array xVal, i.e. xVal = y(xInd); y(xInd) = 0.0 Fortran programmers should normally use idxBase == 1.

```
integer(4) function cusparseCgthrz(handle, nnz, y, xVal, xInd, idxBase)
  type(cusparseHandle) :: handle
  integer(4) :: nnz
  complex(4), device :: y(*)
  complex(4), device :: xVal(*)
  integer(4), device :: xInd(*)
  integer(4) :: idxBase
```

5.2.18. cusparseZgthrz

This function gathers the elements of the vector y listed in the index array xInd into data array xVal, i.e. xVal = y(xInd); y(xInd) = 0.0 Fortran programmers should normally use idxBase == 1.

```
integer(4) function cusparseZgthrz(handle, nnz, y, xVal, xInd, idxBase)
  type(cusparseHandle) :: handle
  integer(4) :: nnz
  complex(8), device :: y(*)
  complex(8), device :: xVal(*)
  integer(4), device :: xInd(*)
  integer(4) :: idxBase
```

5.2.19. cusparseSsctr

This function scatters the elements of the dense format vector x into the vector y in sparse format, i.e. y(xInd) = x Fortran programmers should normally use idxBase == 1.

```
integer(4) function cusparseSsctr(handle, nnz, xVal, xInd, y, idxBase)
  type(cusparseHandle) :: handle
  integer(4) :: nnz
  real(4), device :: xVal(*)
  integer(4), device :: xInd(*)
  real(4), device :: y(*)
  integer(4) :: idxBase
```

5.2.20. cusparseDsctr

This function scatters the elements of the dense format vector x into the vector y in sparse format, i.e. y(xInd) = x Fortran programmers should normally use idxBase == 1.

```
integer(4) function cusparseDsctr(handle, nnz, xVal, xInd, y, idxBase)
  type(cusparseHandle) :: handle
  integer(4) :: nnz
  real(8), device :: xVal(*)
  integer(4), device :: xInd(*)
  real(8), device :: y(*)
  integer(4) :: idxBase
```

5.2.21. cusparseCsctr

This function scatters the elements of the dense format vector x into the vector y in sparse format, i.e. y(xInd) = x Fortran programmers should normally use idxBase == 1.

```
integer(4) function cusparseCsctr(handle, nnz, xVal, xInd, y, idxBase)
  type(cusparseHandle) :: handle
  integer(4) :: nnz
  complex(4), device :: xVal(*)
  integer(4), device :: xInd(*)
  complex(4), device :: y(*)
  integer(4) :: idxBase
```

5.2.22. cusparseZsctr

This function scatters the elements of the dense format vector x into the vector y in sparse format, i.e. y(xInd) = x Fortran programmers should normally use idxBase == 1.

```
integer(4) function cusparseZsctr(handle, nnz, xVal, xInd, y, idxBase)
  type(cusparseHandle) :: handle
  integer(4) :: nnz
  complex(8), device :: xVal(*)
  integer(4), device :: xInd(*)
  complex(8), device :: y(*)
  integer(4) :: idxBase
```

5.2.23. cusparseSroti

SROT applies a plane rotation. X is a sparse vector and Y is dense.

```
integer(4) function cusparseSroti(handle, nnz, xVal, xInd, y, c, s, idxBase)
  type(cusparseHandle) :: handle
  integer :: nnz
  real(4), device :: xVal(*)
  integer(4), device :: xInd(*)
  real(4), device :: y(*)
  real(4), device :: c, s ! device or host variable
  integer :: idxBase
```

5.2.24. cusparseDroti

DROT applies a plane rotation. X is a sparse vector and Y is dense.

```
integer(4) function cusparseDroti(handle, nnz, xVal, xInd, y, c, s, idxBase)
  type(cusparseHandle) :: handle
  integer :: nnz
  real(8), device :: xVal(*)
  integer(4), device :: xInd(*)
```

```
  real(8), device :: y(*)
  real(8), device :: c, s ! device or host variable
  integer :: idxBase
```

5.3. CUSPARSE Level 2 Functions

This section contains interfaces for the level 2 sparse linear algebra functions that perform operations between sparse matrices and dense vectors.

5.3.1. cusparseSbsrmv

BSRMV performs one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A**T*x + beta*y, where alpha and beta are scalars, x and y are vectors and A is an (mb*blockDim) x (nb*blockDim) sparse matrix that is defined in BSR storage format by the three arrays bsrVal, bsrRowPtr, and bsrColInd

```
integer(4) function cusparseSbsrmv(handle, dir, trans, mb, nb, nnzb, alpha,
descr, bsrVal, bsrRowPtr, bsrColInd, blockDim, x, beta, y)
  type(cusparseHandle) :: handle
  integer :: dir
  integer :: trans
  integer :: mb, nb, nnzb
  real(4), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descr
  real(4), device :: bsrVal(*)
  integer(4), device :: bsrRowPtr(*)
  integer(4), device :: bsrColInd(*)
  integer :: blockDim
  real(4), device :: x(*)
  real(4), device :: beta ! device or host variable
  real(4), device :: y(*)
```

5.3.2. cusparseDbsrmv

BSRMV performs one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A**T*x + beta*y, where alpha and beta are scalars, x and y are vectors and A is an (mb*blockDim) x (nb*blockDim) sparse matrix that is defined in BSR storage format by the three arrays bsrVal, bsrRowPtr, and bsrColInd

```
integer(4) function cusparseDbsrmv(handle, dir, trans, mb, nb, nnzb, alpha,
descr, bsrVal, bsrRowPtr, bsrColInd, blockDim, x, beta, y)
  type(cusparseHandle) :: handle
  integer :: dir
  integer :: trans
  integer :: mb, nb, nnzb
  real(8), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descr
  real(8), device :: bsrVal(*)
  integer(4), device :: bsrRowPtr(*)
  integer(4), device :: bsrColInd(*)
  integer :: blockDim
  real(8), device :: x(*)
  real(8), device :: beta ! device or host variable
  real(8), device :: y(*)
```

5.3.3. cusparseCbsrmv

BSRMV performs one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A**T*x + beta*y, where alpha and beta are scalars, x and y are vectors and A is an (mb*blockDim) x (nb*blockDim) sparse matrix that is defined in BSR storage format by the three arrays bsrVal, bsrRowPtr, and bsrColInd

```
integer(4) function cusparseCbsrmv(handle, dir, trans, mb, nb, nnzb, alpha,
descr, bsrVal, bsrRowPtr, bsrColInd, blockDim, x, beta, y)
  type(cusparseHandle) :: handle
  integer :: dir
  integer :: trans
  integer :: mb, nb, nnzb
  complex(4), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descr
  complex(4), device :: bsrVal(*)
  integer(4), device :: bsrRowPtr(*)
  integer(4), device :: bsrColInd(*)
  integer :: blockDim
  complex(4), device :: x(*)
  complex(4), device :: beta ! device or host variable
  complex(4), device :: y(*)
```

5.3.4. cusparseZbsrmv

BSRMV performs one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A**T*x + beta*y, where alpha and beta are scalars, x and y are vectors and A is an (mb*blockDim) x (nb*blockDim) sparse matrix that is defined in BSR storage format by the three arrays bsrVal, bsrRowPtr, and bsrColInd

```
integer(4) function cusparseZbsrmv(handle, dir, trans, mb, nb, nnzb, alpha,
descr, bsrVal, bsrRowPtr, bsrColInd, blockDim, x, beta, y)
  type(cusparseHandle) :: handle
  integer :: dir
  integer :: trans
  integer :: mb, nb, nnzb
  complex(8), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descr
  complex(8), device :: bsrVal(*)
  integer(4), device :: bsrRowPtr(*)
  integer(4), device :: bsrColInd(*)
  integer :: blockDim
  complex(8), device :: x(*)
  complex(8), device :: beta ! device or host variable
  complex(8), device :: y(*)
```

5.3.5. cusparseSbsrxmv

BSRXMV performs a BSRMV and a mask operation.

```
integer(4) function cusparseSbsrxmv(handle, dir, trans, sizeOfMask, mb, nb,
nnzb, alpha, descr, bsrVal, bsrMaskPtr, bsrRowPtr, bsrEndPtr, bsrColInd,
blockDim, x, beta, y)
  type(cusparseHandle) :: handle
  integer :: dir
  integer :: trans
  integer :: sizeOfMask
  integer :: mb, nb, nnzb
  real(4), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descr
  real(4), device :: bsrVal(*)
  integer(4), device :: bsrMaskPtr(*), bsrRowPtr(*), bsrEndPtr(*)
```

```
  integer(4), device :: bsrColInd(*)
  integer :: blockDim
  real(4), device :: x(*)
  real(4), device :: beta ! device or host variable
  real(4), device :: y(*)
```

5.3.6. cusparseDbsrxmv

BSRXMV performs a BSRMV and a mask operation.

```
integer(4) function cusparseDbsrxmv(handle, dir, trans, sizeOfMask, mb, nb,
nnzb, alpha, descr, bsrVal, bsrMaskPtr, bsrRowPtr, bsrEndPtr, bsrColInd,
blockDim, x, beta, y)
  type(cusparseHandle) :: handle
  integer :: dir
  integer :: trans
  integer :: sizeOfMask
  integer :: mb, nb, nnzb
  real(8), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descr
  real(8), device :: bsrVal(*)
  integer(4), device :: bsrMaskPtr(*), bsrRowPtr(*), bsrEndPtr(*)
  integer(4), device :: bsrColInd(*)
  integer :: blockDim
  real(8), device :: x(*)
  real(8), device :: beta ! device or host variable
  real(8), device :: y(*)
```

5.3.7. cusparseCbsrxmv

BSRXMV performs a BSRMV and a mask operation.

```
integer(4) function cusparseCbsrxmv(handle, dir, trans, sizeOfMask, mb, nb,
nnzb, alpha, descr, bsrVal, bsrMaskPtr, bsrRowPtr, bsrEndPtr, bsrColInd,
blockDim, x, beta, y)
  type(cusparseHandle) :: handle
  integer :: dir
  integer :: trans
  integer :: sizeOfMask
  integer :: mb, nb, nnzb
  complex(4), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descr
  complex(4), device :: bsrVal(*)
  integer(4), device :: bsrMaskPtr(*), bsrRowPtr(*), bsrEndPtr(*)
  integer(4), device :: bsrColInd(*)
  integer :: blockDim
  complex(4), device :: x(*)
  complex(4), device :: beta ! device or host variable
  complex(4), device :: y(*)
```

5.3.8. cusparseZbsrxmv

BSRXMV performs a BSRMV and a mask operation.

```
integer(4) function cusparseZbsrxmv(handle, dir, trans, sizeOfMask, mb, nb,
nnzb, alpha, descr, bsrVal, bsrMaskPtr, bsrRowPtr, bsrEndPtr, bsrColInd,
blockDim, x, beta, y)
  type(cusparseHandle) :: handle
  integer :: dir
  integer :: trans
  integer :: sizeOfMask
  integer :: mb, nb, nnzb
  complex(8), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descr
  complex(8), device :: bsrVal(*)
```

```
  integer(4), device :: bsrMaskPtr(*), bsrRowPtr(*), bsrEndPtr(*)
  integer(4), device :: bsrColInd(*)
  integer :: blockDim
  complex(8), device :: x(*)
  complex(8), device :: beta ! device or host variable
  complex(8), device :: y(*)
```

5.3.9. cusparseScsrmv

CSRMV performs one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A**T*x + beta*y, where alpha and beta are scalars, x and y are vectors and A is an m x n sparse matrix that is defined in CSR storage format by the three arrays csrValA, csrRowPtrA, and csrColIndA

```
integer(4) function cusparseScsrmv(handle, trans, m, n, nnz, alpha, descr,
csrVal, csrRowPtr, csrColInd, x, beta, y)
  type(cusparseHandle) :: handle
  integer :: trans
  integer :: m, n, nnz
  real(4), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descr
  real(4), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*)
  integer(4), device :: csrColInd(*)
  real(4), device :: x(*)
  real(4), device :: beta ! device or host variable
  real(4), device :: y(*)
```

5.3.10. cusparseDcsrmv

CSRMV performs one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A**T*x + beta*y, where alpha and beta are scalars, x and y are vectors and A is an m x n sparse matrix that is defined in CSR storage format by the three arrays csrValA, csrRowPtrA, and csrColIndA

```
integer(4) function cusparseDcsrmv(handle, trans, m, n, nnz, alpha, descr,
csrVal, csrRowPtr, csrColInd, x, beta, y)
  type(cusparseHandle) :: handle
  integer :: trans
  integer :: m, n, nnz
  real(8), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descr
  real(8), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*)
  integer(4), device :: csrColInd(*)
  real(8), device :: x(*)
  real(8), device :: beta ! device or host variable
  real(8), device :: y(*)
```

5.3.11. cusparseCcsrmv

CSRMV performs one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A**T*x + beta*y, where alpha and beta are scalars, x and y are vectors and A is an m x n sparse matrix that is defined in CSR storage format by the three arrays csrValA, csrRowPtrA, and csrColIndA

```
integer(4) function cusparseCcsrmv(handle, trans, m, n, nnz, alpha, descr,
csrVal, csrRowPtr, csrColInd, x, beta, y)
  type(cusparseHandle) :: handle
  integer :: trans
  integer :: m, n, nnz
```

```
  complex(4), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descr
  complex(4), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*)
  integer(4), device :: csrColInd(*)
  complex(4), device :: x(*)
  complex(4), device :: beta ! device or host variable
  complex(4), device :: y(*)
```

5.3.12. cusparseZcsrmv

CSRMV performs one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A**T*x + beta*y, where alpha and beta are scalars, x and y are vectors and A is an m x n sparse matrix that is defined in CSR storage format by the three arrays csrValA, csrRowPtrA, and csrColIndA

```
integer(4) function cusparseZcsrmv(handle, trans, m, n, nnz, alpha, descr,
csrVal, csrRowPtr, csrColInd, x, beta, y)
  type(cusparseHandle) :: handle
  integer :: trans
  integer :: m, n, nnz
  complex(8), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descr
  complex(8), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*)
  integer(4), device :: csrColInd(*)
  complex(8), device :: x(*)
  complex(8), device :: beta ! device or host variable
  complex(8), device :: y(*)
```

5.3.13. cusparseScsrsv_analysis

This function performs the analysis phase of csrsv.

```
integer(4) function cusparseScsrsv_analysis(handle, trans, m, nnz, descr,
csrVal, csrRowPtr, csrColInd, info)
  type(cusparseHandle) :: handle
  integer(4) :: trans
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descr
  real(4), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*)
  integer(4), device :: csrColInd(*)
  type(cusparseSolveAnalysisInfo) :: info
```

5.3.14. cusparseDcsrsv_analysis

This function performs the analysis phase of csrsv.

```
integer(4) function cusparseDcsrsv_analysis(handle, trans, m, nnz, descr,
csrVal, csrRowPtr, csrColInd, info)
  type(cusparseHandle) :: handle
  integer(4) :: trans
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descr
  real(8), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*)
  integer(4), device :: csrColInd(*)
  type(cusparseSolveAnalysisInfo) :: info
```

5.3.15. cusparseCcsrsv_analysis

This function performs the analysis phase of csrsv.

```
integer(4) function cusparseCcsrsv_analysis(handle, trans, m, nnz, descr,
csrVal, csrRowPtr, csrColInd, info)
  type(cusparseHandle) :: handle
  integer(4) :: trans
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descr
  complex(4), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*)
  integer(4), device :: csrColInd(*)
  type(cusparseSolveAnalysisInfo) :: info
```

5.3.16. cusparseZcsrsv_analysis

This function performs the analysis phase of csrsv.

```
integer(4) function cusparseZcsrsv_analysis(handle, trans, m, nnz, descr,
csrVal, csrRowPtr, csrColInd, info)
  type(cusparseHandle) :: handle
  integer(4) :: trans
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descr
  complex(8), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*)
  integer(4), device :: csrColInd(*)
  type(cusparseSolveAnalysisInfo) :: info
```

5.3.17. cusparseScsrsv_solve

This function performs the solve phase of csrsv.

```
integer(4) function cusparseScsrsv_solve(handle, trans, m, alpha, descr,
csrVal, csrRowPtr, csrColInd, info, x, y)
  type(cusparseHandle) :: handle
  integer :: trans
  integer :: m
  real(4), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descr
  real(4), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*)
  integer(4), device :: csrColInd(*)
  type(cusparseSolveAnalysisInfo) :: info
  real(4), device :: x(*)
  real(4), device :: y(*)
```

5.3.18. cusparseDcsrsv_solve

This function performs the solve phase of csrsv.

```
integer(4) function cusparseDcsrsv_solve(handle, trans, m, alpha, descr,
csrVal, csrRowPtr, csrColInd, info, x, y)
  type(cusparseHandle) :: handle
  integer :: trans
  integer :: m
  real(8), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descr
  real(8), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*)
  integer(4), device :: csrColInd(*)
  type(cusparseSolveAnalysisInfo) :: info
```

```
  real(8), device :: x(*)
  real(8), device :: y(*)
```

5.3.19. cusparseCcsrsv_solve

This function performs the solve phase of csrsv.

```
integer(4) function cusparseCcsrsv_solve(handle, trans, m, alpha, descr,
csrVal, csrRowPtr, csrColInd, info, x, y)
  type(cusparseHandle) :: handle
  integer :: trans
  integer :: m
  complex(4), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descr
  complex(4), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*)
  integer(4), device :: csrColInd(*)
  type(cusparseSolveAnalysisInfo) :: info
  complex(4), device :: x(*)
  complex(4), device :: y(*)
```

5.3.20. cusparseZcsrsv_solve

This function performs the solve phase of csrsv.

```
integer(4) function cusparseZcsrsv_solve(handle, trans, m, alpha, descr,
csrVal, csrRowPtr, csrColInd, info, x, y)
  type(cusparseHandle) :: handle
  integer :: trans
  integer :: m
  complex(8), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descr
  complex(8), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*)
  integer(4), device :: csrColInd(*)
  type(cusparseSolveAnalysisInfo) :: info
  complex(8), device :: x(*)
  complex(8), device :: y(*)
```

5.3.21. cusparseShybmv

HYBMV performs the matrix-vector operations y := alpha*A*x + beta*y where alpha and beta are scalars, x and y are vectors and A is an m x n sparse matrix that is defined in the HYB storage format.

```
integer(4) function cusparseShybmv(handle, trans, alpha, descr, hyb, x, beta,
y)
  type(cusparseHandle) :: handle
  integer :: trans
  real(4), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descr
  type(cusparseHybMat) :: hyb
  real(4), device :: x(*)
  real(4), device :: beta ! device or host variable
  real(4), device :: y(*)
```

5.3.22. cusparseDhybmv

HYBMV performs the matrix-vector operations y := alpha*A*x + beta*y where alpha and beta are scalars, x and y are vectors and A is an m x n sparse matrix that is defined in the HYB storage format.

```
integer(4) function cusparseDhybmv(handle, trans, alpha, descr, hyb, x, beta,
y)
  type(cusparseHandle) :: handle
  integer :: trans
  real(8), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descr
  type(cusparseHybMat) :: hyb
  real(8), device :: x(*)
  real(8), device :: beta ! device or host variable
  real(8), device :: y(*)
```

5.3.23. cusparseChybmv

HYBMV performs the matrix-vector operations y := alpha*A*x + beta*y where alpha and beta are scalars, x and y are vectors and A is an m x n sparse matrix that is defined in the HYB storage format.

```
integer(4) function cusparseChybmv(handle, trans, alpha, descr, hyb, x, beta,
y)
  type(cusparseHandle) :: handle
  integer :: trans
  complex(4), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descr
  type(cusparseHybMat) :: hyb
  complex(4), device :: x(*)
  complex(4), device :: beta ! device or host variable
  complex(4), device :: y(*)
```

5.3.24. cusparseZhybmv

HYBMV performs the matrix-vector operations y := alpha*A*x + beta*y where alpha and beta are scalars, x and y are vectors and A is an m x n sparse matrix that is defined in the HYB storage format.

```
integer(4) function cusparseZhybmv(handle, trans, alpha, descr, hyb, x, beta,
y)
  type(cusparseHandle) :: handle
  integer :: trans
  complex(8), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descr
  type(cusparseHybMat) :: hyb
  complex(8), device :: x(*)
  complex(8), device :: beta ! device or host variable
  complex(8), device :: y(*)
```

5.3.25. cusparseShybsv_analysis

This function performs the analysis phase of hybsv.

```
integer(4) function cusparseShybsv_analysis(handle, trans, descr, hyb, info)
  type(cusparseHandle) :: handle
  integer(4) :: trans
  type(cusparseMatDescr) :: descr
  type(cusparseHybMat) :: hyb
  type(cusparseSolveAnalysisInfo) :: info
```

5.3.26. cusparseDhybsv_analysis

This function performs the analysis phase of hybsv.

```
integer(4) function cusparseDhybsv_analysis(handle, trans, descr, hyb, info)
  type(cusparseHandle) :: handle
  integer(4) :: trans
  type(cusparseMatDescr) :: descr
  type(cusparseHybMat) :: hyb
  type(cusparseSolveAnalysisInfo) :: info
```

5.3.27. cusparseChybsv_analysis

This function performs the analysis phase of hybsv.

```
integer(4) function cusparseChybsv_analysis(handle, trans, descr, hyb, info)
  type(cusparseHandle) :: handle
  integer(4) :: trans
  type(cusparseMatDescr) :: descr
  type(cusparseHybMat) :: hyb
  type(cusparseSolveAnalysisInfo) :: info
```

5.3.28. cusparseZhybsv_analysis

This function performs the analysis phase of hybsv.

```
integer(4) function cusparseZhybsv_analysis(handle, trans, descr, hyb, info)
  type(cusparseHandle) :: handle
  integer(4) :: trans
  type(cusparseMatDescr) :: descr
  type(cusparseHybMat) :: hyb
  type(cusparseSolveAnalysisInfo) :: info
```

5.3.29. cusparseShybsv_solve

This function performs the solve phase of hybsv.

```
integer(4) function cusparseShybsv_solve(handle, trans, alpha, descr, hyb,
info, x, y)
  type(cusparseHandle) :: handle
  integer :: trans
  real(4), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descr
  type(cusparseHybMat) :: hyb
  type(cusparseSolveAnalysisInfo) :: info
  real(4), device :: x(*)
  real(4), device :: y(*)
```

5.3.30. cusparseDhybsv_solve

This function performs the solve phase of hybsv.

```
integer(4) function cusparseDhybsv_solve(handle, trans, alpha, descr, hyb,
info, x, y)
  type(cusparseHandle) :: handle
  integer :: trans
  real(8), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descr
  type(cusparseHybMat) :: hyb
  type(cusparseSolveAnalysisInfo) :: info
  real(8), device :: x(*)
  real(8), device :: y(*)
```

5.3.31. cusparseChybsv_solve

This function performs the solve phase of hybsv.

```
integer(4) function cusparseChybsv_solve(handle, trans, alpha, descr, hyb,
info, x, y)
  type(cusparseHandle) :: handle
  integer :: trans
  complex(4), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descr
  type(cusparseHybMat) :: hyb
  type(cusparseSolveAnalysisInfo) :: info
  complex(4), device :: x(*)
  complex(4), device :: y(*)
```

5.3.32. cusparseZhybsv_solve

This function performs the solve phase of hybsv.

```
integer(4) function cusparseZhybsv_solve(handle, trans, alpha, descr, hyb,
info, x, y)
  type(cusparseHandle) :: handle
  integer :: trans
  complex(8), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descr
  type(cusparseHybMat) :: hyb
  type(cusparseSolveAnalysisInfo) :: info
  complex(8), device :: x(*)
  complex(8), device :: y(*)
```

5.3.33. cusparseSbsrsv2_bufferSize

This function returns the size of the buffer used in bsrsv2.

```
integer(4) function cusparseSbsrsv2_bufferSize(handle, dirA, transA, mb, nnzb,
descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: dirA, transA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  real(4), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsrsv2Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

5.3.34. cusparseDbsrsv2_bufferSize

This function returns the size of the buffer used in bsrsv2.

```
integer(4) function cusparseDbsrsv2_bufferSize(handle, dirA, transA, mb, nnzb,
descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: dirA, transA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  real(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsrsv2Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

5.3.35. cusparseCbsrsv2_bufferSize

This function returns the size of the buffer used in bsrsv2.

```
integer(4) function cusparseCbsrsv2_bufferSize(handle, dirA, transA, mb, nnzb,
descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: dirA, transA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  complex(4), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsrsv2Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

5.3.36. cusparseZbsrsv2_bufferSize

This function returns the size of the buffer used in bsrsv2.

```
integer(4) function cusparseZbsrsv2_bufferSize(handle, dirA, transA, mb, nnzb,
descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: dirA, transA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  complex(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsrsv2Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

5.3.37. cusparseSbsrsv2_analysis

This function performs the analysis phase of bsrsv2.

```
integer(4) function cusparseSbsrsv2_analysis(handle, dirA, transA, mb, nnzb,
descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dirA, transA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  real(4), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsrsv2Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

5.3.38. cusparseDbsrsv2_analysis

This function performs the analysis phase of bsrsv2.

```
integer(4) function cusparseDbsrsv2_analysis(handle, dirA, transA, mb, nnzb,
descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dirA, transA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  real(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsrsv2Info) :: info
```

```
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

5.3.39. cusparseCbsrsv2_analysis

This function performs the analysis phase of bsrsv2.

```
integer(4) function cusparseCbsrsv2_analysis(handle, dirA, transA, mb, nnzb,
descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dirA, transA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  complex(4), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsrsv2Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

5.3.40. cusparseZbsrsv2_analysis

This function performs the analysis phase of bsrsv2.

```
integer(4) function cusparseZbsrsv2_analysis(handle, dirA, transA, mb, nnzb,
descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dirA, transA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  complex(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsrsv2Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

5.3.41. cusparseSbsrsv2_solve

This function performs the solve phase of bsrsv2.

```
integer(4) function cusparseSbsrsv2_solve(handle, dirA, transA, mb, nnzb,
alpha, descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, x, y, policy,
pBuffer)
  type(cusparseHandle) :: handle
  integer :: dirA, transA, mb, nnzb
  real(4), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descrA
  real(4), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer :: blockDim
  type(cusparseBsrsv2Info) :: info
  real(4), device :: x(*), y(*)
  integer :: policy
  character, device :: pBuffer(*)
```

5.3.42. cusparseDbsrsv2_solve

This function performs the solve phase of bsrsv2.

```
integer(4) function cusparseDbsrsv2_solve(handle, dirA, transA, mb, nnzb,
alpha, descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, x, y, policy,
pBuffer)
  type(cusparseHandle) :: handle
```

```
  integer :: dirA, transA, mb, nnzb
  real(8), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descrA
  real(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer :: blockDim
  type(cusparseBsrsv2Info) :: info
  real(8), device :: x(*), y(*)
  integer :: policy
  character, device :: pBuffer(*)
```

5.3.43. cusparseCbsrsv2_solve

This function performs the solve phase of bsrsv2.

```
integer(4) function cusparseCbsrsv2_solve(handle, dirA, transA, mb, nnzb,
alpha, descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, x, y, policy,
pBuffer)
  type(cusparseHandle) :: handle
  integer :: dirA, transA, mb, nnzb
  complex(4), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descrA
  complex(4), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer :: blockDim
  type(cusparseBsrsv2Info) :: info
  complex(4), device :: x(*), y(*)
  integer :: policy
  character, device :: pBuffer(*)
```

5.3.44. cusparseZbsrsv2_solve

This function performs the solve phase of bsrsv2.

```
integer(4) function cusparseZbsrsv2_solve(handle, dirA, transA, mb, nnzb,
alpha, descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, x, y, policy,
pBuffer)
  type(cusparseHandle) :: handle
  integer :: dirA, transA, mb, nnzb
  complex(8), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descrA
  complex(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer :: blockDim
  type(cusparseBsrsv2Info) :: info
  complex(8), device :: x(*), y(*)
  integer :: policy
  character, device :: pBuffer(*)
```

5.3.45. cusparseXbsrsv2_zeroPivot

This function returns an error code equal to CUSPARSE_STATUS_ZERO_PIVOT and sets position to j when A(j,j) is either structural zero or numerical zero. Otherwise, position is set to -1.

```
integer(4) function cusparseXbsrsv2_zeroPivot(handle, info, position)
  type(cusparseHandle) :: handle
  type(cusparseBsrsv2Info) :: info
  integer(4), device :: position ! device or host variable
```

5.3.46. cusparseScsrsv2_bufferSize

This function returns the size of the buffer used in csrsv2.

```
integer(4) function cusparseScsrsv2_bufferSize(handle, transA, m, nnz, descrA,
csrValA, csrRowPtrA, csrColIndA, info, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: transA, m, nnz
  type(cusparseMatDescr) :: descrA
  real(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsrsv2Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

5.3.47. cusparseDcsrsv2_bufferSize

This function returns the size of the buffer used in csrsv2.

```
integer(4) function cusparseDcsrsv2_bufferSize(handle, transA, m, nnz, descrA,
csrValA, csrRowPtrA, csrColIndA, info, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: transA, m, nnz
  type(cusparseMatDescr) :: descrA
  real(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsrsv2Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

5.3.48. cusparseCcsrsv2_bufferSize

This function returns the size of the buffer used in csrsv2.

```
integer(4) function cusparseCcsrsv2_bufferSize(handle, transA, m, nnz, descrA,
csrValA, csrRowPtrA, csrColIndA, info, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: transA, m, nnz
  type(cusparseMatDescr) :: descrA
  complex(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsrsv2Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

5.3.49. cusparseZcsrsv2_bufferSize

This function returns the size of the buffer used in csrsv2.

```
integer(4) function cusparseZcsrsv2_bufferSize(handle, transA, m, nnz, descrA,
csrValA, csrRowPtrA, csrColIndA, info, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: transA, m, nnz
  type(cusparseMatDescr) :: descrA
  complex(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsrsv2Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

5.3.50. cusparseScsrsv2_analysis

This function performs the analysis phase of csrsv2.

```
integer(4) function cusparseScsrsv2_analysis(handle, transA, m, nnz, descrA,
csrValA, csrRowPtrA, csrColIndA, info, policy, pBuffer)
```

```
  type(cusparseHandle) :: handle
  integer(4) :: transA, m, nnz
  type(cusparseMatDescr) :: descrA
  real(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsrsv2Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

5.3.51. cusparseDcsrsv2_analysis

This function performs the analysis phase of csrsv2.

```
integer(4) function cusparseDcsrsv2_analysis(handle, transA, m, nnz, descrA,
csrValA, csrRowPtrA, csrColIndA, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: transA, m, nnz
  type(cusparseMatDescr) :: descrA
  real(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsrsv2Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

5.3.52. cusparseCcsrsv2_analysis

This function performs the analysis phase of csrsv2.

```
integer(4) function cusparseCcsrsv2_analysis(handle, transA, m, nnz, descrA,
csrValA, csrRowPtrA, csrColIndA, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: transA, m, nnz
  type(cusparseMatDescr) :: descrA
  complex(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsrsv2Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

5.3.53. cusparseZcsrsv2_analysis

This function performs the analysis phase of csrsv2.

```
integer(4) function cusparseZcsrsv2_analysis(handle, transA, m, nnz, descrA,
csrValA, csrRowPtrA, csrColIndA, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: transA, m, nnz
  type(cusparseMatDescr) :: descrA
  complex(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsrsv2Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

5.3.54. cusparseScsrsv2_solve

This function performs the solve phase of csrsv2.

```
integer(4) function cusparseScsrsv2_solve(handle, transA, m, nnz, alpha,
descrA, csrValA, csrRowPtrA, csrColIndA, info, x, y, policy, pBuffer)
  type(cusparseHandle) :: handle
  real(4), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descrA
  real(4), device :: csrValA(*), x(*), y(*)
```

```
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsrsv2Info) :: info
  integer :: policy
  character, device :: pBuffer(*)
```

5.3.55. cusparseDcsrsv2_solve

This function performs the solve phase of csrsv2.

```
integer(4) function cusparseDcsrsv2_solve(handle, transA, m, nnz, alpha,
descrA, csrValA, csrRowPtrA, csrColIndA, info, x, y, policy, pBuffer)
  type(cusparseHandle) :: handle
  real(8), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descrA
  real(8), device :: csrValA(*), x(*), y(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsrsv2Info) :: info
  integer :: policy
  character, device :: pBuffer(*)
```

5.3.56. cusparseCcsrsv2_solve

This function performs the solve phase of csrsv2.

```
integer(4) function cusparseCcsrsv2_solve(handle, transA, m, nnz, alpha,
descrA, csrValA, csrRowPtrA, csrColIndA, info, x, y, policy, pBuffer)
  type(cusparseHandle) :: handle
  complex(4), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descrA
  complex(4), device :: csrValA(*), x(*), y(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsrsv2Info) :: info
  integer :: policy
  character, device :: pBuffer(*)
```

5.3.57. cusparseZcsrsv2_solve

This function performs the solve phase of csrsv2.

```
integer(4) function cusparseZcsrsv2_solve(handle, transA, m, nnz, alpha,
descrA, csrValA, csrRowPtrA, csrColIndA, info, x, y, policy, pBuffer)
  type(cusparseHandle) :: handle
  complex(8), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descrA
  complex(8), device :: csrValA(*), x(*), y(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsrsv2Info) :: info
  integer :: policy
  character, device :: pBuffer(*)
```

5.3.58. cusparseXcsrsv2_zeroPivot

This function returns an error code equal to CUSPARSE_STATUS_ZERO_PIVOT and sets position to j when A(j,j) is either structural zero or numerical zero. Otherwise, position is set to -1.

```
integer(4) function cusparseXcsrsv2_zeroPivot(handle, info, position)
  type(cusparseHandle) :: handle
  type(cusparseCsrsv2Info) :: info
  integer(4), device :: position ! device or host variable
```

5.4. CUSPARSE Level 3 Functions

This section contains interfaces for the level 3 sparse linear algebra functions that perform operations between sparse and dense matrices.

5.4.1. cusparseScsrmm

CSRMM performs one of the matrix-matrix operations C := alpha*op(A) * B + beta*C, where op(A) is one of op(A) = A or op(A) = A**T, alpha and beta are scalars. A is an m x k sparse matrix that is defined in CSR storage format by the three arrays csrValA, csrRowPtrA, and csrColIndA. B and C are dense matrices.

```
integer(4) function cusparseScsrmm(handle, transA, m, n, k, nnz, alpha, descrA,
csrValA, csrRowPtrA, csrColIndA, B, ldb, beta, C, ldc)
  type(cusparseHandle) :: handle
  integer :: transA, m, n, k, nnz
  real(4), device :: alpha, beta ! device or host variable
  type(cusparseMatDescr) :: descrA
  real(4), device :: csrValA(*), B(*), C(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  integer :: ldb, ldc
```

5.4.2. cusparseDcsrmm

CSRMM performs one of the matrix-matrix operations C := alpha*op(A) * B + beta*C, where op(A) is one of op(A) = A or op(A) = A**T, alpha and beta are scalars. A is an m x k sparse matrix that is defined in CSR storage format by the three arrays csrValA, csrRowPtrA, and csrColIndA. B and C are dense matrices.

```
integer(4) function cusparseDcsrmm(handle, transA, m, n, k, nnz, alpha, descrA,
csrValA, csrRowPtrA, csrColIndA, B, ldb, beta, C, ldc)
  type(cusparseHandle) :: handle
  integer :: transA, m, n, k, nnz
  real(8), device :: alpha, beta ! device or host variable
  type(cusparseMatDescr) :: descrA
  real(8), device :: csrValA(*), B(*), C(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  integer :: ldb, ldc
```

5.4.3. cusparseCcsrmm

CSRMM performs one of the matrix-matrix operations C := alpha*op(A) * B + beta*C, where op(A) is one of op(A) = A or op(A) = A**T, alpha and beta are scalars. A is an m x k sparse matrix that is defined in CSR storage format by the three arrays csrValA, csrRowPtrA, and csrColIndA. B and C are dense matrices.

```
integer(4) function cusparseCcsrmm(handle, transA, m, n, k, nnz, alpha, descrA,
csrValA, csrRowPtrA, csrColIndA, B, ldb, beta, C, ldc)
  type(cusparseHandle) :: handle
  integer :: transA, m, n, k, nnz
  complex(4), device :: alpha, beta ! device or host variable
  type(cusparseMatDescr) :: descrA
  complex(4), device :: csrValA(*), B(*), C(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  integer :: ldb, ldc
```

5.4.4. cusparseZcsrmm

CSRMM performs one of the matrix-matrix operations C := alpha*op(A) * B + beta*C, where op(A) is one of op(A) = A or op(A) = A**T, alpha and beta are scalars. A is an m x k sparse matrix that is defined in CSR storage format by the three arrays csrValA, csrRowPtrA, and csrColIndA. B and C are dense matrices.

```
integer(4) function cusparseZcsrmm(handle, transA, m, n, k, nnz, alpha, descrA,
csrValA, csrRowPtrA, csrColIndA, B, ldb, beta, C, ldc)
  type(cusparseHandle) :: handle
  integer :: transA, m, n, k, nnz
  complex(8), device :: alpha, beta ! device or host variable
  type(cusparseMatDescr) :: descrA
  complex(8), device :: csrValA(*), B(*), C(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  integer :: ldb, ldc
```

5.4.5. cusparseScsrmm2

CSRMM2 performs one of the matrix-matrix operations C := alpha*op(A) * op(B) + beta*C, where op(A) is one of op(X) = X or op(X) = X**T, alpha and beta are scalars. A is an m x k sparse matrix that is defined in CSR storage format by the three arrays csrValA, csrRowPtrA, and csrColIndA. B and C are dense matrices.

```
integer(4) function cusparseScsrmm2(handle, transA, transB, m, n, k, nnz,
alpha, descrA, csrValA, csrRowPtrA, csrColIndA, B, ldb, beta, C, ldc)
  type(cusparseHandle) :: handle
  integer :: transA, transB, m, n, k, nnz
  real(4), device :: alpha, beta ! device or host variable
  type(cusparseMatDescr) :: descrA
  real(4), device :: csrValA(*), B(*), C(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  integer :: ldb, ldc
```

5.4.6. cusparseDcsrmm2

CSRMM2 performs one of the matrix-matrix operations C := alpha*op(A) * op(B) + beta*C, where op(A) is one of op(X) = X or op(X) = X**T, alpha and beta are scalars. A is an m x k sparse matrix that is defined in CSR storage format by the three arrays csrValA, csrRowPtrA, and csrColIndA. B and C are dense matrices.

```
integer(4) function cusparseDcsrmm2(handle, transA, transB, m, n, k, nnz,
alpha, descrA, csrValA, csrRowPtrA, csrColIndA, B, ldb, beta, C, ldc)
  type(cusparseHandle) :: handle
  integer :: transA, transB, m, n, k, nnz
  real(8), device :: alpha, beta ! device or host variable
  type(cusparseMatDescr) :: descrA
  real(8), device :: csrValA(*), B(*), C(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  integer :: ldb, ldc
```

5.4.7. cusparseCcsrmm2

CSRMM2 performs one of the matrix-matrix operations C := alpha*op(A) * op(B) + beta*C, where op(A) is one of op(X) = X or op(X) = X**T, alpha and beta are scalars.

A is an m x k sparse matrix that is defined in CSR storage format by the three arrays csrValA, csrRowPtrA, and csrColIndA. B and C are dense matrices.

```
integer(4) function cusparseCcsrmm2(handle, transA, transB, m, n, k, nnz,
alpha, descrA, csrValA, csrRowPtrA, csrColIndA, B, ldb, beta, C, ldc)
  type(cusparseHandle) :: handle
  integer :: transA, transB, m, n, k, nnz
  complex(4), device :: alpha, beta ! device or host variable
  type(cusparseMatDescr) :: descrA
  complex(4), device :: csrValA(*), B(*), C(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  integer :: ldb, ldc
```

5.4.8. cusparseZcsrmm2

CSRMM2 performs one of the matrix-matrix operations C := alpha*op(A) * op(B) + beta*C, where op(A) is one of op(X) = X or op(X) = X**T, alpha and beta are scalars. A is an m x k sparse matrix that is defined in CSR storage format by the three arrays csrValA, csrRowPtrA, and csrColIndA. B and C are dense matrices.

```
integer(4) function cusparseZcsrmm2(handle, transA, transB, m, n, k, nnz,
alpha, descrA, csrValA, csrRowPtrA, csrColIndA, B, ldb, beta, C, ldc)
  type(cusparseHandle) :: handle
  integer :: transA, transB, m, n, k, nnz
  complex(8), device :: alpha, beta ! device or host variable
  type(cusparseMatDescr) :: descrA
  complex(8), device :: csrValA(*), B(*), C(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  integer :: ldb, ldc
```

5.4.9. cusparseScsrsm_analysis

This function performs the analysis phase of csrsm.

```
integer(4) function cusparseScsrsm_analysis(handle, transA, m, nnz, descrA,
csrValA, csrRowPtrA, csrColIndA, info)
  type(cusparseHandle) :: handle
  integer(4) :: transA, m, nnz
  type(cusparseMatDescr) :: descrA
  real(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseSolveAnalysisInfo) :: info
```

5.4.10. cusparseDcsrsm_analysis

This function performs the analysis phase of csrsm.

```
integer(4) function cusparseDcsrsm_analysis(handle, transA, m, nnz, descrA,
csrValA, csrRowPtrA, csrColIndA, info)
  type(cusparseHandle) :: handle
  integer(4) :: transA, m, nnz
  type(cusparseMatDescr) :: descrA
  real(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseSolveAnalysisInfo) :: info
```

5.4.11. cusparseCcsrsm_analysis

This function performs the analysis phase of csrsm.

```
integer(4) function cusparseCcsrsm_analysis(handle, transA, m, nnz, descrA,
csrValA, csrRowPtrA, csrColIndA, info)
  type(cusparseHandle) :: handle
```

```
  integer(4) :: transA, m, nnz
  type(cusparseMatDescr) :: descrA
  complex(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseSolveAnalysisInfo) :: info
```

5.4.12. cusparseZcsrsm_analysis

This function performs the analysis phase of csrsm.

```
integer(4) function cusparseZcsrsm_analysis(handle, transA, m, nnz, descrA,
csrValA, csrRowPtrA, csrColIndA, info)
  type(cusparseHandle) :: handle
  integer(4) :: transA, m, nnz
  type(cusparseMatDescr) :: descrA
  complex(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseSolveAnalysisInfo) :: info
```

5.4.13. cusparseScsrsm_solve

This function performs the solve phase of csrsm.

```
integer(4) function cusparseScsrsm_solve(handle, transA, m, n, alpha, descrA,
csrValA, csrRowPtrA, csrColIndA, info, X, ldx, Y, ldy)
  type(cusparseHandle) :: handle
  integer :: transA, m, n
  real(4), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descrA
  real(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseSolveAnalysisInfo) :: info
  real(4), device :: X(*), Y(*)
  integer :: ldx, ldy
```

5.4.14. cusparseDcsrsm_solve

This function performs the solve phase of csrsm.

```
integer(4) function cusparseDcsrsm_solve(handle, transA, m, n, alpha, descrA,
csrValA, csrRowPtrA, csrColIndA, info, X, ldx, Y, ldy)
  type(cusparseHandle) :: handle
  integer :: transA, m, n
  real(8), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descrA
  real(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseSolveAnalysisInfo) :: info
  real(8), device :: X(*), Y(*)
  integer :: ldx, ldy
```

5.4.15. cusparseCcsrsm_solve

This function performs the solve phase of csrsm.

```
integer(4) function cusparseCcsrsm_solve(handle, transA, m, n, alpha, descrA,
csrValA, csrRowPtrA, csrColIndA, info, X, ldx, Y, ldy)
  type(cusparseHandle) :: handle
  integer :: transA, m, n
  complex(4), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descrA
  complex(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseSolveAnalysisInfo) :: info
```

```
  complex(4), device :: X(*), Y(*)
  integer :: ldx, ldy
```

5.4.16. cusparseZcsrsm_solve

This function performs the solve phase of csrsm.

```
integer(4) function cusparseZcsrsm_solve(handle, transA, m, n, alpha, descrA,
csrValA, csrRowPtrA, csrColIndA, info, X, ldx, Y, ldy)
  type(cusparseHandle) :: handle
  integer :: transA, m, n
  complex(8), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descrA
  complex(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseSolveAnalysisInfo) :: info
  complex(8), device :: X(*), Y(*)
  integer :: ldx, ldy
```

5.4.17. cusparseSbsrmm

BSRMM performs one of the matrix-matrix operations C := alpha*op(A) * op(B) + beta*C, where op(X) is one of op(X) = X or op(X) = X**T, alpha and beta are scalars. A is an mb x kb sparse matrix that is defined in BSR storage format by the three arrays bsrValA, bsrRowPtrA, and bsrColIndA. B and C are dense matrices.

```
integer(4) function cusparseSbsrmm(handle, dirA, transA, transB, mb, n, kb,
nnzb, alpha, descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, B, ldb, beta,
C, ldc)
  type(cusparseHandle) :: handle
  integer :: dirA, transA, transB, mb, n, kb, nnzb, blockDim
  real(4), device :: alpha, beta ! device or host variable
  type(cusparseMatDescr) :: descrA
  real(4), device :: bsrValA(*), B(*), C(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer :: ldb, ldc
```

5.4.18. cusparseDbsrmm

BSRMM performs one of the matrix-matrix operations C := alpha*op(A) * op(B) + beta*C, where op(X) is one of op(X) = X or op(X) = X**T, alpha and beta are scalars. A is an mb x kb sparse matrix that is defined in BSR storage format by the three arrays bsrValA, bsrRowPtrA, and bsrColIndA. B and C are dense matrices.

```
integer(4) function cusparseDbsrmm(handle, dirA, transA, transB, mb, n, kb,
nnzb, alpha, descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, B, ldb, beta,
C, ldc)
  type(cusparseHandle) :: handle
  integer :: dirA, transA, transB, mb, n, kb, nnzb, blockDim
  real(8), device :: alpha, beta ! device or host variable
  type(cusparseMatDescr) :: descrA
  real(8), device :: bsrValA(*), B(*), C(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer :: ldb, ldc
```

5.4.19. cusparseCbsrmm

BSRMM performs one of the matrix-matrix operations C := alpha*op(A) * op(B) + beta*C, where op(X) is one of op(X) = X or op(X) = X**T, alpha and beta are scalars.

A is an mb x kb sparse matrix that is defined in BSR storage format by the three arrays bsrValA, bsrRowPtrA, and bsrColIndA. B and C are dense matrices.

```
integer(4) function cusparseCbsrmm(handle, dirA, transA, transB, mb, n, kb,
nnzb, alpha, descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, B, ldb, beta,
C, ldc)
  type(cusparseHandle) :: handle
  integer :: dirA, transA, transB, mb, n, kb, nnzb, blockDim
  complex(4), device :: alpha, beta ! device or host variable
  type(cusparseMatDescr) :: descrA
  complex(4), device :: bsrValA(*), B(*), C(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer :: ldb, ldc
```

5.4.20. cusparseZbsrmm

BSRMM performs one of the matrix-matrix operations C := alpha*op(A) * op(B) + beta*C, where op(X) is one of op(X) = X or op(X) = X**T, alpha and beta are scalars. A is an mb x kb sparse matrix that is defined in BSR storage format by the three arrays bsrValA, bsrRowPtrA, and bsrColIndA. B and C are dense matrices.

```
integer(4) function cusparseZbsrmm(handle, dirA, transA, transB, mb, n, kb,
nnzb, alpha, descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, B, ldb, beta,
C, ldc)
  type(cusparseHandle) :: handle
  integer :: dirA, transA, transB, mb, n, kb, nnzb, blockDim
  complex(8), device :: alpha, beta ! device or host variable
  type(cusparseMatDescr) :: descrA
  complex(8), device :: bsrValA(*), B(*), C(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer :: ldb, ldc
```

5.4.21. cusparseSbsrsm2_bufferSize

This function returns the size of the buffer used in bsrsm2.

```
integer(4) function cusparseSbsrsm2_bufferSize(handle, dirA, transA, transX,
mb, n, nnzb, descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info,
pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: dirA, transA, transX, mb, n, nnzb, blockDim
  type(cusparseMatDescr) :: descrA
  real(4), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  type(cusparseBsrsm2Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

5.4.22. cusparseDbsrsm2_bufferSize

This function returns the size of the buffer used in bsrsm2.

```
integer(4) function cusparseDbsrsm2_bufferSize(handle, dirA, transA, transX,
mb, n, nnzb, descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info,
pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: dirA, transA, transX, mb, n, nnzb, blockDim
  type(cusparseMatDescr) :: descrA
  real(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  type(cusparseBsrsm2Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

5.4.23. cusparseCbsrsm2_bufferSize

This function returns the size of the buffer used in bsrsm2.

```
integer(4) function cusparseCbsrsm2_bufferSize(handle, dirA, transA, transX,
mb, n, nnzb, descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info,
pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: dirA, transA, transX, mb, n, nnzb, blockDim
  type(cusparseMatDescr) :: descrA
  complex(4), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  type(cusparseBsrsm2Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

5.4.24. cusparseZbsrsm2_bufferSize

This function returns the size of the buffer used in bsrsm2.

```
integer(4) function cusparseZbsrsm2_bufferSize(handle, dirA, transA, transX,
mb, n, nnzb, descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info,
pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: dirA, transA, transX, mb, n, nnzb, blockDim
  type(cusparseMatDescr) :: descrA
  complex(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  type(cusparseBsrsm2Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

5.4.25. cusparseSbsrsm2_analysis

This function performs the analysis phase of bsrsm2.

```
integer(4) function cusparseSbsrsm2_analysis(handle, dirA, transA, transX,
mb, n, nnzb, descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, policy,
pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dirA, transA, transX, mb, n, nnzb, blockDim
  type(cusparseMatDescr) :: descrA
  real(4), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  type(cusparseBsrsm2Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

5.4.26. cusparseDbsrsm2_analysis

This function performs the analysis phase of bsrsm2.

```
integer(4) function cusparseDbsrsm2_analysis(handle, dirA, transA, transX,
mb, n, nnzb, descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, policy,
pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dirA, transA, transX, mb, n, nnzb, blockDim
  type(cusparseMatDescr) :: descrA
  real(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  type(cusparseBsrsm2Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

5.4.27. cusparseCbsrsm2_analysis

This function performs the analysis phase of bsrsm2.

```
integer(4) function cusparseCbsrsm2_analysis(handle, dirA, transA, transX,
mb, n, nnzb, descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, policy,
pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dirA, transA, transX, mb, n, nnzb, blockDim
  type(cusparseMatDescr) :: descrA
  complex(4), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  type(cusparseBsrsm2Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

5.4.28. cusparseZbsrsm2_analysis

This function performs the analysis phase of bsrsm2.

```
integer(4) function cusparseZbsrsm2_analysis(handle, dirA, transA, transX,
mb, n, nnzb, descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, policy,
pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dirA, transA, transX, mb, n, nnzb, blockDim
  type(cusparseMatDescr) :: descrA
  complex(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  type(cusparseBsrsm2Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

5.4.29. cusparseSbsrsm2_solve

This function performs the solve phase of bsrsm2.

```
integer(4) function cusparseSbsrsm2_solve(handle, dirA, transA, transX, mb, n,
nnzb, alpha, descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, x, ldx,
y, ldy, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer :: dirA, transA, transX, mb, n, nnzb
  real(4), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descrA
  real(4), device :: bsrValA(*), x(*), y(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer :: blockDim, policy, ldx, ldy
  type(cusparseBsrsm2Info) :: info
  character, device :: pBuffer(*)
```

5.4.30. cusparseDbsrsm2_solve

This function performs the solve phase of bsrsm2.

```
integer(4) function cusparseDbsrsm2_solve(handle, dirA, transA, transX, mb, n,
nnzb, alpha, descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, x, ldx,
y, ldy, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer :: dirA, transA, transX, mb, n, nnzb
  real(8), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descrA
  real(8), device :: bsrValA(*), x(*), y(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer :: blockDim, policy, ldx, ldy
```

```
  type(cusparseBsrsm2Info) :: info
  character, device :: pBuffer(*)
```

5.4.31. cusparseCbsrsm2_solve

This function performs the solve phase of bsrsm2.

```
integer(4) function cusparseCbsrsm2_solve(handle, dirA, transA, transX, mb, n,
nnzb, alpha, descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, x, ldx,
y, ldy, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer :: dirA, transA, transX, mb, n, nnzb
  complex(4), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descrA
  complex(4), device :: bsrValA(*), x(*), y(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer :: blockDim, policy, ldx, ldy
  type(cusparseBsrsm2Info) :: info
  character, device :: pBuffer(*)
```

5.4.32. cusparseZbsrsm2_solve

This function performs the solve phase of bsrsm2.

```
integer(4) function cusparseZbsrsm2_solve(handle, dirA, transA, transX, mb, n,
nnzb, alpha, descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, x, ldx,
y, ldy, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer :: dirA, transA, transX, mb, n, nnzb
  complex(8), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descrA
  complex(8), device :: bsrValA(*), x(*), y(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer :: blockDim, policy, ldx, ldy
  type(cusparseBsrsm2Info) :: info
  character, device :: pBuffer(*)
```

5.4.33. cusparseXbsrsm2_zeroPivot

This function returns an error code equal to CUSPARSE_STATUS_ZERO_PIVOT and sets position to j when A(j,j) is either structural zero or numerical zero. Otherwise, position is set to -1.

```
integer(4) function cusparseXbsrsm2_zeroPivot(handle, info, position)
  type(cusparseHandle) :: handle
  type(cusparseBsrsm2Info) :: info
  integer(4), device :: position ! device or host variable
```

5.5. CUSPARSE Extra Functions

This section contains interfaces for the extra functions that are used to manipulate sparse matrices.

5.5.1. cusparseXcsrgeamNnz

cusparseXcsrgeamNnz computes the number of nonzero elements which will be produced by CSRGEAM.

```
integer(4) function cusparseXcsrgeamNnz(handle, m, n, descrA, nnzA, csrRowPtrA,
csrColIndA, descrB, nnzB, csrRowPtrB, csrColIndB, descrC, csrRowPtrC,
nnzTotalDevHostPtr)
  type(cusparseHandle) :: handle
  integer :: m, n, nnzA, nnzB
  type(cusparseMatDescr) :: descrA, descrB, descrC
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), csrRowPtrB(*),
csrColIndB(*), csrRowPtrC(*)
  integer(4), device :: nnzTotalDevHostPtr ! device or host variable
```

5.5.2. cusparseScsrgeam

CSRGEAM performs the matrix-matrix operation C := alpha * A + beta * B, alpha and beta are scalars. A, B, and C are m x n sparse matrices that are defined in CSR storage format by the three arrays csrVal{A|B|C}, csrRowPtr{A|B|C}, and csrColInd{A|B|C}. cusparseXcsrgeamNnz should be used to determine csrRowPtrC and the number of nonzero elements in the result.

```
integer(4) function cusparseScsrgeam(handle, m, n,        alpha, descrA,
nnzA, csrValA, csrRowPtrA, csrColIndA,        beta, descrB, nnzB, csrValB,
csrRowPtrB, csrColIndB, descrC, csrValC, csrRowPtrC, csrColIndC)
  type(cusparseHandle) :: handle
  integer :: m, n, nnzA, nnzB
  real(4), device :: alpha, beta ! device or host variable
  type(cusparseMatDescr):: descrA, descrB, descrC
  real(4), device :: csrValA(*), csrValB(*), csrValC(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), csrRowPtrB(*),
csrColIndB(*), csrRowPtrC(*), csrColIndC(*)
```

5.5.3. cusparseDcsrgeam

CSRGEAM performs the matrix-matrix operation C := alpha * A + beta * B, alpha and beta are scalars. A, B, and C are m x n sparse matrices that are defined in CSR storage format by the three arrays csrVal{A|B|C}, csrRowPtr{A|B|C}, and csrColInd{A|B|C}. cusparseXcsrgeamNnz should be used to determine csrRowPtrC and the number of nonzero elements in the result.

```
integer(4) function cusparseDcsrgeam(handle, m, n,        alpha, descrA,
nnzA, csrValA, csrRowPtrA, csrColIndA,        beta, descrB, nnzB, csrValB,
csrRowPtrB, csrColIndB, descrC, csrValC, csrRowPtrC, csrColIndC)
  type(cusparseHandle) :: handle
  integer :: m, n, nnzA, nnzB
  real(8), device :: alpha, beta ! device or host variable
  type(cusparseMatDescr):: descrA, descrB, descrC
  real(8), device :: csrValA(*), csrValB(*), csrValC(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), csrRowPtrB(*),
csrColIndB(*), csrRowPtrC(*), csrColIndC(*)
```

5.5.4. cusparseCcsrgeam

CSRGEAM performs the matrix-matrix operation C := alpha * A + beta * B, alpha and beta are scalars. A, B, and C are m x n sparse matrices that are defined in CSR storage format by the three arrays csrVal{A|B|C}, csrRowPtr{A|B|C}, and csrColInd{A|B|C}.

cusparseXcsrgeamNnz should be used to determine csrRowPtrC and the number of nonzero elements in the result.

```
integer(4) function cusparseCcsrgeam(handle, m, n,        alpha, descrA,
nnzA, csrValA, csrRowPtrA, csrColIndA,        beta, descrB, nnzB, csrValB,
csrRowPtrB, csrColIndB, descrC, csrValC, csrRowPtrC, csrColIndC)
  type(cusparseHandle) :: handle
  integer :: m, n, nnzA, nnzB
  complex(4), device :: alpha, beta ! device or host variable
  type(cusparseMatDescr):: descrA, descrB, descrC
  complex(4), device :: csrValA(*), csrValB(*), csrValC(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), csrRowPtrB(*),
csrColIndB(*), csrRowPtrC(*), csrColIndC(*)
```

5.5.5. cusparseZcsrgeam

CSRGEAM performs the matrix-matrix operation C := alpha * A + beta * B, alpha and beta are scalars. A, B, and C are m x n sparse matrices that are defined in CSR storage format by the three arrays csrVal{A|B|C}, csrRowPtr{A|B|C}, and csrColInd{A|B|C}. cusparseXcsrgeamNnz should be used to determine csrRowPtrC and the number of nonzero elements in the result.

```
integer(4) function cusparseZcsrgeam(handle, m, n,        alpha, descrA,
nnzA, csrValA, csrRowPtrA, csrColIndA,        beta, descrB, nnzB, csrValB,
csrRowPtrB, csrColIndB, descrC, csrValC, csrRowPtrC, csrColIndC)
  type(cusparseHandle) :: handle
  integer :: m, n, nnzA, nnzB
  complex(8), device :: alpha, beta ! device or host variable
  type(cusparseMatDescr):: descrA, descrB, descrC
  complex(8), device :: csrValA(*), csrValB(*), csrValC(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), csrRowPtrB(*),
csrColIndB(*), csrRowPtrC(*), csrColIndC(*)
```

5.5.6. cusparseXcsrgemmNnz

cusparseXcsrgemmNnz computes the number of nonzero elements which will be produced by CSRGEMM.

```
integer(4) function cusparseXcsrgemmNnz(handle, transA, transB, m, n, k,
     descrA, nnzA, csrRowPtrA, csrColIndA,        descrB, nnzB, csrRowPtrB,
csrColIndB, descrC, csrRowPtrC, nnzTotalDevHostPtr)
  type(cusparseHandle) :: handle
  integer :: transA, transB, m, n, k, nnzA, nnzB
  type(cusparseMatDescr) :: descrA, descrB, descrC
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), csrRowPtrB(*),
csrColIndB(*), csrRowPtrC(*)
  integer(4), device :: nnzTotalDevHostPtr ! device or host variable
```

5.5.7. cusparseScsrgemm

CSRGEMM performs the matrix-matrix operation C := op(A) * op(B), where op(X) is one of op(X) = X or op(X) = X**T, A, B, and C are m x k, k x n, and m x n sparse matrices that are defined in CSR storage format by the three arrays csrVal{A|B|C}, csrRowPtr{A|B|C}, and csrColInd{A|B|C}. cusparseXcsrgemmNnz should be used to determine csrRowPtrC and the number of nonzero elements in the result.

```
integer(4) function cusparseScsrgemm(handle, transA, transB, m, n, k,
descrA, nnzA, csrValA, csrRowPtrA, csrColIndA,           descrB, nnzB, csrValB,
csrRowPtrB, csrColIndB,           descrC, csrValC, csrRowPtrC, csrColIndC)
  type(cusparseHandle) :: handle
  integer(4) :: transA, transB, m, n, k, nnzA, nnzB
```

```
  type(cusparseMatDescr) :: descrA, descrB, descrC
  real(4), device :: csrValA(*), csrValB(*), csrValC(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), csrRowPtrB(*),
csrColIndB(*), csrRowPtrC(*), csrColIndC(*)
```

5.5.8. cusparseDcsrgemm

CSRGEMM performs the matrix-matrix operation C := op(A) * op(B), where op(X) is one of op(X) = X or op(X) = X**T, A, B, and C are m x k, k x n, and m x n sparse matrices that are defined in CSR storage format by the three arrays csrVal{A|B|C}, csrRowPtr{A|B|C}, and csrColInd{A|B|C}. cusparseXcsrgemmNnz should be used to determine csrRowPtrC and the number of nonzero elements in the result.

```
integer(4) function cusparseDcsrgemm(handle, transA, transB, m, n, k,
descrA, nnzA, csrValA, csrRowPtrA, csrColIndA,           descrB, nnzB, csrValB,
csrRowPtrB, csrColIndB,           descrC, csrValC, csrRowPtrC, csrColIndC)
  type(cusparseHandle) :: handle
  integer(4) :: transA, transB, m, n, k, nnzA, nnzB
  type(cusparseMatDescr) :: descrA, descrB, descrC
  real(8), device :: csrValA(*), csrValB(*), csrValC(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), csrRowPtrB(*),
csrColIndB(*), csrRowPtrC(*), csrColIndC(*)
```

5.5.9. cusparseCcsrgemm

CSRGEMM performs the matrix-matrix operation C := op(A) * op(B), where op(X) is one of op(X) = X or op(X) = X**T, A, B, and C are m x k, k x n, and m x n sparse matrices that are defined in CSR storage format by the three arrays csrVal{A|B|C}, csrRowPtr{A|B|C}, and csrColInd{A|B|C}. cusparseXcsrgemmNnz should be used to determine csrRowPtrC and the number of nonzero elements in the result.

```
integer(4) function cusparseCcsrgemm(handle, transA, transB, m, n, k,
descrA, nnzA, csrValA, csrRowPtrA, csrColIndA,           descrB, nnzB, csrValB,
csrRowPtrB, csrColIndB,           descrC, csrValC, csrRowPtrC, csrColIndC)
  type(cusparseHandle) :: handle
  integer(4) :: transA, transB, m, n, k, nnzA, nnzB
  type(cusparseMatDescr) :: descrA, descrB, descrC
  complex(4), device :: csrValA(*), csrValB(*), csrValC(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), csrRowPtrB(*),
csrColIndB(*), csrRowPtrC(*), csrColIndC(*)
```

5.5.10. cusparseZcsrgemm

CSRGEMM performs the matrix-matrix operation C := op(A) * op(B), where op(X) is one of op(X) = X or op(X) = X**T, A, B, and C are m x k, k x n, and m x n sparse matrices that are defined in CSR storage format by the three arrays csrVal{A|B|C}, csrRowPtr{A|B|C}, and csrColInd{A|B|C}. cusparseXcsrgemmNnz should be used to determine csrRowPtrC and the number of nonzero elements in the result.

```
integer(4) function cusparseZcsrgemm(handle, transA, transB, m, n, k,
descrA, nnzA, csrValA, csrRowPtrA, csrColIndA,           descrB, nnzB, csrValB,
csrRowPtrB, csrColIndB,           descrC, csrValC, csrRowPtrC, csrColIndC)
  type(cusparseHandle) :: handle
  integer(4) :: transA, transB, m, n, k, nnzA, nnzB
  type(cusparseMatDescr) :: descrA, descrB, descrC
  complex(8), device :: csrValA(*), csrValB(*), csrValC(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), csrRowPtrB(*),
csrColIndB(*), csrRowPtrC(*), csrColIndC(*)
```

5.5.11. cusparseScsrgemm2_bufferSizeExt

This function returns the size of the buffer used in csrgemm2.

```
integer(4) function cusparseScsrgemm2_bufferSizeExt(handle, m, n, k,
     alpha,        descrA, nnzA, csrRowPtrA, csrColIndA,        descrB,
nnzB, csrRowPtrB, csrColIndB,        beta,        descrD, nnzD, csrRowPtrD,
csrColIndD, info, pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  real(4), device :: alpha, beta ! device or host variable
  integer :: m, n, k, nnzA, nnzB, nnzD
  type(cusparseMatDescr) :: descrA, descrB, descrD
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), csrRowPtrB(*),
csrColIndB(*), csrRowPtrD(*), csrColIndD(*)
  type(cusparseCsrgemm2Info) :: info
  integer(8) :: pBufferSizeInBytes
```

5.5.12. cusparseDcsrgemm2_bufferSizeExt

This function returns the size of the buffer used in csrgemm2.

```
integer(4) function cusparseDcsrgemm2_bufferSizeExt(handle, m, n, k,
     alpha,        descrA, nnzA, csrRowPtrA, csrColIndA,        descrB,
nnzB, csrRowPtrB, csrColIndB,        beta,        descrD, nnzD, csrRowPtrD,
csrColIndD, info, pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  real(8), device :: alpha, beta ! device or host variable
  integer :: m, n, k, nnzA, nnzB, nnzD
  type(cusparseMatDescr) :: descrA, descrB, descrD
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), csrRowPtrB(*),
csrColIndB(*), csrRowPtrD(*), csrColIndD(*)
  type(cusparseCsrgemm2Info) :: info
  integer(8) :: pBufferSizeInBytes
```

5.5.13. cusparseCcsrgemm2_bufferSizeExt

This function returns the size of the buffer used in csrgemm2.

```
integer(4) function cusparseCcsrgemm2_bufferSizeExt(handle, m, n, k,
     alpha,        descrA, nnzA, csrRowPtrA, csrColIndA,        descrB,
nnzB, csrRowPtrB, csrColIndB,        beta,        descrD, nnzD, csrRowPtrD,
csrColIndD, info, pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  complex(4), device :: alpha, beta ! device or host variable
  integer :: m, n, k, nnzA, nnzB, nnzD
  type(cusparseMatDescr) :: descrA, descrB, descrD
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), csrRowPtrB(*),
csrColIndB(*), csrRowPtrD(*), csrColIndD(*)
  type(cusparseCsrgemm2Info) :: info
  integer(8) :: pBufferSizeInBytes
```

5.5.14. cusparseZcsrgemm2_bufferSizeExt

This function returns the size of the buffer used in csrgemm2.

```
integer(4) function cusparseZcsrgemm2_bufferSizeExt(handle, m, n, k,
     alpha,        descrA, nnzA, csrRowPtrA, csrColIndA,        descrB,
nnzB, csrRowPtrB, csrColIndB,        beta,        descrD, nnzD, csrRowPtrD,
csrColIndD, info, pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  complex(8), device :: alpha, beta ! device or host variable
  integer :: m, n, k, nnzA, nnzB, nnzD
  type(cusparseMatDescr) :: descrA, descrB, descrD
```

```
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), csrRowPtrB(*),
csrColIndB(*), csrRowPtrD(*), csrColIndD(*)
  type(cusparseCsrgemm2Info) :: info
  integer(8) :: pBufferSizeInBytes
```

5.5.15. cusparseXcsrgemm2Nnz

cusparseXcsrgemm2Nnz computes the number of nonzero elements which will be produced by CSRGEMM2.

```
integer(4) function cusparseXcsrgemm2Nnz(handle,           m, n, k,
  descrA, nnzA, csrRowPtrA, csrColIndA,          descrB, nnzB, csrRowPtrB,
csrColIndB,          descrD, nnzD, csrRowPtrD, csrColIndD,          descrC,
csrRowPtrC, nnzTotalDevHostPtr, info, pBuffer)
  type(cusparseHandle) :: handle
  type(cusparseMatDescr) :: descrA, descrB, descrD, descrC
  type(cusparseCsrgemm2Info) :: info
  integer(4) :: m, n, k, nnzA, nnzB, nnzD
  integer(4), device :: csrRowPtrA(*), csrColIndA(*),
csrRowPtrB(*), csrColIndB(*), csrRowPtrD(*), csrColIndD(*), csrRowPtrC(*)
  integer(c_int) :: nnzTotalDevHostPtr
  character(c_char), device :: pBuffer(*)
```

5.5.16. cusparseScsrgemm2

CSRGEMM2 performs the matrix-matrix operation C := alpha * A * B + beta * D alpha and beta are scalars. A, B, and C are m x k, k x n, and m x n sparse matrices that are defined in CSR storage format by the three arrays csrVal{A|B|C|D}, csrRowPtr{A|B|C|D}, and csrColInd{A|B|C|D}. cusparseXcsrgemm2Nnz should be used to determine csrRowPtrC and the number of nonzero elements in the result.

```
integer(4) function cusparseScsrgemm2(handle,        m, n, k, alpha,
descrA, nnzA, csrValA, csrRowPtrA, csrColIndA,        descrB, nnzB, csrValB,
csrRowPtrB, csrColIndB,        beta,        descrD, nnzD, csrValD, csrRowPtrD,
csrColIndD,        descrC, csrValC, csrRowPtrC, csrColIndC, info, pBuffer)
  type(cusparseHandle) :: handle
  type(cusparseMatDescr) :: descrA, descrB, descrD, descrC
  type(cusparseCsrgemm2Info) :: info
  integer :: m, n, k, nnzA, nnzB, nnzD
  integer(4), device :: csrRowPtrA(*), csrColIndA(*),          csrRowPtrB(*),
csrColIndB(*),          csrRowPtrD(*), csrColIndD(*), csrRowPtrC(*),
csrColIndC(*)
  real(4), device :: csrValA(*), csrValB(*), csrValD(*), csrValC(*)
  real(4), device :: alpha, beta ! device or host variable
  integer(4), device :: nnzTotalDevHostPtr ! device or host variable
  character, device :: pBuffer(*)
```

5.5.17. cusparseDcsrgemm2

CSRGEMM2 performs the matrix-matrix operation C := alpha * A * B + beta * D alpha and beta are scalars. A, B, and C are m x k, k x n, and m x n sparse matrices that are defined in CSR storage format by the three arrays csrVal{A|B|C|D}, csrRowPtr{A|B|C|D}, and csrColInd{A|B|C|D}. cusparseXcsrgemm2Nnz should be used to determine csrRowPtrC and the number of nonzero elements in the result.

```
integer(4) function cusparseDcsrgemm2(handle,        m, n, k, alpha,
descrA, nnzA, csrValA, csrRowPtrA, csrColIndA,        descrB, nnzB, csrValB,
csrRowPtrB, csrColIndB,        beta,        descrD, nnzD, csrValD, csrRowPtrD,
csrColIndD,        descrC, csrValC, csrRowPtrC, csrColIndC, info, pBuffer)
  type(cusparseHandle) :: handle
  type(cusparseMatDescr) :: descrA, descrB, descrD, descrC
```

```
  type(cusparseCsrgemm2Info) :: info
  integer :: m, n, k, nnzA, nnzB, nnzD
  integer(4), device :: csrRowPtrA(*), csrColIndA(*),          csrRowPtrB(*),
csrColIndB(*),          csrRowPtrD(*), csrColIndD(*), csrRowPtrC(*),
csrColIndC(*)
  real(8), device :: csrValA(*), csrValB(*), csrValD(*), csrValC(*)
  real(8), device :: alpha, beta ! device or host variable
  integer(4), device :: nnzTotalDevHostPtr ! device or host variable
  character, device :: pBuffer(*)
```

5.5.18. cusparseCcsrgemm2

CSRGEMM2 performs the matrix-matrix operation C := alpha * A * B + beta * D alpha and beta are scalars. A, B, and C are m x k, k x n, and m x n sparse matrices that are defined in CSR storage format by the three arrays csrVal{A|B|C|D}, csrRowPtr{A|B|C|D}, and csrColInd{A|B|C|D}. cusparseXcsrgemm2Nnz should be used to determine csrRowPtrC and the number of nonzero elements in the result.

```
integer(4) function cusparseCcsrgemm2(handle,        m, n, k, alpha,
descrA, nnzA, csrValA, csrRowPtrA, csrColIndA,        descrB, nnzB, csrValB,
csrRowPtrB, csrColIndB,        beta,        descrD, nnzD, csrValD, csrRowPtrD,
csrColIndD,        descrC, csrValC, csrRowPtrC, csrColIndC, info, pBuffer)
  type(cusparseHandle) :: handle
  type(cusparseMatDescr) :: descrA, descrB, descrD, descrC
  type(cusparseCsrgemm2Info) :: info
  integer :: m, n, k, nnzA, nnzB, nnzD
  integer(4), device :: csrRowPtrA(*), csrColIndA(*),          csrRowPtrB(*),
csrColIndB(*),          csrRowPtrD(*), csrColIndD(*), csrRowPtrC(*),
csrColIndC(*)
  complex(4), device :: csrValA(*), csrValB(*), csrValD(*), csrValC(*)
  complex(4), device :: alpha, beta ! device or host variable
  integer(4), device :: nnzTotalDevHostPtr ! device or host variable
  character, device :: pBuffer(*)
```

5.5.19. cusparseZcsrgemm2

CSRGEMM2 performs the matrix-matrix operation C := alpha * A * B + beta * D alpha and beta are scalars. A, B, and C are m x k, k x n, and m x n sparse matrices that are defined in CSR storage format by the three arrays csrVal{A|B|C|D}, csrRowPtr{A|B|C|D}, and csrColInd{A|B|C|D}. cusparseXcsrgemm2Nnz should be used to determine csrRowPtrC and the number of nonzero elements in the result.

```
integer(4) function cusparseZcsrgemm2(handle,        m, n, k, alpha,
descrA, nnzA, csrValA, csrRowPtrA, csrColIndA,        descrB, nnzB, csrValB,
csrRowPtrB, csrColIndB,        beta,        descrD, nnzD, csrValD, csrRowPtrD,
csrColIndD,        descrC, csrValC, csrRowPtrC, csrColIndC, info, pBuffer)
  type(cusparseHandle) :: handle
  type(cusparseMatDescr) :: descrA, descrB, descrD, descrC
  type(cusparseCsrgemm2Info) :: info
  integer :: m, n, k, nnzA, nnzB, nnzD
  integer(4), device :: csrRowPtrA(*), csrColIndA(*),          csrRowPtrB(*),
csrColIndB(*),          csrRowPtrD(*), csrColIndD(*), csrRowPtrC(*),
csrColIndC(*)
  complex(8), device :: csrValA(*), csrValB(*), csrValD(*), csrValC(*)
  complex(8), device :: alpha, beta ! device or host variable
  integer(4), device :: nnzTotalDevHostPtr ! device or host variable
  character, device :: pBuffer(*)
```

5.6. CUSPARSE Preconditioning Functions

This section contains interfaces for the preconditioning functions that are used in processing sparse matrices.

5.6.1. cusparseScsric0

CSRIC0 computes the incomplete-Cholesky factorization with zero fill-in and no pivoting. A is an m x n Hermitian/symmetric positive definite sparse matrix that is defined in CSR storage format by the three arrays csrValA, csrRowPtrA, and csrColIndA.

```
integer(4) function cusparseScsric0(handle, trans, m,           descrA,
csrValM, csrRowPtrA, csrColIndA, info)
  type(cusparseHandle) :: handle
  integer(4) :: trans, m
  type(cusparseMatDescr) :: descrA
  real(4), device :: csrValM(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseSolveAnalysisInfo) :: info
```

5.6.2. cusparseDcsric0

CSRIC0 computes the incomplete-Cholesky factorization with zero fill-in and no pivoting. A is an m x n Hermitian/symmetric positive definite sparse matrix that is defined in CSR storage format by the three arrays csrValA, csrRowPtrA, and csrColIndA.

```
integer(4) function cusparseDcsric0(handle, trans, m,           descrA,
csrValM, csrRowPtrA, csrColIndA, info)
  type(cusparseHandle) :: handle
  integer(4) :: trans, m
  type(cusparseMatDescr) :: descrA
  real(8), device :: csrValM(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseSolveAnalysisInfo) :: info
```

5.6.3. cusparseCcsric0

CSRIC0 computes the incomplete-Cholesky factorization with zero fill-in and no pivoting. A is an m x n Hermitian/symmetric positive definite sparse matrix that is defined in CSR storage format by the three arrays csrValA, csrRowPtrA, and csrColIndA.

```
integer(4) function cusparseCcsric0(handle, trans, m,           descrA,
csrValM, csrRowPtrA, csrColIndA, info)
  type(cusparseHandle) :: handle
  integer(4) :: trans, m
  type(cusparseMatDescr) :: descrA
  complex(4), device :: csrValM(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseSolveAnalysisInfo) :: info
```

5.6.4. cusparseZcsric0

CSRIC0 computes the incomplete-Cholesky factorization with zero fill-in and no pivoting. A is an m x n Hermitian/symmetric positive definite sparse matrix that is defined in CSR storage format by the three arrays csrValA, csrRowPtrA, and csrColIndA.

```
integer(4) function cusparseZcsric0(handle, trans, m,           descrA,
csrValM, csrRowPtrA, csrColIndA, info)
  type(cusparseHandle) :: handle
  integer(4) :: trans, m
  type(cusparseMatDescr) :: descrA
  complex(8), device :: csrValM(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseSolveAnalysisInfo) :: info
```

5.6.5. cusparseScsrilu0

CSRILU0 computes the incomplete-LU factorization with zero fill-in and no pivoting. A is an m x n sparse matrix that is defined in CSR storage format by the three arrays csrValA, csrRowPtrA, and csrColIndA.

```
integer(4) function cusparseScsrilu0(handle, trans, m,           descrA,
csrValM, csrRowPtrA, csrColIndA, info)
  type(cusparseHandle) :: handle
  integer(4) :: trans, m
  type(cusparseMatDescr) :: descrA
  real(4), device :: csrValM(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseSolveAnalysisInfo) :: info
```

5.6.6. cusparseDcsrilu0

CSRILU0 computes the incomplete-LU factorization with zero fill-in and no pivoting. A is an m x n sparse matrix that is defined in CSR storage format by the three arrays csrValA, csrRowPtrA, and csrColIndA.

```
integer(4) function cusparseDcsrilu0(handle, trans, m,           descrA,
csrValM, csrRowPtrA, csrColIndA, info)
  type(cusparseHandle) :: handle
  integer(4) :: trans, m
  type(cusparseMatDescr) :: descrA
  real(8), device :: csrValM(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseSolveAnalysisInfo) :: info
```

5.6.7. cusparseCcsrilu0

CSRILU0 computes the incomplete-LU factorization with zero fill-in and no pivoting. A is an m x n sparse matrix that is defined in CSR storage format by the three arrays csrValA, csrRowPtrA, and csrColIndA.

```
integer(4) function cusparseCcsrilu0(handle, trans, m,           descrA,
csrValM, csrRowPtrA, csrColIndA, info)
  type(cusparseHandle) :: handle
  integer(4) :: trans, m
  type(cusparseMatDescr) :: descrA
  complex(4), device :: csrValM(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseSolveAnalysisInfo) :: info
```

5.6.8. cusparseZcsrilu0

CSRILU0 computes the incomplete-LU factorization with zero fill-in and no pivoting. A is an m x n sparse matrix that is defined in CSR storage format by the three arrays csrValA, csrRowPtrA, and csrColIndA.

```
integer(4) function cusparseZcsrilu0(handle, trans, m,           descrA,
csrValM, csrRowPtrA, csrColIndA, info)
  type(cusparseHandle) :: handle
  integer(4) :: trans, m
  type(cusparseMatDescr) :: descrA
  complex(8), device :: csrValM(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseSolveAnalysisInfo) :: info
```

5.6.9. cusparseSgtsv

GTSV computes the solution of a tridiagonal linear system with multiple right hand sides: A * Y = a * x The coefficient matrix A of this tri-diagonal linear system is defined with three vectors corresponding to its lower (dl), main (d), and upper (du) matrix diagonals; the right-hand sides are stored in the dense matrix X. The solution Y overwrites the righthand-side matrix X on exit.

```
integer(4) function cusparseSgtsv(handle, m, n, dl, d, du, B, ldb)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, ldb
  real(4), device :: dl(*), d(*), du(*), B(*)
```

5.6.10. cusparseDgtsv

GTSV computes the solution of a tridiagonal linear system with multiple right hand sides: A * Y = a * x The coefficient matrix A of this tri-diagonal linear system is defined with three vectors corresponding to its lower (dl), main (d), and upper (du) matrix diagonals; the right-hand sides are stored in the dense matrix X. The solution Y overwrites the righthand-side matrix X on exit.

```
integer(4) function cusparseDgtsv(handle, m, n, dl, d, du, B, ldb)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, ldb
  real(8), device :: dl(*), d(*), du(*), B(*)
```

5.6.11. cusparseCgtsv

GTSV computes the solution of a tridiagonal linear system with multiple right hand sides: A * Y = a * x The coefficient matrix A of this tri-diagonal linear system is defined with three vectors corresponding to its lower (dl), main (d), and upper (du) matrix diagonals; the right-hand sides are stored in the dense matrix X. The solution Y overwrites the righthand-side matrix X on exit.

```
integer(4) function cusparseCgtsv(handle, m, n, dl, d, du, B, ldb)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, ldb
  complex(4), device :: dl(*), d(*), du(*), B(*)
```

5.6.12. cusparseZgtsv

GTSV computes the solution of a tridiagonal linear system with multiple right hand sides: A * Y = a * x The coefficient matrix A of this tri-diagonal linear system is defined with three vectors corresponding to its lower (dl), main (d), and upper (du) matrix diagonals; the right-hand sides are stored in the dense matrix X. The solution Y overwrites the righthand-side matrix X on exit.

```
integer(4) function cusparseZgtsv(handle, m, n, dl, d, du, B, ldb)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, ldb
  complex(8), device :: dl(*), d(*), du(*), B(*)
```

5.6.13. cusparseSgtsv_nopivot

GTSV computes the solution of a tridiagonal linear system with multiple right hand sides: A * Y = a * x The coefficient matrix A of this tri-diagonal linear system is defined with three vectors corresponding to its lower (dl), main (d), and upper (du) matrix diagonals; the right-hand sides are stored in the dense matrix X. The solution Y overwrites the righthand-side matrix X on exit. This function does not perform pivoting.

```
integer(4) function cusparseSgtsv_nopivot(handle, m, n, dl, d, du, B, ldb)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, ldb
  real(4), device :: dl(*), d(*), du(*), B(*)
```

5.6.14. cusparseDgtsv_nopivot

GTSV computes the solution of a tridiagonal linear system with multiple right hand sides: A * Y = a * x The coefficient matrix A of this tri-diagonal linear system is defined with three vectors corresponding to its lower (dl), main (d), and upper (du) matrix diagonals; the right-hand sides are stored in the dense matrix X. The solution Y overwrites the righthand-side matrix X on exit. This function does not perform pivoting.

```
integer(4) function cusparseDgtsv_nopivot(handle, m, n, dl, d, du, B, ldb)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, ldb
  real(8), device :: dl(*), d(*), du(*), B(*)
```

5.6.15. cusparseCgtsv_nopivot

GTSV computes the solution of a tridiagonal linear system with multiple right hand sides: A * Y = a * x The coefficient matrix A of this tri-diagonal linear system is defined with three vectors corresponding to its lower (dl), main (d), and upper (du) matrix diagonals; the right-hand sides are stored in the dense matrix X. The solution Y overwrites the righthand-side matrix X on exit. This function does not perform pivoting.

```
integer(4) function cusparseCgtsv_nopivot(handle, m, n, dl, d, du, B, ldb)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, ldb
  complex(4), device :: dl(*), d(*), du(*), B(*)
```

5.6.16. cusparseZgtsv_nopivot

GTSV computes the solution of a tridiagonal linear system with multiple right hand sides: A * Y = a * x The coefficient matrix A of this tri-diagonal linear system is defined with three vectors corresponding to its lower (dl), main (d), and upper (du) matrix diagonals; the right-hand sides are stored in the dense matrix X. The solution Y overwrites the righthand-side matrix X on exit. This function does not perform pivoting.

```
integer(4) function cusparseZgtsv_nopivot(handle, m, n, dl, d, du, B, ldb)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, ldb
  complex(8), device :: dl(*), d(*), du(*), B(*)
```

5.6.17. cusparseSgtsvStridedBatch

GTSVStridedBatch computes the solution of multiple tridiagonal linear systems with multiple right hand sides: A * Y = a * x The coefficient matrix A of each tri-diagonal linear system is defined with three vectors corresponding to its lower (dl), main (d), and upper (du) matrix diagonals; the right-hand sides are stored in the dense matrix X. The solution Y overwrites the righthand-side matrix X on exit.

```
integer(4) function cusparseSgtsvStridedBatch(handle, m, dl, d, du, x,
batchCount, batchStride)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, batchCount, batchStride
  real(4), device :: dl(*), d(*), du(*), x(*)
```

5.6.18. cusparseDgtsvStridedBatch

GTSVStridedBatch computes the solution of multiple tridiagonal linear systems with multiple right hand sides: A * Y = a * x The coefficient matrix A of each tri-diagonal linear system is defined with three vectors corresponding to its lower (dl), main (d), and upper (du) matrix diagonals; the right-hand sides are stored in the dense matrix X. The solution Y overwrites the righthand-side matrix X on exit.

```
integer(4) function cusparseDgtsvStridedBatch(handle, m, dl, d, du, x,
batchCount, batchStride)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, batchCount, batchStride
  real(8), device :: dl(*), d(*), du(*), x(*)
```

5.6.19. cusparseCgtsvStridedBatch

GTSVStridedBatch computes the solution of multiple tridiagonal linear systems with multiple right hand sides: A * Y = a * x The coefficient matrix A of each tri-diagonal linear system is defined with three vectors corresponding to its lower (dl), main (d), and upper (du) matrix diagonals; the right-hand sides are stored in the dense matrix X. The solution Y overwrites the righthand-side matrix X on exit.

```
integer(4) function cusparseCgtsvStridedBatch(handle, m, dl, d, du, x,
batchCount, batchStride)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, batchCount, batchStride
  complex(4), device :: dl(*), d(*), du(*), x(*)
```

5.6.20. cusparseZgtsvStridedBatch

GTSVStridedBatch computes the solution of multiple tridiagonal linear systems with multiple right hand sides: A * Y = a * x The coefficient matrix A of each tri-diagonal linear system is defined with three vectors corresponding to its lower (dl), main (d), and upper (du) matrix diagonals; the right-hand sides are stored in the dense matrix X. The solution Y overwrites the righthand-side matrix X on exit.

```
integer(4) function cusparseZgtsvStridedBatch(handle, m, dl, d, du, x,
batchCount, batchStride)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, batchCount, batchStride
  complex(8), device :: dl(*), d(*), du(*), x(*)
```

5.6.21. cusparseSgtsv2_buffersize

Sgtsv2_buffersize returns the size of the buffer, in bytes, required in Sgtsv2().

```
integer(4) function cusparseSgtsv2_bufferSize(handle, m, n, dl, d, du, B, ldb,
pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, ldb
  real(4), device :: dl(m), d(m), du(m), B(ldb,n)
  integer(8) :: pBufferSizeInBytes
```

5.6.22. cusparseDgtsv2_buffersize

Dgtsv2_buffersize returns the size of the buffer, in bytes, required in Dgtsv2().

```
integer(4) function cusparseDgtsv2_bufferSize(handle, m, n, dl, d, du, B, ldb,
pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, ldb
  real(8), device :: dl(m), d(m), du(m), B(ldb,n)
  integer(8) :: pBufferSizeInBytes
```

5.6.23. cusparseCgtsv2_buffersize

Cgtsv2_buffersize returns the size of the buffer, in bytes, required in Cgtsv2().

```
integer(4) function cusparseCgtsv2_bufferSize(handle, m, n, dl, d, du, B, ldb,
pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, ldb
  complex(4), device :: dl(m), d(m), du(m), B(ldb,n)
  integer(8) :: pBufferSizeInBytes
```

5.6.24. cusparseZgtsv2_buffersize

Zgtsv2_buffersize returns the size of the buffer, in bytes, required in Zgtsv2().

```
integer(4) function cusparseZgtsv2_bufferSize(handle, m, n, dl, d, du, B, ldb,
pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, ldb
  complex(8), device :: dl(m), d(m), du(m), B(ldb,n)
  integer(8) :: pBufferSizeInBytes
```

5.6.25. cusparseSgtsv2

Sgtsv2 computes the solution of a tridiagonal linear system with multiple right hand sides: A * X = B The coefficient matrix A of the tri-diagonal linear system is defined with three vectors corresponding to its lower (dl), main (d), and upper (du) matrix diagonals; the right-hand sides are stored in the dense matrix B. The solution X overwrites the righthand-side matrix B on exit. The input m is the size of the linear system. The input n is the number or right-hand sides in B.

```
integer(4) function cusparseSgtsv2(handle, m, n, dl, d, du, B, ldb, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, ldb
  real(4), device :: dl(m), d(m), du(m), B(ldb,n)
  character(1), device :: pBuffer(*)
```

5.6.26. cusparseDgtsv2

Dgtsv2 computes the solution of a tridiagonal linear system with multiple right hand sides: A * X = B The coefficient matrix A of the tri-diagonal linear system is defined with three vectors corresponding to its lower (dl), main (d), and upper (du) matrix diagonals; the right-hand sides are stored in the dense matrix B. The solution X overwrites the righthand-side matrix B on exit. The input m is the size of the linear system. The input n is the number or right-hand sides in B.

```
integer(4) function cusparseDgtsv2(handle, m, n, dl, d, du, B, ldb, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, ldb
  real(8), device :: dl(m), d(m), du(m), B(ldb,n)
  character(1), device :: pBuffer(*)
```

5.6.27. cusparseCgtsv2

Cgtsv2 computes the solution of a tridiagonal linear system with multiple right hand sides: A * X = B The coefficient matrix A of the tri-diagonal linear system is defined with three vectors corresponding to its lower (dl), main (d), and upper (du) matrix diagonals; the right-hand sides are stored in the dense matrix B. The solution X overwrites the righthand-side matrix B on exit. The input m is the size of the linear system. The input n is the number or right-hand sides in B.

```
integer(4) function cusparseCgtsv2(handle, m, n, dl, d, du, B, ldb, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, ldb
  complex(4), device :: dl(m), d(m), du(m), B(ldb,n)
  character(1), device :: pBuffer(*)
```

5.6.28. cusparseZgtsv2

Zgtsv2 computes the solution of a tridiagonal linear system with multiple right hand sides: A * X = B The coefficient matrix A of the tri-diagonal linear system is defined with three vectors corresponding to its lower (dl), main (d), and upper (du) matrix diagonals; the right-hand sides are stored in the dense matrix B. The solution X overwrites the righthand-side matrix B on exit. The input m is the size of the linear system. The input n is the number or right-hand sides in B.

```
integer(4) function cusparseZgtsv2(handle, m, n, dl, d, du, B, ldb, pBuffer)
```

```
  type(cusparseHandle) :: handle
  integer(4) :: m, n, ldb
  complex(8), device :: dl(m), d(m), du(m), B(ldb,n)
  character(1), device :: pBuffer(*)
```

5.6.29. cusparseSgtsv2_nopivot_buffersize

Sgtsv2_nopivot_buffersize returns the size of the buffer, in bytes, required in Sgtsv2_nopivot().

```
integer(4) function cusparseSgtsv2_nopivot_bufferSize(handle, m, n, dl, d, du,
B, ldb, pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, ldb
  real(4), device :: dl(m), d(m), du(m), B(ldb,n)
  integer(8) :: pBufferSizeInBytes
```

5.6.30. cusparseDgtsv2_nopivot_buffersize

Dgtsv2_nopivot_buffersize returns the size of the buffer, in bytes, required in Dgtsv2_nopivot().

```
integer(4) function cusparseDgtsv2_nopivot_bufferSize(handle, m, n, dl, d, du,
B, ldb, pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, ldb
  real(8), device :: dl(m), d(m), du(m), B(ldb,n)
  integer(8) :: pBufferSizeInBytes
```

5.6.31. cusparseCgtsv2_nopivot_buffersize

Cgtsv2_nopivot_buffersize returns the size of the buffer, in bytes, required in Cgtsv2_nopivot().

```
integer(4) function cusparseCgtsv2_nopivot_bufferSize(handle, m, n, dl, d, du,
B, ldb, pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, ldb
  complex(4), device :: dl(m), d(m), du(m), B(ldb,n)
  integer(8) :: pBufferSizeInBytes
```

5.6.32. cusparseZgtsv2_nopivot_buffersize

Zgtsv2_nopivot_buffersize returns the size of the buffer, in bytes, required in Zgtsv2_nopivot().

```
integer(4) function cusparseZgtsv2_nopivot_bufferSize(handle, m, n, dl, d, du,
B, ldb, pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, ldb
  complex(8), device :: dl(m), d(m), du(m), B(ldb,n)
  integer(8) :: pBufferSizeInBytes
```

5.6.33. cusparseSgtsv2_nopivot

Sgtsv2_nopivot computes the solution of a tridiagonal linear system with multiple right hand sides, without pivoting: A * X = B The coefficient matrix A of the tri-diagonal linear system is defined with three vectors corresponding to its lower (dl), main (d), and upper (du) matrix diagonals; the right-hand sides are stored in the dense matrix B. The solution

X overwrites the righthand-side matrix B on exit. The input m is the size of the linear system. The input n is the number or right-hand sides in B.

```
integer(4) function cusparseSgtsv2_nopivot(handle, m, n, dl, d, du, B, ldb,
pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, ldb
  real(4), device :: dl(m), d(m), du(m), B(ldb,n)
  character(1), device :: pBuffer(*)
```

5.6.34. cusparseDgtsv2_nopivot

Dgtsv2_nopivot computes the solution of a tridiagonal linear system with multiple right hand sides, without pivoting: A * X = B The coefficient matrix A of the tri-diagonal linear system is defined with three vectors corresponding to its lower (dl), main (d), and upper (du) matrix diagonals; the right-hand sides are stored in the dense matrix B. The solution X overwrites the righthand-side matrix B on exit. The input m is the size of the linear system. The input n is the number or right-hand sides in B.

```
integer(4) function cusparseDgtsv2_nopivot(handle, m, n, dl, d, du, B, ldb,
pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, ldb
  real(8), device :: dl(m), d(m), du(m), B(ldb,n)
  character(1), device :: pBuffer(*)
```

5.6.35. cusparseCgtsv2_nopivot

Cgtsv2_nopivot computes the solution of a tridiagonal linear system with multiple right hand sides, without pivoting: A * X = B The coefficient matrix A of the tri-diagonal linear system is defined with three vectors corresponding to its lower (dl), main (d), and upper (du) matrix diagonals; the right-hand sides are stored in the dense matrix B. The solution X overwrites the righthand-side matrix B on exit. The input m is the size of the linear system. The input n is the number or right-hand sides in B.

```
integer(4) function cusparseCgtsv2_nopivot(handle, m, n, dl, d, du, B, ldb,
pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, ldb
  complex(4), device :: dl(m), d(m), du(m), B(ldb,n)
  character(1), device :: pBuffer(*)
```

5.6.36. cusparseZgtsv2_nopivot

Zgtsv2_nopivot computes the solution of a tridiagonal linear system with multiple right hand sides, without pivoting: A * X = B The coefficient matrix A of the tri-diagonal linear system is defined with three vectors corresponding to its lower (dl), main (d), and upper (du) matrix diagonals; the right-hand sides are stored in the dense matrix B. The solution X overwrites the righthand-side matrix B on exit. The input m is the size of the linear system. The input n is the number or right-hand sides in B.

```
integer(4) function cusparseZgtsv2_nopivot(handle, m, n, dl, d, du, B, ldb,
pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, ldb
  complex(8), device :: dl(m), d(m), du(m), B(ldb,n)
  character(1), device :: pBuffer(*)
```

5.6.37. cusparseSgtsv2StridedBatch_buffersize

Sgtsv2StridedBatch_buffersize returns the size of the buffer, in bytes, required in Sgtsv2StridedBatch().

```
integer(4) function cusparseSgtsv2StridedBatch_bufferSize(handle, m, dl, d, du,
x, batchCount, batchStride, pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer(4) :: m, batchCount, batchStride
  real(4), device :: dl(*), d(*), du(*), x(*)
  integer(8) :: pBufferSizeInBytes
```

5.6.38. cusparseDgtsv2StridedBatch_buffersize

Dgtsv2StridedBatch_buffersize returns the size of the buffer, in bytes, required in Dgtsv2StridedBatch().

```
integer(4) function cusparseDgtsv2StridedBatch_bufferSize(handle, m, dl, d, du,
x, batchCount, batchStride, pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer(4) :: m, batchCount, batchStride
  real(8), device :: dl(*), d(*), du(*), x(*)
  integer(8) :: pBufferSizeInBytes
```

5.6.39. cusparseCgtsv2StridedBatch_buffersize

Cgtsv2StridedBatch_buffersize returns the size of the buffer, in bytes, required in Cgtsv2StridedBatch().

```
integer(4) function cusparseCgtsv2StridedBatch_bufferSize(handle, m, dl, d, du,
x, batchCount, batchStride, pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer(4) :: m, batchCount, batchStride
  complex(4), device :: dl(*), d(*), du(*), x(*)
  integer(8) :: pBufferSizeInBytes
```

5.6.40. cusparseZgtsv2StridedBatch_buffersize

Zgtsv2StridedBatch_buffersize returns the size of the buffer, in bytes, required in Zgtsv2StridedBatch().

```
integer(4) function cusparseZgtsv2StridedBatch_bufferSize(handle, m, dl, d, du,
x, batchCount, batchStride, pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer(4) :: m, batchCount, batchStride
  complex(8), device :: dl(*), d(*), du(*), x(*)
  integer(8) :: pBufferSizeInBytes
```

5.6.41. cusparseSgtsv2StridedBatch

Sgtsv2StridedBatch computes the solution of multiple tridiagonal linear systems with multiple right hand sides: A * X = B The coefficient matrix A of each tri-diagonal linear system is defined with three vectors corresponding to its lower (dl), main (d), and upper (du) matrix diagonals; the right-hand sides are stored in the dense matrix B. The solution X overwrites the righthand-side matrix B on exit.

```
integer(4) function cusparseSgtsv2StridedBatch(handle, m, dl, d, du, x,
batchCount, batchStride, pBuffer)
  type(cusparseHandle) :: handle
```

```
  integer(4) :: m, batchCount, batchStride
  real(4), device :: dl(*), d(*), du(*), x(*)
  character(1), device :: pBuffer(*)
```

5.6.42. cusparseDgtsv2StridedBatch

Dgtsv2StridedBatch computes the solution of multiple tridiagonal linear systems with multiple right hand sides: A * X = B The coefficient matrix A of each tri-diagonal linear system is defined with three vectors corresponding to its lower (dl), main (d), and upper (du) matrix diagonals; the right-hand sides are stored in the dense matrix B. The solution X overwrites the righthand-side matrix B on exit.

```
integer(4) function cusparseDgtsv2StridedBatch(handle, m, dl, d, du, x,
batchCount, batchStride, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, batchCount, batchStride
  real(8), device :: dl(*), d(*), du(*), x(*)
  character(1), device :: pBuffer(*)
```

5.6.43. cusparseCgtsv2StridedBatch

Cgtsv2StridedBatch computes the solution of multiple tridiagonal linear systems with multiple right hand sides: A * X = B The coefficient matrix A of each tri-diagonal linear system is defined with three vectors corresponding to its lower (dl), main (d), and upper (du) matrix diagonals; the right-hand sides are stored in the dense matrix B. The solution X overwrites the righthand-side matrix B on exit.

```
integer(4) function cusparseCgtsv2StridedBatch(handle, m, dl, d, du, x,
batchCount, batchStride, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, batchCount, batchStride
  complex(4), device :: dl(*), d(*), du(*), x(*)
  character(1), device :: pBuffer(*)
```

5.6.44. cusparseZgtsv2StridedBatch

Zgtsv2StridedBatch computes the solution of multiple tridiagonal linear systems with multiple right hand sides: A * X = B The coefficient matrix A of each tri-diagonal linear system is defined with three vectors corresponding to its lower (dl), main (d), and upper (du) matrix diagonals; the right-hand sides are stored in the dense matrix B. The solution X overwrites the righthand-side matrix B on exit.

```
integer(4) function cusparseZgtsv2StridedBatch(handle, m, dl, d, du, x,
batchCount, batchStride, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, batchCount, batchStride
  complex(8), device :: dl(*), d(*), du(*), x(*)
  character(1), device :: pBuffer(*)
```

5.6.45. cusparseSgtsvInterleavedBatch_buffersize

SgtsvInterleavedBatch_buffersize returns the size of the buffer, in bytes, required in SgtsvInterleavedBatch().

```
integer(4) function cusparseSgtsvInterleavedBatch_bufferSize(handle, algo, m,
dl, d, du, x, batchCount, pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer(4) :: algo, m, batchCount
  real(4), device :: dl(*), d(*), du(*), x(*)
```

```
  integer(8) :: pBufferSizeInBytes
```

5.6.46. cusparseDgtsvInterleavedBatch_buffersize

DgtsvInterleavedBatch_buffersize returns the size of the buffer, in bytes, required in DgtsvInterleavedBatch().

```
integer(4) function cusparseDgtsvInterleavedBatch_bufferSize(handle, algo, m,
dl, d, du, x, batchCount, pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer(4) :: algo, m, batchCount
  real(8), device :: dl(*), d(*), du(*), x(*)
  integer(8) :: pBufferSizeInBytes
```

5.6.47. cusparseCgtsvInterleavedBatch_buffersize

CgtsvInterleavedBatch_buffersize returns the size of the buffer, in bytes, required in CgtsvInterleavedBatch().

```
integer(4) function cusparseCgtsvInterleavedBatch_bufferSize(handle, algo, m,
dl, d, du, x, batchCount, pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer(4) :: algo, m, batchCount
  complex(4), device :: dl(*), d(*), du(*), x(*)
  integer(8) :: pBufferSizeInBytes
```

5.6.48. cusparseZgtsvInterleavedBatch_buffersize

ZgtsvInterleavedBatch_buffersize returns the size of the buffer, in bytes, required in ZgtsvInterleavedBatch().

```
integer(4) function cusparseZgtsvInterleavedBatch_bufferSize(handle, algo, m,
dl, d, du, x, batchCount, pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer(4) :: algo, m, batchCount
  complex(8), device :: dl(*), d(*), du(*), x(*)
  integer(8) :: pBufferSizeInBytes
```

5.6.49. cusparseSgtsvInterleavedBatch

SgtsvInterleavedBatch computes the solution of multiple tridiagonal linear systems with multiple right hand sides: A * X = B The coefficient matrix A of each tri-diagonal linear system is defined with three vectors corresponding to its lower (dl), main (d), and upper (du) matrix diagonals; the right-hand sides are stored in the dense matrix B. The solution X overwrites the righthand-side matrix B on exit. This routine differs from the SgtsvStridedBatch routines in that the data for the diagonals, RHS, and solution vectors are interleaved, from one to batchCount, rather than stored one after another. See the CUSPARSE_Library document for currently supported algo values.

```
integer(4) function cusparseSgtsvInterleavedBatch(handle, algo, m, dl, d, du,
x, batchCount, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: algo, m, batchCount
  real(4), device :: dl(*), d(*), du(*), x(*)
  character(1), device :: pBuffer(*)
```

5.6.50. cusparseDgtsvInterleavedBatch

DgtsvInterleavedBatch computes the solution of multiple tridiagonal linear systems with multiple right hand sides: A * X = B The coefficient matrix A of each tri-diagonal linear system is defined with three vectors corresponding to its lower (dl), main (d), and upper (du) matrix diagonals; the right-hand sides are stored in the dense matrix B. The solution X overwrites the righthand-side matrix B on exit. This routine differs from the DgtsvStridedBatch routines in that the data for the diagonals, RHS, and solution vectors are interleaved, from one to batchCount, rather than stored one after another. See the CUSPARSE_Library document for currently supported algo values.

```
integer(4) function cusparseDgtsvInterleavedBatch(handle, algo, m, dl, d, du,
x, batchCount, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: algo, m, batchCount
  real(8), device :: dl(*), d(*), du(*), x(*)
  character(1), device :: pBuffer(*)
```

5.6.51. cusparseCgtsvInterleavedBatch

CgtsvInterleavedBatch computes the solution of multiple tridiagonal linear systems with multiple right hand sides: A * X = B The coefficient matrix A of each tri-diagonal linear system is defined with three vectors corresponding to its lower (dl), main (d), and upper (du) matrix diagonals; the right-hand sides are stored in the dense matrix B. The solution X overwrites the righthand-side matrix B on exit. This routine differs from the CgtsvStridedBatch routines in that the data for the diagonals, RHS, and solution vectors are interleaved, from one to batchCount, rather than stored one after another. See the CUSPARSE_Library document for currently supported algo values.

```
integer(4) function cusparseCgtsvInterleavedBatch(handle, algo, m, dl, d, du,
x, batchCount, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: algo, m, batchCount
  complex(4), device :: dl(*), d(*), du(*), x(*)
  character(1), device :: pBuffer(*)
```

5.6.52. cusparseZgtsvInterleavedBatch

ZgtsvInterleavedBatch computes the solution of multiple tridiagonal linear systems with multiple right hand sides: A * X = B The coefficient matrix A of each tri-diagonal linear system is defined with three vectors corresponding to its lower (dl), main (d), and upper (du) matrix diagonals; the right-hand sides are stored in the dense matrix B. The solution X overwrites the righthand-side matrix B on exit. This routine differs from the ZgtsvStridedBatch routines in that the data for the diagonals, RHS, and solution vectors are interleaved, from one to batchCount, rather than stored one after another. See the CUSPARSE_Library document for currently supported algo values.

```
integer(4) function cusparseZgtsvInterleavedBatch(handle, algo, m, dl, d, du,
x, batchCount, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: algo, m, batchCount
  complex(8), device :: dl(*), d(*), du(*), x(*)
  character(1), device :: pBuffer(*)
```

5.6.53. cusparseSgpsvInterleavedBatch_buffersize

SgpsvInterleavedBatch_buffersize returns the size of the buffer, in bytes, required in SgpsvInterleavedBatch().

```
integer(4) function cusparseSgpsvInterleavedBatch_bufferSize(handle, algo, m,
ds, dl, d, du, dw, x, batchCount, pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer(4) :: algo, m, batchCount
  real(4), device :: ds(*), dl(*), d(*), du(*), dw(*), x(*)
  integer(8) :: pBufferSizeInBytes
```

5.6.54. cusparseDgpsvInterleavedBatch_buffersize

DgpsvInterleavedBatch_buffersize returns the size of the buffer, in bytes, required in DgpsvInterleavedBatch().

```
integer(4) function cusparseDgpsvInterleavedBatch_bufferSize(handle, algo, m,
ds, dl, d, du, dw, x, batchCount, pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer(4) :: algo, m, batchCount
  real(8), device :: ds(*), dl(*), d(*), du(*), dw(*), x(*)
  integer(8) :: pBufferSizeInBytes
```

5.6.55. cusparseCgpsvInterleavedBatch_buffersize

CgpsvInterleavedBatch_buffersize returns the size of the buffer, in bytes, required in CgpsvInterleavedBatch().

```
integer(4) function cusparseCgpsvInterleavedBatch_bufferSize(handle, algo, m,
ds, dl, d, du, dw, x, batchCount, pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer(4) :: algo, m, batchCount
  complex(4), device :: ds(*), dl(*), d(*), du(*), dw(*), x(*)
  integer(8) :: pBufferSizeInBytes
```

5.6.56. cusparseZgpsvInterleavedBatch_buffersize

ZgpsvInterleavedBatch_buffersize returns the size of the buffer, in bytes, required in ZgpsvInterleavedBatch().

```
integer(4) function cusparseZgpsvInterleavedBatch_bufferSize(handle, algo, m,
ds, dl, d, du, dw, x, batchCount, pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer(4) :: algo, m, batchCount
  complex(8), device :: ds(*), dl(*), d(*), du(*), dw(*), x(*)
  integer(8) :: pBufferSizeInBytes
```

5.6.57. cusparseSgpsvInterleavedBatch

SgpsvInterleavedBatch computes the solution of multiple pentadiagonal linear systems with multiple right hand sides: A * X = B The coefficient matrix A of each penta-diagonal linear system is defined with five vectors corresponding to its lower (ds, dl), main (d), and upper (du, dw) matrix diagonals; the right-hand sides are stored in the dense matrix B. The solution X overwrites the righthand-side matrix B on exit. This routine differs from StridedBatch routines in that the data for the diagonals, RHS, and solution vectors

are interleaved, from one to batchCount, rather than stored one after another. See the CUSPARSE_Library document for currently supported algo values.

```
integer(4) function cusparseSgpsvInterleavedBatch(handle, algo, m, ds, dl, d,
du, dw, x, batchCount, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: algo, m, batchCount
  real(4), device :: ds(*), dl(*), d(*), du(*), dw(*), x(*)
  character(1), device :: pBuffer(*)
```

5.6.58. cusparseDgpsvInterleavedBatch

DgpsvInterleavedBatch computes the solution of multiple pentadiagonal linear systems with multiple right hand sides: A * X = B The coefficient matrix A of each penta-diagonal linear system is defined with five vectors corresponding to its lower (ds, dl), main (d), and upper (du, dw) matrix diagonals; the right-hand sides are stored in the dense matrix B. The solution X overwrites the righthand-side matrix B on exit. This routine differs from StridedBatch routines in that the data for the diagonals, RHS, and solution vectors are interleaved, from one to batchCount, rather than stored one after another. See the CUSPARSE_Library document for currently supported algo values.

```
integer(4) function cusparseDgpsvInterleavedBatch(handle, algo, m, ds, dl, d,
du, dw, x, batchCount, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: algo, m, batchCount
  real(8), device :: ds(*), dl(*), d(*), du(*), dw(*), x(*)
  character(1), device :: pBuffer(*)
```

5.6.59. cusparseCgpsvInterleavedBatch

CgpsvInterleavedBatch computes the solution of multiple pentadiagonal linear systems with multiple right hand sides: A * X = B The coefficient matrix A of each penta-diagonal linear system is defined with five vectors corresponding to its lower (ds, dl), main (d), and upper (du, dw) matrix diagonals; the right-hand sides are stored in the dense matrix B. The solution X overwrites the righthand-side matrix B on exit. This routine differs from StridedBatch routines in that the data for the diagonals, RHS, and solution vectors are interleaved, from one to batchCount, rather than stored one after another. See the CUSPARSE_Library document for currently supported algo values.

```
integer(4) function cusparseCgpsvInterleavedBatch(handle, algo, m, ds, dl, d,
du, dw, x, batchCount, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: algo, m, batchCount
  complex(4), device :: ds(*), dl(*), d(*), du(*), dw(*), x(*)
  character(1), device :: pBuffer(*)
```

5.6.60. cusparseZgpsvInterleavedBatch

ZgpsvInterleavedBatch computes the solution of multiple pentadiagonal linear systems with multiple right hand sides: A * X = B The coefficient matrix A of each penta-diagonal linear system is defined with five vectors corresponding to its lower (ds, dl), main (d), and upper (du, dw) matrix diagonals; the right-hand sides are stored in the dense matrix B. The solution X overwrites the righthand-side matrix B on exit. This routine differs from StridedBatch routines in that the data for the diagonals, RHS, and solution vectors

are interleaved, from one to batchCount, rather than stored one after another. See the CUSPARSE_Library document for currently supported algo values.

```
integer(4) function cusparseZgpsvInterleavedBatch(handle, algo, m, ds, dl, d,
du, dw, x, batchCount, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: algo, m, batchCount
  complex(8), device :: ds(*), dl(*), d(*), du(*), dw(*), x(*)
  character(1), device :: pBuffer(*)
```

5.6.61. cusparseScsric02_bufferSize

This function returns the size of the buffer used in csric02.

```
integer(4) function cusparseScsric02_bufferSize(handle, m, nnz, descrA,
csrValA, csrRowPtrA, csrColIndA, info, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descrA
  real(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsric02Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

5.6.62. cusparseDcsric02_bufferSize

This function returns the size of the buffer used in csric02.

```
integer(4) function cusparseDcsric02_bufferSize(handle, m, nnz, descrA,
csrValA, csrRowPtrA, csrColIndA, info, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descrA
  real(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsric02Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

5.6.63. cusparseCcsric02_bufferSize

This function returns the size of the buffer used in csric02.

```
integer(4) function cusparseCcsric02_bufferSize(handle, m, nnz, descrA,
csrValA, csrRowPtrA, csrColIndA, info, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descrA
  complex(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsric02Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

5.6.64. cusparseZcsric02_bufferSize

This function returns the size of the buffer used in csric02.

```
integer(4) function cusparseZcsric02_bufferSize(handle, m, nnz, descrA,
csrValA, csrRowPtrA, csrColIndA, info, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descrA
  complex(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsric02Info) :: info
```

```
  integer(4) :: pBufferSize ! integer(8) also accepted
```

5.6.65. cusparseScsric02_analysis

This function performs the analysis phase of csric02.

```
integer(4) function cusparseScsric02_analysis(handle, m, nnz, descrA, csrValA,
csrRowPtrA, csrColIndA, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descrA
  real(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsric02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

5.6.66. cusparseDcsric02_analysis

This function performs the analysis phase of csric02.

```
integer(4) function cusparseDcsric02_analysis(handle, m, nnz, descrA, csrValA,
csrRowPtrA, csrColIndA, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descrA
  real(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsric02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

5.6.67. cusparseCcsric02_analysis

This function performs the analysis phase of csric02.

```
integer(4) function cusparseCcsric02_analysis(handle, m, nnz, descrA, csrValA,
csrRowPtrA, csrColIndA, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descrA
  complex(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsric02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

5.6.68. cusparseZcsric02_analysis

This function performs the analysis phase of csric02.

```
integer(4) function cusparseZcsric02_analysis(handle, m, nnz, descrA, csrValA,
csrRowPtrA, csrColIndA, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descrA
  complex(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsric02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

5.6.69. cusparseScsric02

CSRIC02 performs the solve phase of computing the incomplete-Cholesky factorization with zero fill-in and no pivoting.

```
integer(4) function cusparseScsric02(handle, m, nnz, descrA, csrValA,
csrRowPtrA, csrColIndA, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descrA
  real(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsric02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

5.6.70. cusparseDcsric02

CSRIC02 performs the solve phase of computing the incomplete-Cholesky factorization with zero fill-in and no pivoting.

```
integer(4) function cusparseDcsric02(handle, m, nnz, descrA, csrValA,
csrRowPtrA, csrColIndA, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descrA
  real(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsric02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

5.6.71. cusparseCcsric02

CSRIC02 performs the solve phase of computing the incomplete-Cholesky factorization with zero fill-in and no pivoting.

```
integer(4) function cusparseCcsric02(handle, m, nnz, descrA, csrValA,
csrRowPtrA, csrColIndA, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descrA
  complex(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsric02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

5.6.72. cusparseZcsric02

CSRIC02 performs the solve phase of computing the incomplete-Cholesky factorization with zero fill-in and no pivoting.

```
integer(4) function cusparseZcsric02(handle, m, nnz, descrA, csrValA,
csrRowPtrA, csrColIndA, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descrA
  complex(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsric02Info) :: info
```

```
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

5.6.73. cusparseXcsric02_zeroPivot

This function returns an error code equal to CUSPARSE_STATUS_ZERO_PIVOT and sets position to j when A(j,j) is either structural zero or numerical zero. Otherwise, position is set to -1.

```
integer(4) function cusparseXcsric02_zeroPivot(handle, info, position)
  type(cusparseHandle) :: handle
  type(cusparseCsric02Info) :: info
  integer(4), device :: position ! device or host variable
```

5.6.74. cusparseScsrilu02_numericBoost

This function boosts the value to replace a numerical value in incomplete LU factorization, based on the tol input argument.

```
integer(4) function cusparseScsrilu02_numericBoost(handle, info, enable_boost,
tol, boost_val)
  type(cusparseHandle) :: handle
  type(cusparseCsrilu02Info) :: info
  integer :: enable_boost
  real(8), device :: tol ! device or host variable
  real(4), device :: boost_val ! device or host variable
```

5.6.75. cusparseDcsrilu02_numericBoost

This function boosts the value to replace a numerical value in incomplete LU factorization, based on the tol input argument.

```
integer(4) function cusparseDcsrilu02_numericBoost(handle, info, enable_boost,
tol, boost_val)
  type(cusparseHandle) :: handle
  type(cusparseCsrilu02Info) :: info
  integer :: enable_boost
  real(8), device :: tol ! device or host variable
  real(8), device :: boost_val ! device or host variable
```

5.6.76. cusparseCcsrilu02_numericBoost

This function boosts the value to replace a numerical value in incomplete LU factorization, based on the tol input argument.

```
integer(4) function cusparseCcsrilu02_numericBoost(handle, info, enable_boost,
tol, boost_val)
  type(cusparseHandle) :: handle
  type(cusparseCsrilu02Info) :: info
  integer :: enable_boost
  real(8), device :: tol ! device or host variable
  complex(4), device :: boost_val ! device or host variable
```

5.6.77. cusparseZcsrilu02_numericBoost

This function boosts the value to replace a numerical value in incomplete LU factorization, based on the tol input argument.

```
integer(4) function cusparseZcsrilu02_numericBoost(handle, info, enable_boost,
tol, boost_val)
  type(cusparseHandle) :: handle
```

```
  type(cusparseCsrilu02Info) :: info
  integer :: enable_boost
  real(8), device :: tol ! device or host variable
  complex(8), device :: boost_val ! device or host variable
```

5.6.78. cusparseScsrilu02_bufferSize

This function returns the size of the buffer used in csrilu02.

```
integer(4) function cusparseScsrilu02_bufferSize(handle, m, nnz, descrA,
csrValA, csrRowPtrA, csrColIndA, info, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descrA
  real(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsrilu02Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

5.6.79. cusparseDcsrilu02_bufferSize

This function returns the size of the buffer used in csrilu02.

```
integer(4) function cusparseDcsrilu02_bufferSize(handle, m, nnz, descrA,
csrValA, csrRowPtrA, csrColIndA, info, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descrA
  real(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsrilu02Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

5.6.80. cusparseCcsrilu02_bufferSize

This function returns the size of the buffer used in csrilu02.

```
integer(4) function cusparseCcsrilu02_bufferSize(handle, m, nnz, descrA,
csrValA, csrRowPtrA, csrColIndA, info, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descrA
  complex(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsrilu02Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

5.6.81. cusparseZcsrilu02_bufferSize

This function returns the size of the buffer used in csrilu02.

```
integer(4) function cusparseZcsrilu02_bufferSize(handle, m, nnz, descrA,
csrValA, csrRowPtrA, csrColIndA, info, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descrA
  complex(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsrilu02Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

5.6.82. cusparseScsrilu02_analysis

This function performs the analysis phase of csrilu02.

```
integer(4) function cusparseScsrilu02_analysis(handle, m, nnz, descrA, csrValA,
csrRowPtrA, csrColIndA, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descrA
  real(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsrilu02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

5.6.83. cusparseDcsrilu02_analysis

This function performs the analysis phase of csrilu02.

```
integer(4) function cusparseDcsrilu02_analysis(handle, m, nnz, descrA, csrValA,
csrRowPtrA, csrColIndA, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descrA
  real(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsrilu02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

5.6.84. cusparseCcsrilu02_analysis

This function performs the analysis phase of csrilu02.

```
integer(4) function cusparseCcsrilu02_analysis(handle, m, nnz, descrA, csrValA,
csrRowPtrA, csrColIndA, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descrA
  complex(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsrilu02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

5.6.85. cusparseZcsrilu02_analysis

This function performs the analysis phase of csrilu02.

```
integer(4) function cusparseZcsrilu02_analysis(handle, m, nnz, descrA, csrValA,
csrRowPtrA, csrColIndA, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descrA
  complex(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsrilu02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

5.6.86. cusparseScsrilu02

CSRILU02 performs the solve phase of the incomplete-LU factorization with zero fill-in and no pivoting. A is an m x m sparse matrix that is defined in CSR storage format by the three arrays csrValA_valM, csrRowPtrA, and csrColIndA.

```
integer(4) function cusparseScsrilu02(handle, m, nnz, descrA,
  type(cusparseHandle) :: handle
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descrA
  real(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsrilu02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

5.6.87. cusparseDcsrilu02

CSRILU02 performs the solve phase of the incomplete-LU factorization with zero fill-in and no pivoting. A is an m x m sparse matrix that is defined in CSR storage format by the three arrays csrValA_valM, csrRowPtrA, and csrColIndA.

```
integer(4) function cusparseDcsrilu02(handle, m, nnz, descrA, csrValA,
csrRowPtrA, csrColIndA, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descrA
  real(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsrilu02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

5.6.88. cusparseCcsrilu02

CSRILU02 performs the solve phase of the incomplete-LU factorization with zero fill-in and no pivoting. A is an m x m sparse matrix that is defined in CSR storage format by the three arrays csrValA_valM, csrRowPtrA, and csrColIndA.

```
integer(4) function cusparseCcsrilu02(handle, m, nnz, descrA, csrValA,
csrRowPtrA, csrColIndA, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descrA
  complex(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsrilu02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

5.6.89. cusparseZcsrilu02

CSRILU02 performs the solve phase of the incomplete-LU factorization with zero fill-in and no pivoting. A is an m x m sparse matrix that is defined in CSR storage format by the three arrays csrValA_valM, csrRowPtrA, and csrColIndA.

```
integer(4) function cusparseZcsrilu02(handle, m, nnz, descrA, csrValA,
csrRowPtrA, csrColIndA, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, nnz
```

```
  type(cusparseMatDescr) :: descrA
  complex(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsrilu02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

5.6.90. cusparseXcsrilu02_zeroPivot

This function returns an error code equal to CUSPARSE_STATUS_ZERO_PIVOT and sets position to j when A(j,j) is either structural zero or numerical zero. Otherwise, position is set to -1.

```
integer(4) function cusparseXcsrilu02_zeroPivot(handle, info, position)
  type(cusparseHandle) :: handle
  type(cusparseCsrilu02Info) :: info
  integer(4), device :: position ! device or host variable
```

5.6.91. cusparseSbsric02_bufferSize

This function returns the size of the buffer used in bsric02.

```
integer(4) function cusparseSbsric02_bufferSize(handle, dirA, mb, nnzb, descrA,
bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  real(4), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsric02Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

5.6.92. cusparseDbsric02_bufferSize

This function returns the size of the buffer used in bsric02.

```
integer(4) function cusparseDbsric02_bufferSize(handle, dirA, mb, nnzb, descrA,
bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  real(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsric02Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

5.6.93. cusparseCbsric02_bufferSize

This function returns the size of the buffer used in bsric02.

```
integer(4) function cusparseCbsric02_bufferSize(handle, dirA, mb, nnzb, descrA,
bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  complex(4), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
```

```
  type(cusparseBsric02Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

5.6.94. cusparseZbsric02_bufferSize

This function returns the size of the buffer used in bsric02.

```
integer(4) function cusparseZbsric02_bufferSize(handle, dirA, mb, nnzb, descrA,
bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  complex(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsric02Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

5.6.95. cusparseSbsric02_analysis

This function performs the analysis phase of bsric02.

```
integer(4) function cusparseSbsric02_analysis(handle, dirA, mb, nnzb, descrA,
bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  real(4), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsric02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

5.6.96. cusparseDbsric02_analysis

This function performs the analysis phase of bsric02.

```
integer(4) function cusparseDbsric02_analysis(handle, dirA, mb, nnzb, descrA,
bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  real(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsric02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

5.6.97. cusparseCbsric02_analysis

This function performs the analysis phase of bsric02.

```
integer(4) function cusparseCbsric02_analysis(handle, dirA, mb, nnzb, descrA,
bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  complex(4), device :: bsrValA(*)
```

```
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsric02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

5.6.98. cusparseZbsric02_analysis

This function performs the analysis phase of bsric02.

```
integer(4) function cusparseZbsric02_analysis(handle, dirA, mb, nnzb, descrA,
bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  complex(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsric02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

5.6.99. cusparseSbsric02

BSRIC02 performs the solve phase of the incomplete-Cholesky factorization with zero fill-in and no pivoting. A is an (mb*blockDim) x (mb*blockDim) sparse matrix that is defined in BSR storage format by the three arrays bsrValA, bsrRowPtrA, and bsrColIndA.

```
integer(4) function cusparseSbsric02(handle, dirA, mb, nnzb, descrA, bsrValA,
bsrRowPtrA, bsrColIndA, blockDim, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  real(4), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsric02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

5.6.100. cusparseDbsric02

BSRIC02 performs the solve phase of the incomplete-Cholesky factorization with zero fill-in and no pivoting. A is an (mb*blockDim) x (mb*blockDim) sparse matrix that is defined in BSR storage format by the three arrays bsrValA, bsrRowPtrA, and bsrColIndA.

```
integer(4) function cusparseDbsric02(handle, dirA, mb, nnzb, descrA, bsrValA,
bsrRowPtrA, bsrColIndA, blockDim, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  real(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsric02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

5.6.101. cusparseCbsric02

BSRIC02 performs the solve phase of the incomplete-Cholesky factorization with zero fill-in and no pivoting. A is an (mb*blockDim) x (mb*blockDim) sparse matrix that is defined in BSR storage format by the three arrays bsrValA, bsrRowPtrA, and bsrColIndA.

```
integer(4) function cusparseCbsric02(handle, dirA, mb, nnzb, descrA, bsrValA,
bsrRowPtrA, bsrColIndA, blockDim, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  complex(4), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsric02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

5.6.102. cusparseZbsric02

BSRIC02 performs the solve phase of the incomplete-Cholesky factorization with zero fill-in and no pivoting. A is an (mb*blockDim) x (mb*blockDim) sparse matrix that is defined in BSR storage format by the three arrays bsrValA, bsrRowPtrA, and bsrColIndA.

```
integer(4) function cusparseZbsric02(handle, dirA, mb, nnzb, descrA, bsrValA,
bsrRowPtrA, bsrColIndA, blockDim, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  complex(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsric02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

5.6.103. cusparseXbsric02_zeroPivot

This function returns an error code equal to CUSPARSE_STATUS_ZERO_PIVOT and sets position to j when A(j,j) is either structural zero or numerical zero. Otherwise, position is set to -1.

```
integer(4) function cusparseXbsric02_zeroPivot(handle, info, position)
  type(cusparseHandle) :: handle
  type(cusparseBsric02Info) :: info
  integer(4), device :: position ! device or host variable
```

5.6.104. cusparseSbsrilu02_numericBoost

This function boosts the value to replace a numerical value in incomplete LU factorization, based on the tol input argument.

```
integer(4) function cusparseSbsrilu02_numericBoost(handle, info, enable_boost,
tol, boost_val)
  type(cusparseHandle) :: handle
  type(cusparseBsrilu02Info) :: info
```

```
  integer :: enable_boost
  real(8), device :: tol ! device or host variable
  real(4), device :: boost_val ! device or host variable
```

5.6.105. cusparseDbsrilu02_numericBoost

This function boosts the value to replace a numerical value in incomplete LU factorization, based on the tol input argument.

```
integer(4) function cusparseDbsrilu02_numericBoost(handle, info, enable_boost,
tol, boost_val)
  type(cusparseHandle) :: handle
  type(cusparseBsrilu02Info) :: info
  integer :: enable_boost
  real(8), device :: tol ! device or host variable
  real(8), device :: boost_val ! device or host variable
```

5.6.106. cusparseCbsrilu02_numericBoost

This function boosts the value to replace a numerical value in incomplete LU factorization, based on the tol input argument.

```
integer(4) function cusparseCbsrilu02_numericBoost(handle, info, enable_boost,
tol, boost_val)
  type(cusparseHandle) :: handle
  type(cusparseBsrilu02Info) :: info
  integer :: enable_boost
  real(8), device :: tol ! device or host variable
  complex(4), device :: boost_val ! device or host variable
```

5.6.107. cusparseZbsrilu02_numericBoost

This function boosts the value to replace a numerical value in incomplete LU factorization, based on the tol input argument.

```
integer(4) function cusparseZbsrilu02_numericBoost(handle, info, enable_boost,
tol, boost_val)
  type(cusparseHandle) :: handle
  type(cusparseBsrilu02Info) :: info
  integer :: enable_boost
  real(8), device :: tol ! device or host variable
  complex(8), device :: boost_val ! device or host variable
```

5.6.108. cusparseSbsrilu02_bufferSize

This function returns the size of the buffer used in bsrilu02.

```
integer(4) function cusparseSbsrilu02_bufferSize(handle, dirA, mb, nnzb,
descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  real(4), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsrilu02Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

5.6.109. cusparseDbsrilu02_bufferSize

This function returns the size of the buffer used in bsrilu02.

```
integer(4) function cusparseDbsrilu02_bufferSize(handle, dirA, mb, nnzb,
descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  real(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsrilu02Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

5.6.110. cusparseCbsrilu02_bufferSize

This function returns the size of the buffer used in bsrilu02.

```
integer(4) function cusparseCbsrilu02_bufferSize(handle, dirA, mb, nnzb,
descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  complex(4), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsrilu02Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

5.6.111. cusparseZbsrilu02_bufferSize

This function returns the size of the buffer used in bsrilu02.

```
integer(4) function cusparseZbsrilu02_bufferSize(handle, dirA, mb, nnzb,
descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  complex(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsrilu02Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

5.6.112. cusparseSbsrilu02_analysis

This function performs the analysis phase of bsrilu02.

```
integer(4) function cusparseSbsrilu02_analysis(handle, dirA, mb, nnzb, descrA,
bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  real(4), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsrilu02Info) :: info
  integer(4) :: policy
```

```
  character(c_char), device :: pBuffer(*)
```

5.6.113. cusparseDbsrilu02_analysis

This function performs the analysis phase of bsrilu02.

```
integer(4) function cusparseDbsrilu02_analysis(handle, dirA, mb, nnzb, descrA,
bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  real(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsrilu02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

5.6.114. cusparseCbsrilu02_analysis

This function performs the analysis phase of bsrilu02.

```
integer(4) function cusparseCbsrilu02_analysis(handle, dirA, mb, nnzb, descrA,
bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  complex(4), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsrilu02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

5.6.115. cusparseZbsrilu02_analysis

This function performs the analysis phase of bsrilu02.

```
integer(4) function cusparseZbsrilu02_analysis(handle, dirA, mb, nnzb, descrA,
bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  complex(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsrilu02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

5.6.116. cusparseSbsrilu02

BSRILU02 performs the solve phase of the incomplete-LU factorization with zero fill-in and no pivoting. A is an (mb*blockDim) x (mb*blockDim) sparse matrix that is defined in BSR storage format by the three arrays bsrValA, bsrRowPtrA, and bsrColIndA.

```
integer(4) function cusparseSbsrilu02(handle, dirA, mb, nnzb, descrA, bsrValA,
bsrRowPtrA, bsrColIndA, blockDim, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dirA
```

```
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  real(4), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsrilu02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

5.6.117. cusparseDbsrilu02

BSRILU02 performs the solve phase of the incomplete-LU factorization with zero fill-in and no pivoting. A is an (mb*blockDim) x (mb*blockDim) sparse matrix that is defined in BSR storage format by the three arrays bsrValA, bsrRowPtrA, and bsrColIndA.

```
integer(4) function cusparseDbsrilu02(handle, dirA, mb, nnzb, descrA, bsrValA,
bsrRowPtrA, bsrColIndA, blockDim, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  real(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsrilu02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

5.6.118. cusparseCbsrilu02

BSRILU02 performs the solve phase of the incomplete-LU factorization with zero fill-in and no pivoting. A is an (mb*blockDim) x (mb*blockDim) sparse matrix that is defined in BSR storage format by the three arrays bsrValA, bsrRowPtrA, and bsrColIndA.

```
integer(4) function cusparseCbsrilu02(handle, dirA, mb, nnzb, descrA, bsrValA,
bsrRowPtrA, bsrColIndA, blockDim, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  complex(4), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsrilu02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

5.6.119. cusparseZbsrilu02

BSRILU02 performs the solve phase of the incomplete-LU factorization with zero fill-in and no pivoting. A is an (mb*blockDim) x (mb*blockDim) sparse matrix that is defined in BSR storage format by the three arrays bsrValA, bsrRowPtrA, and bsrColIndA.

```
integer(4) function cusparseZbsrilu02(handle, dirA, mb, nnzb, descrA, bsrValA,
bsrRowPtrA, bsrColIndA, blockDim, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  complex(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsrilu02Info) :: info
```

```
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

5.6.120. cusparseXbsrilu02_zeroPivot

This function returns an error code equal to CUSPARSE_STATUS_ZERO_PIVOT and sets position to j when A(j,j) is either structural zero or numerical zero. Otherwise, position is set to -1.

```
integer(4) function cusparseXbsrilu02_zeroPivot(handle, info, position)
  type(cusparseHandle) :: handle
  type(cusparseBsrilu02Info) :: info
  integer(4), device :: position ! device or host variable
```

5.7. CUSPARSE Reordering Functions

This section contains interfaces for the reordering functions that are used to manipulate sparse matrices.

5.7.1. cusparseScsrColor

This function performs the coloring of the adjacency graph associated with the matrix A stored in CSR format.

```
integer(4) function cusparseScsrColor(handle, m, nnz, descrA, csrValA,
csrRowPtrA, csrColIndA, fractionToColor, ncolors, coloring, reordering, info)
  type(cusparseHandle) :: handle
  type(cusparseMatDescr) :: descrA
  type(cusparseColorInfo) :: info
  integer :: m, nnz
  real(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), coloring(*),
reordering(*)
  real(4), device :: fractionToColor ! device or host variable
  integer(4), device :: ncolors ! device or host variable
```

5.7.2. cusparseDcsrColor

This function performs the coloring of the adjacency graph associated with the matrix A stored in CSR format.

```
integer(4) function cusparseDcsrColor(handle, m, nnz, descrA, csrValA,
csrRowPtrA, csrColIndA, fractionToColor, ncolors, coloring, reordering, info)
  type(cusparseHandle) :: handle
  type(cusparseMatDescr) :: descrA
  type(cusparseColorInfo) :: info
  integer :: m, nnz
  real(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), coloring(*),
reordering(*)
  real(8), device :: fractionToColor ! device or host variable
  integer(4), device :: ncolors ! device or host variable
```

5.7.3. cusparseCcsrColor

This function performs the coloring of the adjacency graph associated with the matrix A stored in CSR format.

```
integer(4) function cusparseCcsrColor(handle, m, nnz, descrA, csrValA,
csrRowPtrA, csrColIndA, fractionToColor, ncolors, coloring, reordering, info)
  type(cusparseHandle) :: handle
  type(cusparseMatDescr) :: descrA
  type(cusparseColorInfo) :: info
  integer :: m, nnz
  complex(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), coloring(*),
reordering(*)
  real(4), device :: fractionToColor ! device or host variable
  integer(4), device :: ncolors ! device or host variable
```

5.7.4. cusparseZcsrColor

This function performs the coloring of the adjacency graph associated with the matrix A stored in CSR format.

```
integer(4) function cusparseZcsrColor(handle, m, nnz, descrA, csrValA,
csrRowPtrA, csrColIndA, fractionToColor, ncolors, coloring, reordering, info)
  type(cusparseHandle) :: handle
  type(cusparseMatDescr) :: descrA
  type(cusparseColorInfo) :: info
  integer :: m, nnz
  complex(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), coloring(*),
reordering(*)
  real(8), device :: fractionToColor ! device or host variable
  integer(4), device :: ncolors ! device or host variable
```

5.8. CUSPARSE Format Conversion Functions

This section contains interfaces for the conversion functions that are used to switch between different sparse and dense matrix storage formats.

5.8.1. cusparseSbsr2csr

This function converts a sparse matrix in BSR format that is defined by the three arrays bsrValA, bsrRowPtrA, and bsrColIndA into a sparse matrix in CSR format that is defined by the arrays csrValC, csrRowPtrC, and csrColIndC.

```
integer(4) function cusparseSbsr2csr(handle, dirA, nm, nb, descrA, bsrValA,
bsrRowPtrA, bsrColIndA, blockDim, descrC, csrValC, csrRowPtrC, csrColIndC)
  type(cusparseHandle) :: handle
  integer(4) :: dirA, mb, nb, blockDim
  type(cusparseMatDescr) :: descrA, descrC
  real(4), device :: bsrValA(*), csrValC(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*), csrRowPtrC(*),
csrColIndC(*)
```

5.8.2. cusparseDbsr2csr

This function converts a sparse matrix in BSR format that is defined by the three arrays bsrValA, bsrRowPtrA, and bsrColIndA into a sparse matrix in CSR format that is defined by the arrays csrValC, csrRowPtrC, and csrColIndC.

```
integer(4) function cusparseDbsr2csr(handle, dirA, nm, nb, descrA, bsrValA,
bsrRowPtrA, bsrColIndA, blockDim, descrC, csrValC, csrRowPtrC, csrColIndC)
  type(cusparseHandle) :: handle
  integer(4) :: dirA, mb, nb, blockDim
  type(cusparseMatDescr) :: descrA, descrC
  real(8), device :: bsrValA(*), csrValC(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*), csrRowPtrC(*),
csrColIndC(*)
```

5.8.3. cusparseCbsr2csr

This function converts a sparse matrix in BSR format that is defined by the three arrays bsrValA, bsrRowPtrA, and bsrColIndA into a sparse matrix in CSR format that is defined by the arrays csrValC, csrRowPtrC, and csrColIndC.

```
integer(4) function cusparseCbsr2csr(handle, dirA, nm, nb, descrA, bsrValA,
bsrRowPtrA, bsrColIndA, blockDim, descrC, csrValC, csrRowPtrC, csrColIndC)
  type(cusparseHandle) :: handle
  integer(4) :: dirA, mb, nb, blockDim
  type(cusparseMatDescr) :: descrA, descrC
  complex(4), device :: bsrValA(*), csrValC(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*), csrRowPtrC(*),
csrColIndC(*)
```

5.8.4. cusparseZbsr2csr

This function converts a sparse matrix in BSR format that is defined by the three arrays bsrValA, bsrRowPtrA, and bsrColIndA into a sparse matrix in CSR format that is defined by the arrays csrValC, csrRowPtrC, and csrColIndC.

```
integer(4) function cusparseZbsr2csr(handle, dirA, nm, nb, descrA, bsrValA,
bsrRowPtrA, bsrColIndA, blockDim, descrC, csrValC, csrRowPtrC, csrColIndC)
  type(cusparseHandle) :: handle
  integer(4) :: dirA, mb, nb, blockDim
  type(cusparseMatDescr) :: descrA, descrC
  complex(8), device :: bsrValA(*), csrValC(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*), csrRowPtrC(*),
csrColIndC(*)
```

5.8.5. cusparseXcoo2csr

This function converts the array containing the uncompressed row indices (corresponding to COO format) into an array of compressed row pointers (corresponding to CSR format).

```
integer(4) function cusparseXcoo2csr(handle, cooRowInd, nnz, m, csrRowPtr,
idxBase)
  type(cusparseHandle) :: handle
  integer(4) :: nnz, m, idxBase
  integer(4), device :: cooRowInd(*), csrRowPtr(*)
```

5.8.6. cusparseScsc2dense

This function converts the sparse matrix in CSC format that is defined by the three arrays cscValA, cscColPtrA, and cscRowIndA into the matrix A in dense format. The dense matrix A is filled in with the values of the sparse matrix and with zeros elsewhere.

```
integer(4) function cusparseScsc2dense(handle, m, n, descrA, cscValA,
cscRowIndA, cscColPtrA, A, lda)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, lda
  type(cusparseMatDescr) :: descrA
  real(4), device :: cscValA(*), A(*)
  integer(4), device :: cscRowIndA(*), cscColPtrA(*)
```

5.8.7. cusparseDcsc2dense

This function converts the sparse matrix in CSC format that is defined by the three arrays cscValA, cscColPtrA, and cscRowIndA into the matrix A in dense format. The dense matrix A is filled in with the values of the sparse matrix and with zeros elsewhere.

```
integer(4) function cusparseDcsc2dense(handle, m, n, descrA, cscValA,
cscRowIndA, cscColPtrA, A, lda)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, lda
  type(cusparseMatDescr) :: descrA
  real(8), device :: cscValA(*), A(*)
  integer(4), device :: cscRowIndA(*), cscColPtrA(*)
```

5.8.8. cusparseCcsc2dense

This function converts the sparse matrix in CSC format that is defined by the three arrays cscValA, cscColPtrA, and cscRowIndA into the matrix A in dense format. The dense matrix A is filled in with the values of the sparse matrix and with zeros elsewhere.

```
integer(4) function cusparseCcsc2dense(handle, m, n, descrA, cscValA,
cscRowIndA, cscColPtrA, A, lda)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, lda
  type(cusparseMatDescr) :: descrA
  complex(4), device :: cscValA(*), A(*)
  integer(4), device :: cscRowIndA(*), cscColPtrA(*)
```

5.8.9. cusparseZcsc2dense

This function converts the sparse matrix in CSC format that is defined by the three arrays cscValA, cscColPtrA, and cscRowIndA into the matrix A in dense format. The dense matrix A is filled in with the values of the sparse matrix and with zeros elsewhere.

```
integer(4) function cusparseZcsc2dense(handle, m, n, descrA, cscValA,
cscRowIndA, cscColPtrA, A, lda)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, lda
  type(cusparseMatDescr) :: descrA
  complex(8), device :: cscValA(*), A(*)
  integer(4), device :: cscRowIndA(*), cscColPtrA(*)
```

5.8.10. cusparseScsc2hyb

This function converts the sparse matrix in CSC format that is defined by the three arrays cscValA, cscColPtrA, and cscRowIndA into the sparse matrix A in HYB format.

```
integer(4) function cusparseScsc2hyb(handle, m, n, descrA, cscValA, cscRowIndA,
cscColPtrA, hybA, userEllWidth, partitionType)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, userEllWidth, partitionType
  type(cusparseMatDescr) :: descrA
  real(4), device :: cscValA(*)
  integer(4), device :: cscRowIndA(*), cscColPtrA(*)
  type(cusparseHybMat) :: hybA
```

5.8.11. cusparseDcsc2hyb

This function converts the sparse matrix in CSC format that is defined by the three arrays cscValA, cscColPtrA, and cscRowIndA into the sparse matrix A in HYB format.

```
integer(4) function cusparseDcsc2hyb(handle, m, n, descrA, cscValA, cscRowIndA,
cscColPtrA, hybA, userEllWidth, partitionType)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, userEllWidth, partitionType
  type(cusparseMatDescr) :: descrA
  real(8), device :: cscValA(*)
  integer(4), device :: cscRowIndA(*), cscColPtrA(*)
  type(cusparseHybMat) :: hybA
```

5.8.12. cusparseCcsc2hyb

This function converts the sparse matrix in CSC format that is defined by the three arrays cscValA, cscColPtrA, and cscRowIndA into the sparse matrix A in HYB format.

```
integer(4) function cusparseCcsc2hyb(handle, m, n, descrA, cscValA, cscRowIndA,
cscColPtrA, hybA, userEllWidth, partitionType)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, userEllWidth, partitionType
  type(cusparseMatDescr) :: descrA
  complex(4), device :: cscValA(*)
  integer(4), device :: cscRowIndA(*), cscColPtrA(*)
  type(cusparseHybMat) :: hybA
```

5.8.13. cusparseZcsc2hyb

This function converts the sparse matrix in CSC format that is defined by the three arrays cscValA, cscColPtrA, and cscRowIndA into the sparse matrix A in HYB format.

```
integer(4) function cusparseZcsc2hyb(handle, m, n, descrA, cscValA, cscRowIndA,
cscColPtrA, hybA, userEllWidth, partitionType)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, userEllWidth, partitionType
  type(cusparseMatDescr) :: descrA
  complex(8), device :: cscValA(*)
  integer(4), device :: cscRowIndA(*), cscColPtrA(*)
  type(cusparseHybMat) :: hybA
```

5.8.14. cusparseXcsr2bsrNnz

cusparseXcsrgeamNnz computes the number of nonzero elements which will be produced by CSRGEAM.

```
integer(4) function cusparseXcsr2bsrNnz(handle, dirA, m, n, descrA, csrRowPtrA,
csrColIndA, blockDim, descrC, bsrRowPtrC, nnzTotalDevHostPtr)
  type(cusparseHandle) :: handle
  integer :: dirA, m, n, blockdim
  type(cusparseMatDescr) :: descrA, descrC
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), bsrRowPtrC(*)
  integer(4), device :: nnzTotalDevHostPtr ! device or host variable
```

5.8.15. cusparseScsr2bsr

This function converts a sparse matrix in CSR storage format that is defined by the three arrays csrValA, csrRowPtrA, and csrColIndA into a sparse matrix in BSR format that is defined by arrays bsrValC, bsrRowPtrC, and bsrColIndC.

```
integer(4) function cusparseScsr2bsr(handle, dirA, m, n, descrA, csrValA,
csrRowPtrA, csrColIndA, blockDim, descrC, bsrValC, bsrRowPtrC, bsrColIndC)
  type(cusparseHandle) :: handle
  integer(4) :: dirA, m, n, blockdim
  type(cusparseMatDescr) :: descrA, descrC
  real(4), device :: csrValA(*), bsrValC(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), bsrRowPtrC(*),
bsrColIndC(*)
```

5.8.16. cusparseDcsr2bsr

This function converts a sparse matrix in CSR storage format that is defined by the three arrays csrValA, csrRowPtrA, and csrColIndA into a sparse matrix in BSR format that is defined by arrays bsrValC, bsrRowPtrC, and bsrColIndC.

```
integer(4) function cusparseDcsr2bsr(handle, dirA, m, n, descrA, csrValA,
csrRowPtrA, csrColIndA, blockDim, descrC, bsrValC, bsrRowPtrC, bsrColIndC)
  type(cusparseHandle) :: handle
  integer(4) :: dirA, m, n, blockdim
  type(cusparseMatDescr) :: descrA, descrC
  real(8), device :: csrValA(*), bsrValC(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), bsrRowPtrC(*),
bsrColIndC(*)
```

5.8.17. cusparseCcsr2bsr

This function converts a sparse matrix in CSR storage format that is defined by the three arrays csrValA, csrRowPtrA, and csrColIndA into a sparse matrix in BSR format that is defined by arrays bsrValC, bsrRowPtrC, and bsrColIndC.

```
integer(4) function cusparseCcsr2bsr(handle, dirA, m, n, descrA, csrValA,
csrRowPtrA, csrColIndA, blockDim, descrC, bsrValC, bsrRowPtrC, bsrColIndC)
  type(cusparseHandle) :: handle
  integer(4) :: dirA, m, n, blockdim
  type(cusparseMatDescr) :: descrA, descrC
  complex(4), device :: csrValA(*), bsrValC(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), bsrRowPtrC(*),
bsrColIndC(*)
```

5.8.18. cusparseZcsr2bsr

This function converts a sparse matrix in CSR storage format that is defined by the three arrays csrValA, csrRowPtrA, and csrColIndA into a sparse matrix in BSR format that is defined by arrays bsrValC, bsrRowPtrC, and bsrColIndC.

```
integer(4) function cusparseZcsr2bsr(handle, dirA, m, n, descrA, csrValA,
csrRowPtrA, csrColIndA, blockDim, descrC, bsrValC, bsrRowPtrC, bsrColIndC)
  type(cusparseHandle) :: handle
  integer(4) :: dirA, m, n, blockdim
  type(cusparseMatDescr) :: descrA, descrC
  complex(8), device :: csrValA(*), bsrValC(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), bsrRowPtrC(*),
bsrColIndC(*)
```

5.8.19. cusparseXcsr2coo

This function converts the array containing the compressed row pointers (corresponding to CSR format) into an array of uncompressed row indices (corresponding to COO format).

```
integer(4) function cusparseXcsr2coo(handle, csrRowPtr, nnz, m, cooRowInd,
idxBase)
  type(cusparseHandle) :: handle
  integer(4) :: nnz, m, idxBase
  integer(4), device :: csrRowPtr(*), cooRowInd(*)
```

5.8.20. cusparseScsr2csc

This function converts a sparse matrix in CSR storage format that is defined by the three arrays csrVal, csrRowPtr, and csrColInd into a sparse matrix in CSC format that is defined by arrays cscVal, cscRowInd, and cscColPtr.

```
integer(4) function cusparseScsr2csc(handle, m, n, nnz, csrVal, csrRowPtr,
csrColInd, cscVal, cscRowInd, cscColPtr, copyValues, idxBase)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, nnz, copyValues, idxBase
  real(4), device :: csrVal(*), cscVal(*)
  integer(4), device :: csrRowPtr(*), csrColInd(*), cscRowInd(*), cscColPtr(*)
```

5.8.21. cusparseDcsr2csc

This function converts a sparse matrix in CSR storage format that is defined by the three arrays csrVal, csrRowPtr, and csrColInd into a sparse matrix in CSC format that is defined by arrays cscVal, cscRowInd, and cscColPtr.

```
integer(4) function cusparseDcsr2csc(handle, m, n, nnz, csrVal, csrRowPtr,
csrColInd, cscVal, cscRowInd, cscColPtr, copyValues, idxBase)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, nnz, copyValues, idxBase
  real(8), device :: csrVal(*), cscVal(*)
  integer(4), device :: csrRowPtr(*), csrColInd(*), cscRowInd(*), cscColPtr(*)
```

5.8.22. cusparseCcsr2csc

This function converts a sparse matrix in CSR storage format that is defined by the three arrays csrVal, csrRowPtr, and csrColInd into a sparse matrix in CSC format that is defined by arrays cscVal, cscRowInd, and cscColPtr.

```
integer(4) function cusparseCcsr2csc(handle, m, n, nnz, csrVal, csrRowPtr,
csrColInd, cscVal, cscRowInd, cscColPtr, copyValues, idxBase)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, nnz, copyValues, idxBase
  complex(4), device :: csrVal(*), cscVal(*)
  integer(4), device :: csrRowPtr(*), csrColInd(*), cscRowInd(*), cscColPtr(*)
```

5.8.23. cusparseZcsr2csc

This function converts a sparse matrix in CSR storage format that is defined by the three arrays csrVal, csrRowPtr, and csrColInd into a sparse matrix in CSC format that is defined by arrays cscVal, cscRowInd, and cscColPtr.

```
integer(4) function cusparseZcsr2csc(handle, m, n, nnz, csrVal, csrRowPtr,
csrColInd, cscVal, cscRowInd, cscColPtr, copyValues, idxBase)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, nnz, copyValues, idxBase
  complex(8), device :: csrVal(*), cscVal(*)
  integer(4), device :: csrRowPtr(*), csrColInd(*), cscRowInd(*), cscColPtr(*)
```

5.8.24. cusparseScsr2dense

This function converts the sparse matrix in CSR format that is defined by the three arrays cscValA, cscRowPtrA, and cscColIndA into the matrix A in dense format. The dense matrix A is filled in with the values of the sparse matrix and with zeros elsewhere.

```
integer(4) function cusparseScsr2dense(handle, m, n, descrA, csrValA,
csrRowPtrA, csrColIndA, A, lda)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, lda
  type(cusparseMatDescr) :: descrA
  real(4), device :: csrValA(*), A(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
```

5.8.25. cusparseDcsr2dense

This function converts the sparse matrix in CSR format that is defined by the three arrays cscValA, cscRowPtrA, and cscColIndA into the matrix A in dense format. The dense matrix A is filled in with the values of the sparse matrix and with zeros elsewhere.

```
integer(4) function cusparseDcsr2dense(handle, m, n, descrA, csrValA,
csrRowPtrA, csrColIndA, A, lda)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, lda
  type(cusparseMatDescr) :: descrA
  real(8), device :: csrValA(*), A(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
```

5.8.26. cusparseCcsr2dense

This function converts the sparse matrix in CSR format that is defined by the three arrays cscValA, cscRowPtrA, and cscColIndA into the matrix A in dense format. The dense matrix A is filled in with the values of the sparse matrix and with zeros elsewhere.

```
integer(4) function cusparseCcsr2dense(handle, m, n, descrA, csrValA,
csrRowPtrA, csrColIndA, A, lda)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, lda
  type(cusparseMatDescr) :: descrA
  complex(4), device :: csrValA(*), A(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
```

5.8.27. cusparseZcsr2dense

This function converts the sparse matrix in CSR format that is defined by the three arrays cscValA, cscRowPtrA, and cscColIndA into the matrix A in dense format. The dense matrix A is filled in with the values of the sparse matrix and with zeros elsewhere.

```
integer(4) function cusparseZcsr2dense(handle, m, n, descrA, csrValA,
csrRowPtrA, csrColIndA, A, lda)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, lda
  type(cusparseMatDescr) :: descrA
  complex(8), device :: csrValA(*), A(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
```

5.8.28. cusparseScsr2hyb

This function converts the sparse matrix in CSR format that is defined by the three arrays cscValA, cscRowPtrA, and cscColIndA into a sparse matrix in HYB format.

```
integer(4) function cusparseScsr2hyb(handle, m, n, descrA, csrValA, csrRowPtrA,
csrColIndA, hybA, userEllWidth, partitionType)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, userEllWidth, partitionType
  type(cusparseMatDescr) :: descrA
  type(cusparseHybMat) :: hybA
  real(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
```

5.8.29. cusparseDcsr2hyb

This function converts the sparse matrix in CSR format that is defined by the three arrays cscValA, cscRowPtrA, and cscColIndA into a sparse matrix in HYB format.

```
integer(4) function cusparseDcsr2hyb(handle, m, n, descrA, csrValA, csrRowPtrA,
csrColIndA, hybA, userEllWidth, partitionType)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, userEllWidth, partitionType
  type(cusparseMatDescr) :: descrA
  type(cusparseHybMat) :: hybA
  real(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
```

5.8.30. cusparseCcsr2hyb

This function converts the sparse matrix in CSR format that is defined by the three arrays cscValA, cscRowPtrA, and cscColIndA into a sparse matrix in HYB format.

```
integer(4) function cusparseCcsr2hyb(handle, m, n, descrA, csrValA, csrRowPtrA,
csrColIndA, hybA, userEllWidth, partitionType)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, userEllWidth, partitionType
  type(cusparseMatDescr) :: descrA
  type(cusparseHybMat) :: hybA
  complex(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
```

5.8.31. cusparseZcsr2hyb

This function converts the sparse matrix in CSR format that is defined by the three arrays cscValA, cscRowPtrA, and cscColIndA into a sparse matrix in HYB format.

```
integer(4) function cusparseZcsr2hyb(handle, m, n, descrA, csrValA, csrRowPtrA,
csrColIndA, hybA, userEllWidth, partitionType)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, userEllWidth, partitionType
  type(cusparseMatDescr) :: descrA
  type(cusparseHybMat) :: hybA
  complex(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
```

5.8.32. cusparseSdense2csc

This function converts the matrix A in dense format into a sparse matrix in CSC format.

```
integer(4) function cusparseSdense2csc(handle, m, n, descrA, A, lda, nnzPerCol,
cscValA, cscRowIndA, cscColPtrA)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, lda
  type(cusparseMatDescr) :: descrA
  real(4), device :: A(*), cscValA(*)
  integer(4), device :: nnzPerCol(*), cscRowIndA(*), cscColPtrA(*)
```

5.8.33. cusparseDdense2csc

This function converts the matrix A in dense format into a sparse matrix in CSC format.

```
integer(4) function cusparseDdense2csc(handle, m, n, descrA, A, lda, nnzPerCol,
cscValA, cscRowIndA, cscColPtrA)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, lda
  type(cusparseMatDescr) :: descrA
  real(8), device :: A(*), cscValA(*)
  integer(4), device :: nnzPerCol(*), cscRowIndA(*), cscColPtrA(*)
```

5.8.34. cusparseCdense2csc

This function converts the matrix A in dense format into a sparse matrix in CSC format.

```
integer(4) function cusparseCdense2csc(handle, m, n, descrA, A, lda, nnzPerCol,
cscValA, cscRowIndA, cscColPtrA)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, lda
  type(cusparseMatDescr) :: descrA
```

```
  complex(4), device :: A(*), cscValA(*)
  integer(4), device :: nnzPerCol(*), cscRowIndA(*), cscColPtrA(*)
```

5.8.35. cusparseZdense2csc

This function converts the matrix A in dense format into a sparse matrix in CSC format.

```
integer(4) function cusparseZdense2csc(handle, m, n, descrA, A, lda, nnzPerCol,
cscValA, cscRowIndA, cscColPtrA)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, lda
  type(cusparseMatDescr) :: descrA
  complex(8), device :: A(*), cscValA(*)
  integer(4), device :: nnzPerCol(*), cscRowIndA(*), cscColPtrA(*)
```

5.8.36. cusparseSdense2csr

This function converts the matrix A in dense format into a sparse matrix in CSR format.

```
integer(4) function cusparseSdense2csr(handle, m, n, descrA, A, lda, nnzPerRow,
csrValA, csrRowPtrA, csrColIndA)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, lda
  type(cusparseMatDescr) :: descrA
  real(4), device :: A(*), csrValA(*)
  integer(4), device :: nnzPerRow(*), csrRowPtrA(*), csrColIndA(*)
```

5.8.37. cusparseDdense2csr

This function converts the matrix A in dense format into a sparse matrix in CSR format.

```
integer(4) function cusparseDdense2csr(handle, m, n, descrA, A, lda, nnzPerRow,
csrValA, csrRowPtrA, csrColIndA)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, lda
  type(cusparseMatDescr) :: descrA
  real(8), device :: A(*), csrValA(*)
  integer(4), device :: nnzPerRow(*), csrRowPtrA(*), csrColIndA(*)
```

5.8.38. cusparseCdense2csr

This function converts the matrix A in dense format into a sparse matrix in CSR format.

```
integer(4) function cusparseCdense2csr(handle, m, n, descrA, A, lda, nnzPerRow,
csrValA, csrRowPtrA, csrColIndA)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, lda
  type(cusparseMatDescr) :: descrA
  complex(4), device :: A(*), csrValA(*)
  integer(4), device :: nnzPerRow(*), csrRowPtrA(*), csrColIndA(*)
```

5.8.39. cusparseZdense2csr

This function converts the matrix A in dense format into a sparse matrix in CSR format.

```
integer(4) function cusparseZdense2csr(handle, m, n, descrA, A, lda, nnzPerRow,
csrValA, csrRowPtrA, csrColIndA)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, lda
  type(cusparseMatDescr) :: descrA
  complex(8), device :: A(*), csrValA(*)
  integer(4), device :: nnzPerRow(*), csrRowPtrA(*), csrColIndA(*)
```

5.8.40. cusparseSdense2hyb

This function converts the matrix A in dense format into a sparse matrix in HYB format.

```
integer(4) function cusparseSdense2hyb(handle, m, n, descrA, A, lda, nnzPerRow,
hybA, userEllWidth, partitionType)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, lda, userEllWidth, partitionType
  type(cusparseMatDescr) :: descrA
  type(cusparseHybMat) :: hybA
  real(4), device :: A(*)
  integer(4), device :: nnzPerRow(*)
```

5.8.41. cusparseDdense2hyb

This function converts the matrix A in dense format into a sparse matrix in HYB format.

```
integer(4) function cusparseDdense2hyb(handle, m, n, descrA, A, lda, nnzPerRow,
hybA, userEllWidth, partitionType)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, lda, userEllWidth, partitionType
  type(cusparseMatDescr) :: descrA
  type(cusparseHybMat) :: hybA
  real(8), device :: A(*)
  integer(4), device :: nnzPerRow(*)
```

5.8.42. cusparseCdense2hyb

This function converts the matrix A in dense format into a sparse matrix in HYB format.

```
integer(4) function cusparseCdense2hyb(handle, m, n, descrA, A, lda, nnzPerRow,
hybA, userEllWidth, partitionType)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, lda, userEllWidth, partitionType
  type(cusparseMatDescr) :: descrA
  type(cusparseHybMat) :: hybA
  complex(4), device :: A(*)
  integer(4), device :: nnzPerRow(*)
```

5.8.43. cusparseZdense2hyb

This function converts the matrix A in dense format into a sparse matrix in HYB format.

```
integer(4) function cusparseZdense2hyb(handle, m, n, descrA, A, lda, nnzPerRow,
hybA, userEllWidth, partitionType)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, lda, userEllWidth, partitionType
  type(cusparseMatDescr) :: descrA
  type(cusparseHybMat) :: hybA
  complex(8), device :: A(*)
  integer(4), device :: nnzPerRow(*)
```

5.8.44. cusparseShyb2csc

This function converts the sparse matrix A in HYB format into a sparse matrix in CSC format.

```
integer(4) function cusparseShyb2csc(handle, descrA, hybA, cscValA, cscRowIndA,
cscColPtrA)
  type(cusparseHandle) :: handle
  type(cusparseMatDescr) :: descrA
  type(cusparseHybMat) :: hybA
```

```
  real(4), device :: cscValA(*)
  integer(4), device :: cscRowIndA(*), cscColPtrA(*)
```

5.8.45. cusparseDhyb2csc

This function converts the sparse matrix A in HYB format into a sparse matrix in CSC format.

```
integer(4) function cusparseDhyb2csc(handle, descrA, hybA, cscValA, cscRowIndA,
cscColPtrA)
  type(cusparseHandle) :: handle
  type(cusparseMatDescr) :: descrA
  type(cusparseHybMat) :: hybA
  real(8), device :: cscValA(*)
  integer(4), device :: cscRowIndA(*), cscColPtrA(*)
```

5.8.46. cusparseChyb2csc

This function converts the sparse matrix A in HYB format into a sparse matrix in CSC format.

```
integer(4) function cusparseChyb2csc(handle, descrA, hybA, cscValA, cscRowIndA,
cscColPtrA)
  type(cusparseHandle) :: handle
  type(cusparseMatDescr) :: descrA
  type(cusparseHybMat) :: hybA
  complex(4), device :: cscValA(*)
  integer(4), device :: cscRowIndA(*), cscColPtrA(*)
```

5.8.47. cusparseZhyb2csc

This function converts the sparse matrix A in HYB format into a sparse matrix in CSC format.

```
integer(4) function cusparseZhyb2csc(handle, descrA, hybA, cscValA, cscRowIndA,
cscColPtrA)
  type(cusparseHandle) :: handle
  type(cusparseMatDescr) :: descrA
  type(cusparseHybMat) :: hybA
  complex(8), device :: cscValA(*)
  integer(4), device :: cscRowIndA(*), cscColPtrA(*)
```

5.8.48. cusparseShyb2csr

This function converts the sparse matrix A in HYB format into a sparse matrix in CSR format.

```
integer(4) function cusparseShyb2csr(handle, descrA, hybA, csrValA, csrRowPtrA,
csrColIndA)
  type(cusparseHandle) :: handle
  type(cusparseMatDescr) :: descrA
  type(cusparseHybMat) :: hybA
  real(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
```

5.8.49. cusparseDhyb2csr

This function converts the sparse matrix A in HYB format into a sparse matrix in CSR format.

```
integer(4) function cusparseDhyb2csr(handle, descrA, hybA, csrValA, csrRowPtrA,
csrColIndA)
  type(cusparseHandle) :: handle
  type(cusparseMatDescr) :: descrA
  type(cusparseHybMat) :: hybA
  real(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
```

5.8.50. cusparseChyb2csr

This function converts the sparse matrix A in HYB format into a sparse matrix in CSR format.

```
integer(4) function cusparseChyb2csr(handle, descrA, hybA, csrValA, csrRowPtrA,
csrColIndA)
  type(cusparseHandle) :: handle
  type(cusparseMatDescr) :: descrA
  type(cusparseHybMat) :: hybA
  complex(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
```

5.8.51. cusparseZhyb2csr

This function converts the sparse matrix A in HYB format into a sparse matrix in CSR format.

```
integer(4) function cusparseZhyb2csr(handle, descrA, hybA, csrValA, csrRowPtrA,
csrColIndA)
  type(cusparseHandle) :: handle
  type(cusparseMatDescr) :: descrA
  type(cusparseHybMat) :: hybA
  complex(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
```

5.8.52. cusparseShyb2dense

This function converts the sparse matrix in HYB format into the matrix A in dense format. The dense matrix A is filled in with the values of the sparse matrix and with zeros elsewhere.

```
integer(4) function cusparseShyb2dense(handle, descrA, hybA, A, lda)
  type(cusparseHandle) :: handle
  type(cusparseMatDescr) :: descrA
  type(cusparseHybMat) :: hybA
  real(4), device :: A(*)
  integer(4) :: lda
```

5.8.53. cusparseDhyb2dense

This function converts the sparse matrix in HYB format into the matrix A in dense format. The dense matrix A is filled in with the values of the sparse matrix and with zeros elsewhere.

```
integer(4) function cusparseDhyb2dense(handle, descrA, hybA, A, lda)
```

```
  type(cusparseHandle) :: handle
  type(cusparseMatDescr) :: descrA
  type(cusparseHybMat) :: hybA
  real(8), device :: A(*)
  integer(4) :: lda
```

5.8.54. cusparseChyb2dense

This function converts the sparse matrix in HYB format into the matrix A in dense format. The dense matrix A is filled in with the values of the sparse matrix and with zeros elsewhere.

```
integer(4) function cusparseChyb2dense(handle, descrA, hybA, A, lda)
  type(cusparseHandle) :: handle
  type(cusparseMatDescr) :: descrA
  type(cusparseHybMat) :: hybA
  complex(4), device :: A(*)
  integer(4) :: lda
```

5.8.55. cusparseZhyb2dense

This function converts the sparse matrix in HYB format into the matrix A in dense format. The dense matrix A is filled in with the values of the sparse matrix and with zeros elsewhere.

```
integer(4) function cusparseZhyb2dense(handle, descrA, hybA, A, lda)
  type(cusparseHandle) :: handle
  type(cusparseMatDescr) :: descrA
  type(cusparseHybMat) :: hybA
  complex(8), device :: A(*)
  integer(4) :: lda
```

5.8.56. cusparseSnnz

This function computes the number of nonzero elements per row or column and the total number of nonzero elements in a dense matrix.

```
integer(4) function cusparseSnnz(handle, dirA, m, n, descrA, A, lda,
nnzPerRowColumn, nnzTotalDevHostPtr)
  type(cusparseHandle) :: handle
  integer :: dirA, m, n, lda
  type(cusparseMatDescr) :: descrA
  real(4), device :: A(*)
  integer(4), device :: nnzPerRowColumn(*)
  integer(4), device :: nnzTotalDevHostPtr ! device or host variable
```

5.8.57. cusparseDnnz

This function computes the number of nonzero elements per row or column and the total number of nonzero elements in a dense matrix.

```
integer(4) function cusparseDnnz(handle, dirA, m, n, descrA, A, lda,
nnzPerRowColumn, nnzTotalDevHostPtr)
  type(cusparseHandle) :: handle
  integer :: dirA, m, n, lda
  type(cusparseMatDescr) :: descrA
  real(8), device :: A(*)
  integer(4), device :: nnzPerRowColumn(*)
  integer(4), device :: nnzTotalDevHostPtr ! device or host variable
```

5.8.58. cusparseCnnz

This function computes the number of nonzero elements per row or column and the total number of nonzero elements in a dense matrix.

```
integer(4) function cusparseCnnz(handle, dirA, m, n, descrA, A, lda,
nnzPerRowColumn, nnzTotalDevHostPtr)
  type(cusparseHandle) :: handle
  integer :: dirA, m, n, lda
  type(cusparseMatDescr) :: descrA
  complex(4), device :: A(*)
  integer(4), device :: nnzPerRowColumn(*)
  integer(4), device :: nnzTotalDevHostPtr ! device or host variable
```

5.8.59. cusparseZnnz

This function computes the number of nonzero elements per row or column and the total number of nonzero elements in a dense matrix.

```
integer(4) function cusparseZnnz(handle, dirA, m, n, descrA, A, lda,
nnzPerRowColumn, nnzTotalDevHostPtr)
  type(cusparseHandle) :: handle
  integer :: dirA, m, n, lda
  type(cusparseMatDescr) :: descrA
  complex(8), device :: A(*)
  integer(4), device :: nnzPerRowColumn(*)
  integer(4), device :: nnzTotalDevHostPtr ! device or host variable
```

5.8.60. cusparseSgebsr2gebsc_bufferSize

This function returns the size of the buffer used in gebsr2gebsc.

```
integer(4) function cusparseSgebsr2gebsc_bufferSize(handle, mb, nb, nnzb,
bsrVal, bsrRowPtr, bsrColInd, rowBlockDim, colBlockDim, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: mb, nb, nnzb
  real(4), device :: bsrVal(*)
  integer(4), device :: bsrRowPtr(*), bsrColInd(*)
  integer(4) :: rowBlockDim, colBlockDim
  integer(4) :: pBufferSize ! integer(8) also accepted
```

5.8.61. cusparseDgebsr2gebsc_bufferSize

This function returns the size of the buffer used in gebsr2gebsc.

```
integer(4) function cusparseDgebsr2gebsc_bufferSize(handle, mb, nb, nnzb,
bsrVal, bsrRowPtr, bsrColInd, rowBlockDim, colBlockDim, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: mb, nb, nnzb
  real(8), device :: bsrVal(*)
  integer(4), device :: bsrRowPtr(*), bsrColInd(*)
  integer(4) :: rowBlockDim, colBlockDim
  integer(4) :: pBufferSize ! integer(8) also accepted
```

5.8.62. cusparseCgebsr2gebsc_bufferSize

This function returns the size of the buffer used in gebsr2gebsc.

```
integer(4) function cusparseCgebsr2gebsc_bufferSize(handle, mb, nb, nnzb,
bsrVal, bsrRowPtr, bsrColInd, rowBlockDim, colBlockDim, pBufferSize)
  type(cusparseHandle) :: handle
```

```
  integer(4) :: mb, nb, nnzb
  complex(4), device :: bsrVal(*)
  integer(4), device :: bsrRowPtr(*), bsrColInd(*)
  integer(4) :: rowBlockDim, colBlockDim
  integer(4) :: pBufferSize ! integer(8) also accepted
```

5.8.63. cusparseZgebsr2gebsc_bufferSize

This function returns the size of the buffer used in gebsr2gebsc.

```
integer(4) function cusparseZgebsr2gebsc_bufferSize(handle, mb, nb, nnzb,
bsrVal, bsrRowPtr, bsrColInd, rowBlockDim, colBlockDim, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: mb, nb, nnzb
  complex(8), device :: bsrVal(*)
  integer(4), device :: bsrRowPtr(*), bsrColInd(*)
  integer(4) :: rowBlockDim, colBlockDim
  integer(4) :: pBufferSize ! integer(8) also accepted
```

5.8.64. cusparseSgebsr2gebsc

This function converts a sparse matrix in general block-CSR storage format to a sparse matrix in general block-CSC storage format.

```
integer(4) function cusparseSgebsr2gebsc(handle, mb, nb, nnzb, bsrVal,
bsrRowPtr, bsrColInd, rowBlockDim, colBlockDim, bscVal, bscRowInd, bscColPtr,
copyValues, baseIdx, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: mb, nb, nnzb
  real(4), device :: bsrVal(*)
  integer(4), device :: bsrRowPtr(*), bsrColInd(*)
  integer(4) :: rowBlockDim, colBlockDim
  real(4), device :: bscVal(*)
  integer(4), device :: bscRowInd(*), bscColPtr(*)
  integer(4) :: copyValues, baseIdx
  character(c_char), device :: pBuffer(*)
```

5.8.65. cusparseDgebsr2gebsc

This function converts a sparse matrix in general block-CSR storage format to a sparse matrix in general block-CSC storage format.

```
integer(4) function cusparseDgebsr2gebsc(handle, mb, nb, nnzb, bsrVal,
bsrRowPtr, bsrColInd, rowBlockDim, colBlockDim, bscVal, bscRowInd, bscColPtr,
copyValues, baseIdx, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: mb, nb, nnzb
  real(8), device :: bsrVal(*)
  integer(4), device :: bsrRowPtr(*), bsrColInd(*)
  integer(4) :: rowBlockDim, colBlockDim
  real(8), device :: bscVal(*)
  integer(4), device :: bscRowInd(*), bscColPtr(*)
  integer(4) :: copyValues, baseIdx
  character(c_char), device :: pBuffer(*)
```

5.8.66. cusparseCgebsr2gebsc

This function converts a sparse matrix in general block-CSR storage format to a sparse matrix in general block-CSC storage format.

```
integer(4) function cusparseCgebsr2gebsc(handle, mb, nb, nnzb, bsrVal,
bsrRowPtr, bsrColInd, rowBlockDim, colBlockDim, bscVal, bscRowInd, bscColPtr,
copyValues, baseIdx, pBuffer)
```

```
  type(cusparseHandle) :: handle
  integer(4) :: mb, nb, nnzb
  complex(4), device :: bsrVal(*)
  integer(4), device :: bsrRowPtr(*), bsrColInd(*)
  integer(4) :: rowBlockDim, colBlockDim
  complex(4), device :: bscVal(*)
  integer(4), device :: bscRowInd(*), bscColPtr(*)
  integer(4) :: copyValues, baseIdx
  character(c_char), device :: pBuffer(*)
```

5.8.67. cusparseZgebsr2gebsc

This function converts a sparse matrix in general block-CSR storage format to a sparse matrix in general block-CSC storage format.

```
integer(4) function cusparseZgebsr2gebsc(handle, mb, nb, nnzb, bsrVal,
bsrRowPtr, bsrColInd, rowBlockDim, colBlockDim, bscVal, bscRowInd, bscColPtr,
copyValues, baseIdx, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: mb, nb, nnzb
  complex(8), device :: bsrVal(*)
  integer(4), device :: bsrRowPtr(*), bsrColInd(*)
  integer(4) :: rowBlockDim, colBlockDim
  complex(8), device :: bscVal(*)
  integer(4), device :: bscRowInd(*), bscColPtr(*)
  integer(4) :: copyValues, baseIdx
  character(c_char), device :: pBuffer(*)
```

5.8.68. cusparseSgebsr2gebsr_bufferSize

This function returns the size of the buffer used in gebsr2gebsrnnz and gebsr2gebsr.

```
integer(4) function cusparseSgebsr2gebsr_bufferSize(handle, mb, nb, nnzb,
bsrVal, bsrRowPtr, bsrColInd, rowBlockDimA, colBlockDimA, rowBlockDimC,
colBlockDimC, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: mb, nb, nnzb
  real(4), device :: bsrVal(*)
  integer(4), device :: bsrRowPtr(*), bsrColInd(*)
  integer(4) :: rowBlockDimA, colBlockDimA, rowBlockDimC, colBlockDimC
  integer(4) :: pBufferSize ! integer(8) also accepted
```

5.8.69. cusparseDgebsr2gebsr_bufferSize

This function returns the size of the buffer used in gebsr2gebsrnnz and gebsr2gebsr.

```
integer(4) function cusparseDgebsr2gebsr_bufferSize(handle, mb, nb, nnzb,
bsrVal, bsrRowPtr, bsrColInd, rowBlockDimA, colBlockDimA, rowBlockDimC,
colBlockDimC, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: mb, nb, nnzb
  real(8), device :: bsrVal(*)
  integer(4), device :: bsrRowPtr(*), bsrColInd(*)
  integer(4) :: rowBlockDimA, colBlockDimA, rowBlockDimC, colBlockDimC
  integer(4) :: pBufferSize ! integer(8) also accepted
```

5.8.70. cusparseCgebsr2gebsr_bufferSize

This function returns the size of the buffer used in gebsr2gebsrnnz and gebsr2gebsr.

```
integer(4) function cusparseCgebsr2gebsr_bufferSize(handle, mb, nb, nnzb,
bsrVal, bsrRowPtr, bsrColInd, rowBlockDimA, colBlockDimA, rowBlockDimC,
colBlockDimC, pBufferSize)
  type(cusparseHandle) :: handle
```

```
  integer(4) :: mb, nb, nnzb
  complex(4), device :: bsrVal(*)
  integer(4), device :: bsrRowPtr(*), bsrColInd(*)
  integer(4) :: rowBlockDimA, colBlockDimA, rowBlockDimC, colBlockDimC
  integer(4) :: pBufferSize ! integer(8) also accepted
```

5.8.71. cusparseZgebsr2gebsr_bufferSize

This function returns the size of the buffer used in gebsr2gebsrnnz and gebsr2gebsr.

```
integer(4) function cusparseZgebsr2gebsr_bufferSize(handle, mb, nb, nnzb,
bsrVal, bsrRowPtr, bsrColInd, rowBlockDimA, colBlockDimA, rowBlockDimC,
colBlockDimC, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: mb, nb, nnzb
  complex(8), device :: bsrVal(*)
  integer(4), device :: bsrRowPtr(*), bsrColInd(*)
  integer(4) :: rowBlockDimA, colBlockDimA, rowBlockDimC, colBlockDimC
  integer(4) :: pBufferSize ! integer(8) also accepted
```

5.8.72. cusparseXgebsr2gebsrNnz

cusparseXcsrgeamNnz computes the number of nonzero elements which will be produced by CSRGEAM.

```
integer(4) function cusparseXgebsr2gebsrNnz(handle, dir, mb, nb, nnzb, descrA,
bsrRowPtrA, bsrColIndA, rowBlockDimA, colBlockDimA, descrC, bsrRowPtrC,
rowBlockDimC, colBlockDimC, nnzTotalDevHostPtr, pBuffer)
  type(cusparseHandle) :: handle
  integer :: dir, mb, nb, nnzb
  type(cusparseMatDescr) :: descrA
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer :: rowBlockDimA, colBlockDimA
  type(cusparseMatDescr) :: descrC
  integer(4), device :: bsrRowPtrC(*)
  integer :: rowBlockDimC, colBlockDimC
  integer(4), device :: nnzTotalDevHostPtr ! device or host variable
  character, device :: pBuffer(*)
```

5.8.73. cusparseSgebsr2gebsr

This function converts a sparse matrix in general BSR storage format that is defined by the three arrays bsrValA, bsrRowPtrA, and bsrColIndA into a sparse matrix in another general BSR format that is defined by arrays bsrValC, bsrRowPtrC, and bsrColIndC.

```
integer(4) function cusparseSgebsr2gebsr(handle, dir, mb, nb, nnzb, descrA,
bsrValA, bsrRowPtrA, bsrColIndA, rowBlockDimA, colBlockDimA, descrC, bsrValC,
bsrRowPtrC, bsrColIndC, rowBlockDimC, colBlockDimC, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dir, mb, nb, nnzb
  type(cusparseMatDescr) :: descrA
  real(4), device :: bsrValA(*), bsrValC(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: rowBlockDimA, colBlockDimA
  type(cusparseMatDescr) :: descrC
  integer(4), device :: bsrRowPtrC(*), bsrColIndC(*)
  integer(4) :: rowBlockDimC, colBlockDimC
  character(c_char), device :: pBuffer(*)
```

5.8.74. cusparseDgebsr2gebsr

This function converts a sparse matrix in general BSR storage format that is defined by the three arrays bsrValA, bsrRowPtrA, and bsrColIndA into a sparse matrix in another general BSR format that is defined by arrays bsrValC, bsrRowPtrC, and bsrColIndC.

```
integer(4) function cusparseDgebsr2gebsr(handle, dir, mb, nb, nnzb, descrA,
bsrValA, bsrRowPtrA, bsrColIndA, rowBlockDimA, colBlockDimA, descrC, bsrValC,
bsrRowPtrC, bsrColIndC, rowBlockDimC, colBlockDimC, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dir, mb, nb, nnzb
  type(cusparseMatDescr) :: descrA
  real(8), device :: bsrValA(*), bsrValC(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: rowBlockDimA, colBlockDimA
  type(cusparseMatDescr) :: descrC
  integer(4), device :: bsrRowPtrC(*), bsrColIndC(*)
  integer(4) :: rowBlockDimC, colBlockDimC
  character(c_char), device :: pBuffer(*)
```

5.8.75. cusparseCgebsr2gebsr

This function converts a sparse matrix in general BSR storage format that is defined by the three arrays bsrValA, bsrRowPtrA, and bsrColIndA into a sparse matrix in another general BSR format that is defined by arrays bsrValC, bsrRowPtrC, and bsrColIndC.

```
integer(4) function cusparseCgebsr2gebsr(handle, dir, mb, nb, nnzb, descrA,
bsrValA, bsrRowPtrA, bsrColIndA, rowBlockDimA, colBlockDimA, descrC, bsrValC,
bsrRowPtrC, bsrColIndC, rowBlockDimC, colBlockDimC, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dir, mb, nb, nnzb
  type(cusparseMatDescr) :: descrA
  complex(4), device :: bsrValA(*), bsrValC(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: rowBlockDimA, colBlockDimA
  type(cusparseMatDescr) :: descrC
  integer(4), device :: bsrRowPtrC(*), bsrColIndC(*)
  integer(4) :: rowBlockDimC, colBlockDimC
  character(c_char), device :: pBuffer(*)
```

5.8.76. cusparseZgebsr2gebsr

This function converts a sparse matrix in general BSR storage format that is defined by the three arrays bsrValA, bsrRowPtrA, and bsrColIndA into a sparse matrix in another general BSR format that is defined by arrays bsrValC, bsrRowPtrC, and bsrColIndC.

```
integer(4) function cusparseZgebsr2gebsr(handle, dir, mb, nb, nnzb, descrA,
bsrValA, bsrRowPtrA, bsrColIndA, rowBlockDimA, colBlockDimA, descrC, bsrValC,
bsrRowPtrC, bsrColIndC, rowBlockDimC, colBlockDimC, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dir, mb, nb, nnzb
  type(cusparseMatDescr) :: descrA
  complex(8), device :: bsrValA(*), bsrValC(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: rowBlockDimA, colBlockDimA
  type(cusparseMatDescr) :: descrC
  integer(4), device :: bsrRowPtrC(*), bsrColIndC(*)
  integer(4) :: rowBlockDimC, colBlockDimC
  character(c_char), device :: pBuffer(*)
```

5.8.77. cusparseSgebsr2csr

This function converts a sparse matrix in general BSR storage format that is defined by the three arrays bsrValA, bsrRowPtrA, and bsrColIndA into a sparse matrix in CSR format that is defined by arrays csrValC, csrRowPtrC, and csrColIndC.

```
integer(4) function cusparseSgebsr2csr(handle, dir, mb, nb, descrA, bsrValA,
bsrRowPtrA, bsrColIndA, rowBlockDimA, colBlockDimA, descrC, csrValC,
csrRowPtrC, csrColIndC)
  type(cusparseHandle) :: handle
  integer(4) :: dir, mb, nb, nnzb
  type(cusparseMatDescr) :: descrA
  real(4), device :: bsrValA(*), csrValC(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: rowBlockDimA, colBlockDimA
  type(cusparseMatDescr) :: descrC
  integer(4), device :: csrRowPtrC(*), csrColIndC(*)
```

5.8.78. cusparseDgebsr2csr

This function converts a sparse matrix in general BSR storage format that is defined by the three arrays bsrValA, bsrRowPtrA, and bsrColIndA into a sparse matrix in CSR format that is defined by arrays csrValC, csrRowPtrC, and csrColIndC.

```
integer(4) function cusparseDgebsr2csr(handle, dir, mb, nb, descrA, bsrValA,
bsrRowPtrA, bsrColIndA, rowBlockDimA, colBlockDimA, descrC, csrValC,
csrRowPtrC, csrColIndC)
  type(cusparseHandle) :: handle
  integer(4) :: dir, mb, nb, nnzb
  type(cusparseMatDescr) :: descrA
  real(8), device :: bsrValA(*), csrValC(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: rowBlockDimA, colBlockDimA
  type(cusparseMatDescr) :: descrC
  integer(4), device :: csrRowPtrC(*), csrColIndC(*)
```

5.8.79. cusparseCgebsr2csr

This function converts a sparse matrix in general BSR storage format that is defined by the three arrays bsrValA, bsrRowPtrA, and bsrColIndA into a sparse matrix in CSR format that is defined by arrays csrValC, csrRowPtrC, and csrColIndC.

```
integer(4) function cusparseCgebsr2csr(handle, dir, mb, nb, descrA, bsrValA,
bsrRowPtrA, bsrColIndA, rowBlockDimA, colBlockDimA, descrC, csrValC,
csrRowPtrC, csrColIndC)
  type(cusparseHandle) :: handle
  integer(4) :: dir, mb, nb, nnzb
  type(cusparseMatDescr) :: descrA
  complex(4), device :: bsrValA(*), csrValC(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: rowBlockDimA, colBlockDimA
  type(cusparseMatDescr) :: descrC
  integer(4), device :: csrRowPtrC(*), csrColIndC(*)
```

5.8.80. cusparseZgebsr2csr

This function converts a sparse matrix in general BSR storage format that is defined by the three arrays bsrValA, bsrRowPtrA, and bsrColIndA into a sparse matrix in CSR format that is defined by arrays csrValC, csrRowPtrC, and csrColIndC.

```
integer(4) function cusparseZgebsr2csr(handle, dir, mb, nb, descrA, bsrValA,
bsrRowPtrA, bsrColIndA, rowBlockDimA, colBlockDimA, descrC, csrValC,
csrRowPtrC, csrColIndC)
  type(cusparseHandle) :: handle
  integer(4) :: dir, mb, nb, nnzb
  type(cusparseMatDescr) :: descrA
  complex(8), device :: bsrValA(*), csrValC(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: rowBlockDimA, colBlockDimA
  type(cusparseMatDescr) :: descrC
  integer(4), device :: csrRowPtrC(*), csrColIndC(*)
```

5.8.81. cusparseScsr2gebsr_bufferSize

This function returns the size of the buffer used in csr2gebsrnnz and csr2gebsr.

```
integer(4) function cusparseScsr2gebsr_bufferSize(handle, dir, m, n, descrA,
csrVal, csrRowPtr, csrColInd, rowBlockDim, colBlockDim, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: dir, m, n
  real(4), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*), csrColInd(*)
  integer(4) :: rowBlockDim, colBlockDim
  integer(4) :: pBufferSize ! integer(8) also accepted
```

5.8.82. cusparseDcsr2gebsr_bufferSize

This function returns the size of the buffer used in csr2gebsrnnz and csr2gebsr.

```
integer(4) function cusparseDcsr2gebsr_bufferSize(handle, dir, m, n, descrA,
csrVal, csrRowPtr, csrColInd, rowBlockDim, colBlockDim, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: dir, m, n
  real(8), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*), csrColInd(*)
  integer(4) :: rowBlockDim, colBlockDim
  integer(4) :: pBufferSize ! integer(8) also accepted
```

5.8.83. cusparseCcsr2gebsr_bufferSize

This function returns the size of the buffer used in csr2gebsrnnz and csr2gebsr.

```
integer(4) function cusparseCcsr2gebsr_bufferSize(handle, dir, m, n, descrA,
csrVal, csrRowPtr, csrColInd, rowBlockDim, colBlockDim, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: dir, m, n
  complex(4), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*), csrColInd(*)
  integer(4) :: rowBlockDim, colBlockDim
  integer(4) :: pBufferSize ! integer(8) also accepted
```

5.8.84. cusparseZcsr2gebsr_bufferSize

This function returns the size of the buffer used in csr2gebsrnnz and csr2gebsr.

```
integer(4) function cusparseZcsr2gebsr_bufferSize(handle, dir, m, n, descrA,
csrVal, csrRowPtr, csrColInd, rowBlockDim, colBlockDim, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: dir, m, n
  complex(8), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*), csrColInd(*)
  integer(4) :: rowBlockDim, colBlockDim
  integer(4) :: pBufferSize ! integer(8) also accepted
```

5.8.85. cusparseXcsr2gebsrNnz

cusparseXcsrgeamNnz computes the number of nonzero elements which will be produced by CSRGEAM.

```
integer(4) function cusparseXcsr2gebsrNnz(handle, dir, m, n, descrA,
csrRowPtrA, csrColIndA, descrC, bsrRowPtrC, rowBlockDimC, colBlockDimC,
nnzTotalDevHostPtr, pBuffer)
  type(cusparseHandle) :: handle
  integer :: dir, m, n
  type(cusparseMatDescr) :: descrA
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseMatDescr) :: descrC
  integer(4), device :: bsrRowPtrC(*)
  integer :: rowBlockDimC, colBlockDimC
  integer(4), device :: nnzTotalDevHostPtr ! device or host variable
  character, device :: pBuffer(*)
```

5.8.86. cusparseScsr2gebsr

This function converts a sparse matrix in CSR storage format that is defined by the three arrays csrValA, csrRowPtrA, and csrColIndA into a sparse matrix in general BSR format that is defined by arrays bsrValC, bsrRowPtrC, and bsrColIndC.

```
integer(4) function cusparseScsr2gebsr(handle, dir, m, n, descrA, csrValA,
csrRowPtrA, csrColIndA, descrC, bsrValC, bsrRowPtrC, bsrColIndC, rowBlockDimC,
colBlockDimC, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dir, mb, nb, nnzb
  type(cusparseMatDescr) :: descrA
  real(4), device :: csrValA(*), bsrValC(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseMatDescr) :: descrC
  integer(4), device :: bsrRowPtrC(*), bsrColIndC(*)
  integer(4) :: rowBlockDimC, colBlockDimC
  character(c_char), device :: pBuffer(*)
```

5.8.87. cusparseDcsr2gebsr

This function converts a sparse matrix in CSR storage format that is defined by the three arrays csrValA, csrRowPtrA, and csrColIndA into a sparse matrix in general BSR format that is defined by arrays bsrValC, bsrRowPtrC, and bsrColIndC.

```
integer(4) function cusparseDcsr2gebsr(handle, dir, m, n, descrA, csrValA,
csrRowPtrA, csrColIndA, descrC, bsrValC, bsrRowPtrC, bsrColIndC, rowBlockDimC,
colBlockDimC, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dir, mb, nb, nnzb
```

```
  type(cusparseMatDescr) :: descrA
  real(8), device :: csrValA(*), bsrValC(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseMatDescr) :: descrC
  integer(4), device :: bsrRowPtrC(*), bsrColIndC(*)
  integer(4) :: rowBlockDimC, colBlockDimC
  character(c_char), device :: pBuffer(*)
```

5.8.88. cusparseCcsr2gebsr

This function converts a sparse matrix in CSR storage format that is defined by the three arrays csrValA, csrRowPtrA, and csrColIndA into a sparse matrix in general BSR format that is defined by arrays bsrValC, bsrRowPtrC, and bsrColIndC.

```
integer(4) function cusparseCcsr2gebsr(handle, dir, m, n, descrA, csrValA,
csrRowPtrA, csrColIndA, descrC, bsrValC, bsrRowPtrC, bsrColIndC, rowBlockDimC,
colBlockDimC, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dir, mb, nb, nnzb
  type(cusparseMatDescr) :: descrA
  complex(4), device :: csrValA(*), bsrValC(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseMatDescr) :: descrC
  integer(4), device :: bsrRowPtrC(*), bsrColIndC(*)
  integer(4) :: rowBlockDimC, colBlockDimC
  character(c_char), device :: pBuffer(*)
```

5.8.89. cusparseZcsr2gebsr

This function converts a sparse matrix in CSR storage format that is defined by the three arrays csrValA, csrRowPtrA, and csrColIndA into a sparse matrix in general BSR format that is defined by arrays bsrValC, bsrRowPtrC, and bsrColIndC.

```
integer(4) function cusparseZcsr2gebsr(handle, dir, m, n, descrA, csrValA,
csrRowPtrA, csrColIndA, descrC, bsrValC, bsrRowPtrC, bsrColIndC, rowBlockDimC,
colBlockDimC, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dir, mb, nb, nnzb
  type(cusparseMatDescr) :: descrA
  complex(8), device :: csrValA(*), bsrValC(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseMatDescr) :: descrC
  integer(4), device :: bsrRowPtrC(*), bsrColIndC(*)
  integer(4) :: rowBlockDimC, colBlockDimC
  character(c_char), device :: pBuffer(*)
```

5.8.90. cusparseCreateIdentityPermutation

This function creates an identity map. The output parameter p represents such map by p = 0:1:(n-1). This function is typically used with coosort, csrsort, cscsort, and csr2csc_indexOnly.

```
integer(4) function cusparseCreateIdentityPermutation(handle, n, p)
  type(cusparseHandle) :: handle
  integer(4) :: n
  integer(4), device :: p(*)
```

5.8.91. cusparseXcoosort_bufferSize

This function returns the size of the buffer used in coosort.

```
integer(4) function cusparseXcoosort_bufferSize(handle, m, n, nnz, cooRows,
cooCols, pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, nnz
  integer(4), device :: cooRows(*), cooCols(*)
  integer(8) :: pBufferSizeInBytes
```

5.8.92. cusparseXcoosortByRow

This function sorts the sparse matrix stored in COO format.

```
integer(4) function cusparseXcoosortByRow(handle, m, n, nnz, cooRows, cooCols,
P, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, nnz
  integer(4), device :: cooRows(*), cooCols(*), P(*)
  character(c_char), device :: pBuffer(*)
```

5.8.93. cusparseXcoosortByColumn

This function sorts the sparse matrix stored in COO format.

```
integer(4) function cusparseXcoosortByColumn(handle, m, n, nnz, cooRows,
cooCols, P, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, nnz
  integer(4), device :: cooRows(*), cooCols(*), P(*)
  character(c_char), device :: pBuffer(*)
```

5.8.94. cusparseXcsrsort_bufferSize

This function returns the size of the buffer used in csrsort.

```
integer(4) function cusparseXcsrsort_bufferSize(handle, m, n, nnz, csrRowInd,
csrColInd, pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, nnz
  integer(4), device :: csrRowInd(*), csrColInd(*)
  integer(8) :: pBufferSizeInBytes
```

5.8.95. cusparseXcsrsort

This function sorts the sparse matrix stored in CSR format.

```
integer(4) function cusparseXcsrsort(handle, m, n, nnz, csrRowInd, csrColInd,
P, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, nnz
  integer(4), device :: csrRowInd(*), csrColInd(*), P(*)
  character(c_char), device :: pBuffer(*)
```

5.8.96. cusparseXcscsort_bufferSize

This function returns the size of the buffer used in cscsort.

```
integer(4) function cusparseXcscsort_bufferSize(handle, m, n, nnz, cscColPtr,
cscRowInd, pBufferSizeInBytes)
```

```
  type(cusparseHandle) :: handle
  integer(4) :: m, n, nnz
  integer(4), device :: cscColPtr(*), cscRowInd(*)
  integer(8) :: pBufferSizeInBytes
```

5.8.97. cusparseXcscsort

This function sorts the sparse matrix stored in CSC format.

```
integer(4) function cusparseXcscsort(handle, m, n, nnz, cscColPtr, cscRowInd,
P, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, nnz
  integer(4), device :: cscColPtr(*), cscRowInd(*), P(*)
  character(c_char), device :: pBuffer(*)
```

5.8.98. cusparseScsru2csr_bufferSize

This function returns the size of the buffer used in csru2csr.

```
integer(4) function cusparseScsru2csr_bufferSize(handle, m, n, nnz, csrVal,
csrRowPtr, csrColInd, info, pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, nnz
  real(4), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*), csrColInd(*)
  type(cusparseCsru2csrInfo) :: info
  integer(8) :: pBufferSizeInBytes
```

5.8.99. cusparseDcsru2csr_bufferSize

This function returns the size of the buffer used in csru2csr.

```
integer(4) function cusparseDcsru2csr_bufferSize(handle, m, n, nnz, csrVal,
csrRowPtr, csrColInd, info, pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, nnz
  real(8), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*), csrColInd(*)
  type(cusparseCsru2csrInfo) :: info
  integer(8) :: pBufferSizeInBytes
```

5.8.100. cusparseCcsru2csr_bufferSize

This function returns the size of the buffer used in csru2csr.

```
integer(4) function cusparseCcsru2csr_bufferSize(handle, m, n, nnz, csrVal,
csrRowPtr, csrColInd, info, pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, nnz
  complex(4), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*), csrColInd(*)
  type(cusparseCsru2csrInfo) :: info
  integer(8) :: pBufferSizeInBytes
```

5.8.101. cusparseZcsru2csr_bufferSize

This function returns the size of the buffer used in csru2csr.

```
integer(4) function cusparseZcsru2csr_bufferSize(handle, m, n, nnz, csrVal,
csrRowPtr, csrColInd, info, pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, nnz
```

```
  complex(8), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*), csrColInd(*)
  type(cusparseCsru2csrInfo) :: info
  integer(8) :: pBufferSizeInBytes
```

5.8.102. cusparseScsru2csr

This function transfers unsorted CSR format to CSR format.

```
integer(4) function cusparseScsru2csr(handle, m, n, nnz, descrA, csrVal,
csrRowPtr, csrColInd, info, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, nnz
  type(cusparseMatDescr) :: descrA
  real(4), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*), csrColInd(*)
  type(cusparseCsru2csrInfo) :: info
  character(c_char), device :: pBuffer(*)
```

5.8.103. cusparseDcsru2csr

This function transfers unsorted CSR format to CSR format.

```
integer(4) function cusparseDcsru2csr(handle, m, n, nnz, descrA, csrVal,
csrRowPtr, csrColInd, info, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, nnz
  type(cusparseMatDescr) :: descrA
  real(8), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*), csrColInd(*)
  type(cusparseCsru2csrInfo) :: info
  character(c_char), device :: pBuffer(*)
```

5.8.104. cusparseCcsru2csr

This function transfers unsorted CSR format to CSR format.

```
integer(4) function cusparseCcsru2csr(handle, m, n, nnz, descrA, csrVal,
csrRowPtr, csrColInd, info, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, nnz
  type(cusparseMatDescr) :: descrA
  complex(4), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*), csrColInd(*)
  type(cusparseCsru2csrInfo) :: info
  character(c_char), device :: pBuffer(*)
```

5.8.105. cusparseZcsru2csr

This function transfers unsorted CSR format to CSR format.

```
integer(4) function cusparseZcsru2csr(handle, m, n, nnz, descrA, csrVal,
csrRowPtr, csrColInd, info, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, nnz
  type(cusparseMatDescr) :: descrA
  complex(8), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*), csrColInd(*)
  type(cusparseCsru2csrInfo) :: info
  character(c_char), device :: pBuffer(*)
```

5.8.106. cusparseScsr2csru

This function performs the backwards transformation from sorted CSR format to unsorted CSR format.

```
integer(4) function cusparseScsr2csru(handle, m, n, nnz, descrA, csrVal,
csrRowPtr, csrColInd, info, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, nnz
  type(cusparseMatDescr) :: descrA
  real(4), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*), csrColInd(*)
  type(cusparseCsru2csrInfo) :: info
  character(c_char), device :: pBuffer(*)
```

5.8.107. cusparseDcsr2csru

This function performs the backwards transformation from sorted CSR format to unsorted CSR format.

```
integer(4) function cusparseDcsr2csru(handle, m, n, nnz, descrA, csrVal,
csrRowPtr, csrColInd, info, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, nnz
  type(cusparseMatDescr) :: descrA
  real(8), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*), csrColInd(*)
  type(cusparseCsru2csrInfo) :: info
  character(c_char), device :: pBuffer(*)
```

5.8.108. cusparseCcsr2csru

This function performs the backwards transformation from sorted CSR format to unsorted CSR format.

```
integer(4) function cusparseCcsr2csru(handle, m, n, nnz, descrA, csrVal,
csrRowPtr, csrColInd, info, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, nnz
  type(cusparseMatDescr) :: descrA
  complex(4), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*), csrColInd(*)
  type(cusparseCsru2csrInfo) :: info
  character(c_char), device :: pBuffer(*)
```

5.8.109. cusparseZcsr2csru

This function performs the backwards transformation from sorted CSR format to unsorted CSR format.

```
integer(4) function cusparseZcsr2csru(handle, m, n, nnz, descrA, csrVal,
csrRowPtr, csrColInd, info, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, nnz
  type(cusparseMatDescr) :: descrA
  complex(8), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*), csrColInd(*)
  type(cusparseCsru2csrInfo) :: info
  character(c_char), device :: pBuffer(*)
```

5.9. CUSPARSE Generic API Functions

This section contains interfaces for the generic API functions that perform vector-vector (SpVV), matrix-vector (SpMV), and matrix-matrix (SpMM) operations.

5.9.1. cusparseCreateSpVec

This function initializes the sparse vector descriptor used in the generic API. The type, kind, and rank of the input arguments "indices" and "values" are actually ignored, and taken from the input arguments "idxType" and "valueType". The vectors are assumed to be contiguous. The "idxBase" argument is typically CUSPARSE_INDEX_BASE_ONE in Fortran.

```
integer(4) function cusparseCreateSpVec(descr, size, nnz, indices, values,
idxType, idxBase, valueType)
  type(cusparseSpVecDescr) :: descr
  integer(8) :: size, nnz
  integer(4), device :: indices(*)
  real(4), device    :: values(*)
  integer(4) :: idxType, idxBase, valueType
```

5.9.2. cusparseDestroySpVec

This function releases the host memory associated with the sparse vector descriptor used in the generic API.

```
integer(4) function cusparseDestroySpVec(descr)
  type(cusparseSpVecDescr) :: descr
```

5.9.3. cusparseSpVecGet

This function returns the fields within the sparse vector descriptor used in the generic API.

```
integer(4) function cusparseSpVecGet(descr, size, nnz, indices, values,
idxType, idxBase, valueType)
  type(cusparseSpVecDescr) :: descr
  integer(8) :: size, nnz
  type(c_devptr) :: indices
  type(c_devptr) :: values
  integer(4) :: idxType, idxBase, valueType
```

5.9.4. cusparseSpVecGetIndexBase

This function returns "idxBase" field within the sparse vector descriptor used in the generic API.

```
integer(4) function cusparseSpVecGetIndexBase(descr, idxBase)
  type(cusparseSpVecDescr) :: descr
  integer(4) :: idxBase
```

5.9.5. cusparseSpVecGetValues

This function returns the "values" field within the sparse vector descriptor used in the generic API.

```
integer(4) function cusparseSpVecGetValues(descr, values)
  type(cusparseSpVecDescr) :: descr
  type(c_devptr) :: values
```

5.9.6. cusparseSpVecSetValues

This function sets the "values" field within the sparse vector descriptor used in the generic API. The type, kind and rank of the "values" argument is ignored; the type is determined by the valueType field in the descriptor.

```
integer(4) function cusparseSpVecSetValues(descr, values)
  type(cusparseSpVecDescr) :: descr
  real(4), device :: values(*)
```

5.9.7. cusparseCreateDnVec

This function initializes the dense vector descriptor used in the generic API. The type, kind, and rank of the "values" input argument is ignored, and taken from the input argument "valueType". The vector is assumed to be contiguous.

```
integer(4) function cusparseCreateDnVec(descr, size, values, valueType)
  type(cusparseDnVecDescr) :: descr
  integer(8) :: size
  real(4), device    :: values(*)
  integer(4) :: valueType
```

5.9.8. cusparseDestroyDnVec

This function releases the host memory associated with the dense vector descriptor used in the generic API.

```
integer(4) function cusparseDestroyDnVec(descr)
  type(cusparseDnVecDescr) :: descr
```

5.9.9. cusparseDnVecGet

This function returns the fields within the dense vector descriptor used in the generic API.

```
integer(4) function cusparseDnVecGet(descr, size, values, valueType)
  type(cusparseDnVecDescr) :: descr
  integer(8) :: size
  type(c_devptr) :: values
  integer(4) :: valueType
```

5.9.10. cusparseDnVecGetValues

This function returns the "values" field within the dense vector descriptor used in the generic API.

```
integer(4) function cusparseDnVecGetValues(descr, values)
  type(cusparseDnVecDescr) :: descr
  type(c_devptr) :: values
```

5.9.11. cusparseDnVecSetValues

This function sets the "values" field within the dense vector descriptor used in the generic API. The type, kind and rank of the "values" argument is ignored; the type is determined by the valueType field in the descriptor.

```
integer(4) function cusparseDnVecSetValues(descr, values)
  type(cusparseDnVecDescr) :: descr
  real(4), device :: values(*)
```

5.9.12. cusparseCreateCoo

This function initializes the sparse matrix descriptor in COO format used in the generic API. The type, kind, and rank of the input arguments "cooRowInd", "cooColInd", and "cooValues" are actually ignored, and taken from the input arguments "idxType" and "valueType". The "idxBase" argument is typically CUSPARSE_INDEX_BASE_ONE in Fortran.

```
integer(4) function cusparseCreateCoo(descr, rows, cols, nnz, cooRowInd,
cooColInd, cooValues, idxType, idxBase, valueType)
  type(cusparseSpMatDescr) :: descr
  integer(8) :: rows, cols, nnz
  integer(4), device :: cooRowInd(*), cooColInd(*)
  real(4), device    :: cooValues(*)
  integer(4) :: idxType, idxBase, valueType
```

5.9.13. cusparseCreateCooAoS

This function initializes the sparse matrix descriptor in COO format, with Array of Structures layout, used in the generic API. The type, kind, and rank of the input arguments "cooInd" and "cooValues" are actually ignored, and taken from the input arguments "idxType" and "valueType". The "idxBase" argument is typically CUSPARSE_INDEX_BASE_ONE in Fortran.

```
integer(4) function cusparseCreateCooAoS(descr, rows, cols, nnz, cooInd,
cooValues, idxType, idxBase, valueType)
  type(cusparseSpMatDescr) :: descr
  integer(8) :: rows, cols, nnz
  integer(4), device :: cooInd(*)
  real(4), device    :: cooValues(*)
  integer(4) :: idxType, idxBase, valueType
```

5.9.14. cusparseCreateCsr

This function initializes the sparse matrix descriptor in CSR format used in the generic API. The type, kind, and rank of the input arguments "csrRowOffsets", "csrColInd", and "csrValues" are actually ignored, and taken from the input arguments "csrRowOffsetsType", "csrColIndType", and "valueType". The "idxBase" argument is typically CUSPARSE_INDEX_BASE_ONE in Fortran.

```
integer(4) function cusparseCreateCsr(descr, rows, cols, nnz, csrRowOffsets,
csrColInd, csrValues, csrRowOffsetsType, csrColIndType, idxBase, valueType)
  type(cusparseSpMatDescr) :: descr
  integer(8) :: rows, cols, nnz
  integer(4), device :: csrRowOffsets(*), csrColInd(*)
  real(4), device    :: csrValues(*)
  integer(4) :: csrRowOffsetsType, csrColIndType, idxBase, valueType
```

5.9.15. cusparseDestroySpMat

This function releases the host memory associated with the sparse matrix descriptor used in the generic API.

```
integer(4) function cusparseDestroySpMat(descr)
  type(cusparseSpMatDescr) :: descr
```

5.9.16. cusparseCooGet

This function returns the fields from the sparse matrix descriptor in COO format used in the generic API.

```
integer(4) function cusparseCooGet(descr, rows, cols, nnz, cooRowInd,
cooColInd, cooValues, idxType, idxBase, valueType)
  type(cusparseSpMatDescr) :: descr
  integer(8) :: rows, cols, nnz
  type(c_devptr) :: cooRowInd, cooColInd, cooValues
  integer(4) :: idxType, idxBase, valueType
```

5.9.17. cusparseCooAoSGet

This function returns the fields from the sparse matrix descriptor in COO format, Array of Structures layout, used in the generic API.

```
integer(4) function cusparseCooAoSGet(descr, rows, cols, nnz, cooInd,
cooValues, idxType, idxBase, valueType)
  type(cusparseSpMatDescr) :: descr
  integer(8) :: rows, cols, nnz
  type(c_devptr) :: cooInd, cooValues
  integer(4) :: idxType, idxBase, valueType
```

5.9.18. cusparseCsrGet

This function returns the fields from the sparse matrix descriptor in CSR format used in the generic API.

```
integer(4) function cusparseCsrGet(descr, rows, cols, nnz, csrRowOffsets,
csrColInd, csrValues, csrRowOffsetsType, crsColIndType, idxBase, valueType)
  type(cusparseSpMatDescr) :: descr
  integer(8) :: rows, cols, nnz
  type(c_devptr) :: csrRowOffsets, csrColInd, csrValues
  integer(4) :: csrRowOffsetsType, csrColIndType, idxBase, valueType
```

5.9.19. cusparseSpMatGetFormat

This function returns the "format" field within the sparse matrix descriptor used in the generic API. Valid formats for the generic API are CUSPARSE_FORMAT_CSR, CUSPARSE_FORMAT_COO, and CUSPARSE_FORMAT_COO_AOS.

```
integer(4) function cusparseSpMatGetFormat(descr, format)
  type(cusparseSpMatDescr) :: descr
  integer(4) :: format
```

5.9.20. cusparseSpMatGetIndexBase

This function returns "idxBase" field within the sparse matrix descriptor used in the generic API.

```
integer(4) function cusparseSpMatGetIndexBase(descr, idxBase)
  type(cusparseSpMatDescr) :: descr
  integer(4) :: idxBase
```

5.9.21. cusparseSpMatGetValues

This function returns the "values" field within the sparse matrix descriptor used in the generic API.

```
integer(4) function cusparseSpMatGetValues(descr, values)
  type(cusparseSpMatDescr) :: descr
  type(c_devptr) :: values
```

5.9.22. cusparseSpMatSetValues

This function sets the "values" field within the sparse matrix descriptor used in the generic API. The type, kind and rank of the "values" argument is ignored; the type is determined by the valueType field in the descriptor.

```
integer(4) function cusparseSpMatSetValues(descr, values)
  type(cusparseSpMatDescr) :: descr
  real(4), device :: values(*)
```

5.9.23. cusparseSpMatGetStridedBatch

This function returns the "batchCount" field within the sparse matrix descriptor used in the generic API.

```
integer(4) function cusparseSpMatGetStridedBatch(descr, batchCount)
  type(cusparseSpMatDescr) :: descr
  integer(4) :: batchCount
```

5.9.24. cusparseSpMatSetStridedBatch

This function sets the "batchCount" field within the sparse matrix descriptor used in the generic API.

```
integer(4) function cusparseSpMatSetStridedBatch(descr, batchCount)
  type(cusparseSpMatDescr) :: descr
  integer(4) :: batchCount
```

5.9.25. cusparseCreateDnMat

This function initializes the dense matrix descriptor used in the generic API. The type, kind, and rank of the "values" input argument is ignored, and taken from the input argument "valueType". The "order" argument in Fortran should normally be CUSPARSE_ORDER_COL.

```
integer(4) function cusparseCreateDnMat(descr, rows, cols, ld, values,
valueType, order)
  type(cusparseDnMatDescr) :: descr
  integer(8) :: rows, cols, ld
  real(4), device    :: values(*)
```

```
  integer(4), value  :: valueType, order
```

5.9.26. cusparseDestroyDnMat

This function releases the host memory associated with the dense matrix descriptor used in the generic API.

```
integer(4) function cusparseDestroyDnMat(descr)
  type(cusparseDnMatDescr) :: descr
```

5.9.27. cusparseDnMatGet

This function returns the fields from the dense matrix descriptor used in the generic API.

```
integer(4) function cusparseDnMatGet(descr, rows, cols, ld, values, valueType,
order)
  type(cusparseDnMatDescr) :: descr
  integer(8) :: rows, cols, ld
  type(c_devptr) :: values
  integer(4) :: valueType, order
```

5.9.28. cusparseDnMatGetValues

This function returns the "values" field within the dense matrix descriptor used in the generic API.

```
integer(4) function cusparseDnMatGetValues(descr, values)
  type(cusparseDnMatDescr) :: descr
  type(c_devptr) :: values
```

5.9.29. cusparseDnMatSetValues

This function sets the "values" field within the dense matrix descriptor used in the generic API. The type, kind and rank of the "values" argument is ignored; the type is determined by the valueType field in the descriptor.

```
integer(4) function cusparseDnMatSetValues(descr, values)
  type(cusparseDnMatDescr) :: descr
  real(4), device :: values(*)
```

5.9.30. cusparseDnMatGetStridedBatch

This function returns the "batchCount" field within the dense matrix descriptor used in the generic API.

```
integer(4) function cusparseDnMatGetStridedBatch(descr, batchCount)
  type(cusparseDnMatDescr) :: descr
  integer(4) :: batchCount
```

5.9.31. cusparseDnMatSetStridedBatch

This function sets the "batchCount" field within the dense matrix descriptor used in the generic API.

```
integer(4) function cusparseDnMatSetStridedBatch(descr, batchCount)
  type(cusparseDnMatDescr) :: descr
  integer(4) :: batchCount
```

5.9.32. cusparseSpVV_bufferSize

This function returns the size of the workspace needed by cusparseSpVV(). The value returned is in bytes.

```
integer(4) function cusparseSpVV_bufferSize(handle, opX, vecX, vecY, result,
computeType, bufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: opX
  type(cusparseSpVecDescr) :: vecX
  type(cusparseDnVecDescr) :: vecY
  real(4), device    :: result ! device or host variable
  integer(4) :: computeType
  integer(8), intent(out) :: bufferSize
```

5.9.33. cusparseSpVV

This function forms the dot product of a sparse vector "vecX" and a dense vector "vecY". The "buffer" argument can be any type, but the size should be greater than or equal to the size returned from cusparseSpVV_buffersize(). See the CUSPARSE Library documentation for datatype and "computetype" combinations supported in each release.

```
integer(4) function cusparseSpVV(handle, opX, vecX, vecY, result, computeType,
buffer)
  type(cusparseHandle) :: handle
  integer(4) :: opX
  type(cusparseSpVecDescr) :: vecX
  type(cusparseDnVecDescr) :: vecY
  real(4), device    :: result ! device or host variable
  integer(4) :: computeType
  integer(4), device :: buffer(*)
```

5.9.34. cusparseSpMV_bufferSize

This function returns the size of the workspace needed by cusparseSpMV(). The value returned is in bytes.

```
integer(4) function cusparseSpMV_bufferSize(handle, opA, alpha, matA, vecX,
beta, vecY, computeType, alg, bufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: opA
  real(4) :: alpha, beta ! device or host variable
  type(cusparseSpMatDescr) :: matA
  type(cusparseDnVecDescr) :: vecX, vecY
  integer(4) :: computeType, alg
  integer(8), intent(out) :: bufferSize
```

5.9.35. cusparseSpMV

This function forms the multiplication of a sparse matrix "matA" and a dense vector "vecX" to produce dense vector "vecY". The "buffer" argument can be any type, but the size should be greater than or equal to the size returned from cusparseSpMV_buffersize(). The type of arguments "alpha" and "beta" should match the "computeType" argument. See the CUSPARSE Library documentation for the datatype, "computeType", sparse format, and "alg" combinations supported in each release.

```
integer(4) function cusparseSpMV(handle, opA, alpha, matA, vecX, beta, vecY,
computeType, alg, buffer)
  type(cusparseHandle) :: handle
```

```
  integer(4) :: opA
  real(4), device :: alpha, beta ! device or host variable
  type(cusparseSpMatDescr) :: matA
  type(cusparseDnVecDescr) :: vecX, vecY
  integer(4) :: computeType, alg
  integer(4), device :: buffer(*)
```

5.9.36. cusparseSpMM_bufferSize

This function returns the size of the workspace needed by cusparseSpMM(). The value returned is in bytes.

```
integer(4) function cusparseSpMM_bufferSize(handle, opA, opB, alpha, matA,
matB, beta, matC, computeType, alg, bufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: opA, opB
  real(4), device :: alpha, beta ! device or host variable
  type(cusparseSpMatDescr) :: matA
  type(cusparseDnMatDescr) :: matB, matC
  integer(4) :: computeType, alg
  integer(8), intent(out) :: bufferSize
```

5.9.37. cusparseSpMM

This function forms the multiplication of a sparse matrix "matA" and a dense matrix "matB" to produce dense matrix "matC". The "buffer" argument can be any type, but the size should be greater than or equal to the size returned from cusparseSpMM_buffersize(). The type of arguments "alpha" and "beta" should match the "computeType" argument. See the CUSPARSE Library documentation for the datatype, "computeType", sparse format, and "alg" combinations supported in each release.

```
integer(4) function cusparseSpMM(handle, opA, opB, alpha, matA, matB, beta,
matC, computeType, alg, buffer)
  type(cusparseHandle) :: handle
  integer(4) :: opA, opB
  real(4), device :: alpha, beta ! device or host variable
  type(cusparseSpMatDescr) :: matA
  type(cusparseDnMatDescr) :: matB, matC
  integer(4) :: computeType, alg
  integer(4), device :: buffer(*)
```

Chapter 6. TENSOR PRIMITIVES RUNTIME LIBRARY APIS

This section describes the Fortran interfaces to the CUDA cuTENSOR library. The cuTENSOR functions are only accessible from host code. Most of the runtime API routines, other than some utilities, are functions that return an error code; they return a value of CUTENSOR_STATUS_SUCCESS if the call was successful, or another cuTENSOR status return value if there was an error. Unlike earlier Fortran modules, we have created a cutensorStatus derived type for the return values. We have also overloaded the **.eq.** and **.ne.** logical operators for testing the return status.

Currently we provide two levels of Fortran interfaces to the cuTENSOR library, a low-level module which maps 1-1 to the C interfaces in cuTENSOR v1.0.0, and an experimental high-level module which maps several standard Fortran intrinsic functions to the functionality contained within the cuTENSOR library.

Chapter 8 contains examples of accessing the cuTENSOR library routines from OpenACC and CUDA Fortran. In both cases, the interfaces to the library can be exposed by adding the line **use cutensor** to your program unit.

Unless a specific kind is provided, the plain integer type used in the interfaces implies integer(4) and the plain real type implies real(4).

6.1. CUTENSOR Definitions and Helper Functions

This section contains definitions and data types used in the cuTENSOR library and interfaces to the cuTENSOR helper functions.

The cuTENSOR module contains the following derived type definitions:

```
! Definitions from cutensor.h
integer, parameter :: CUTENSOR_MAJOR = 1
integer, parameter :: CUTENSOR_MINOR = 1
integer, parameter :: CUTENSOR_PATCH = 0
```

```
! Types from cutensor/types.h
! Algorithm Control
type, bind(c) :: cutensorAlgo
    integer(4) :: algo
end type
```

```
type(cutensorAlgo), parameter :: &
    CUTENSOR_ALGO_GETT    = cutensorAlgo(-4), &
    CUTENSOR_ALGO_TGETT   = cutensorAlgo(-3), &
    CUTENSOR_ALGO_TTGT    = cutensorAlgo(-2), &
    CUTENSOR_ALGO_DEFAULT = cutensorAlgo(-1)
```

```
! Workspace Control
type, bind(c) :: cutensorWorksizePreference
    integer(4) :: wksp
end type
type(cutensorWorksizePreference), parameter :: &
    CUTENSOR_WORKSPACE_MIN         = cutensorWorksizePreference(1), &
    CUTENSOR_WORKSPACE_RECOMMENDED = cutensorWorksizePreference(2), &
    CUTENSOR_WORKSPACE_MAX         = cutensorWorksizePreference(3)
```

```
! Unary and Binary Element-wise Operations
type, bind(c) :: cutensorOperator
    integer(4) :: opno
end type
type(cutensorOperator), parameter :: &
    ! Unary
    CUTENSOR_OP_IDENTITY = cutensorOperator(1), & ! Identity operator
    CUTENSOR_OP_SQRT    = cutensorOperator(2), &  ! Square root
    CUTENSOR_OP_RELU    = cutensorOperator(8), &  ! Rectified linear unit
    CUTENSOR_OP_CONJ    = cutensorOperator(9), &  ! Complex conjugate
    CUTENSOR_OP_RCP     = cutensorOperator(10), & ! Reciprocal
    CUTENSOR_OP_SIGMOID = cutensorOperator(11), & ! y=1/(1+exp(-x))
    CUTENSOR_OP_TANH    = cutensorOperator(12), & ! y=tanh(x)
    CUTENSOR_OP_EXP     = cutensorOperator(22), & ! Exponentiation.
    CUTENSOR_OP_LOG     = cutensorOperator(23), & ! Log (base e).
    CUTENSOR_OP_ABS     = cutensorOperator(24), & ! Absolute value.
    CUTENSOR_OP_NEG     = cutensorOperator(25), & ! Negation.
    CUTENSOR_OP_SIN     = cutensorOperator(26), & ! Sine.
    CUTENSOR_OP_COS     = cutensorOperator(27), & ! Cosine.
    CUTENSOR_OP_TAN     = cutensorOperator(28), & ! Tangent.
    CUTENSOR_OP_SINH    = cutensorOperator(29), & ! Hyperbolic sine.
    CUTENSOR_OP_COSH    = cutensorOperator(30), & ! Hyperbolic cosine.
    CUTENSOR_OP_ASIN    = cutensorOperator(31), & ! Inverse sine.
    CUTENSOR_OP_ACOS    = cutensorOperator(32), & ! Inverse cosine.
    CUTENSOR_OP_ATAN    = cutensorOperator(33), & ! Inverse tangent.
    CUTENSOR_OP_ASINH   = cutensorOperator(34), & ! Inverse hyperbolic sine.
    CUTENSOR_OP_ACOSH   = cutensorOperator(35), & ! Inverse hyperbolic cosine.
    CUTENSOR_OP_ATANH   = cutensorOperator(36), & ! Inverse hyperbolic tangent.
    CUTENSOR_OP_CEIL    = cutensorOperator(37), & ! Ceiling.
    CUTENSOR_OP_FLOOR   = cutensorOperator(38), & ! Floor.
    ! Binary
    CUTENSOR_OP_ADD     = cutensorOperator(3), &  ! Addition of two elements
    CUTENSOR_OP_MUL     = cutensorOperator(5), &  ! Multiplication of 2 elements
    CUTENSOR_OP_MAX     = cutensorOperator(6), &  ! Maximum of two elements
    CUTENSOR_OP_MIN     = cutensorOperator(7), &  ! Minimum of two elements
    CUTENSOR_OP_UNKNOWN = cutensorOperator(126)   ! reserved for internal use
 only
```

```
! Status Return Values
type, bind(c) :: cutensorStatus
    integer(4) :: stat
end type
type(cutensorStatus), parameter :: &
    ! The operation completed successfully.
    CUTENSOR_STATUS_SUCCESS                = cutensorStatus(0), &
    ! The cuTENSOR library was not initialized.
    CUTENSOR_STATUS_NOT_INITIALIZED        = cutensorStatus(1), &
    ! Resource allocation failed inside the cuTENSOR library.
    CUTENSOR_STATUS_ALLOC_FAILED           = cutensorStatus(3), &
    ! An unsupported value or parameter was passed to the function.
    CUTENSOR_STATUS_INVALID_VALUE          = cutensorStatus(7), &
    ! Indicates that the device is either not ready,
    ! or the target architecture is not supported.
```

```
    CUTENSOR_STATUS_ARCH_MISMATCH          = cutensorStatus(8), &
    ! An access to GPU memory space failed, which is usually caused
    ! by a failure to bind a texture.
    CUTENSOR_STATUS_MAPPING_ERROR          = cutensorStatus(11), &
    ! The GPU program failed to execute. This is often caused by a
    ! launch failure of the kernel on the GPU, which can be caused by
    ! multiple reasons.
    CUTENSOR_STATUS_EXECUTION_FAILED       = cutensorStatus(13), &
    ! An internal cuTENSOR error has occurred.
    CUTENSOR_STATUS_INTERNAL_ERROR         = cutensorStatus(14), &
    ! The requested operation is not supported.
    CUTENSOR_STATUS_NOT_SUPPORTED          = cutensorStatus(15), &
    ! The functionality requested requires some license and an error
    ! was detected when trying to check the current licensing.
    CUTENSOR_STATUS_LICENSE_ERROR          = cutensorStatus(16), &
    ! A call to CUBLAS did not succeed.
    CUTENSOR_STATUS_CUBLAS_ERROR           = cutensorStatus(17), &
    ! Some unknown CUDA error has occurred.
    CUTENSOR_STATUS_CUDA_ERROR             = cutensorStatus(18), &
    ! The provided workspace was insufficient.
    CUTENSOR_STATUS_INSUFFICIENT_WORKSPACE = cutensorStatus(19), &
    ! Indicates that the driver version is insufficient.
    CUTENSOR_STATUS_INSUFFICIENT_DRIVER    = cutensorStatus(20)
```

```
! Compute Type
type, bind(c) :: cutensorComputeType
    integer(4) :: ctyp
end type
type(cutensorComputeType), parameter :: &
    CUTENSOR_R_MIN_16F = cutensorComputeType(2**0), & ! real as a half
    CUTENSOR_C_MIN_16F = cutensorComputeType(2**1), & ! complex as a half
    CUTENSOR_R_MIN_32F = cutensorComputeType(2**2), & ! real as a float
    CUTENSOR_C_MIN_32F = cutensorComputeType(2**3), & ! complex as a float
    CUTENSOR_R_MIN_64F = cutensorComputeType(2**4), & ! real as a double
    CUTENSOR_C_MIN_64F = cutensorComputeType(2**5), & ! complex as a double
    CUTENSOR_R_MIN_8U  = cutensorComputeType(2**6), & ! real as a uint8
    CUTENSOR_R_MIN_32U = cutensorComputeType(2**7), & ! real as a uint32
    CUTENSOR_R_MIN_8I  = cutensorComputeType(2**8), & ! real as a int8
    CUTENSOR_R_MIN_32I = cutensorComputeType(2**9), & ! real as a int32
    CUTENSOR_R_MIN_16BF = cutensorComputeType(2**10), & ! real as a bfloat16
    CUTENSOR_R_MIN_TF32 = cutensorComputeType(2**11), & ! real as a tf32
    CUTENSOR_C_MIN_TF32 = cutensorComputeType(2**12) ! complex as a tf32
```

```
! cuTENSOR descriptors and other types to hold state
type cutensorHandle
  integer(8) :: fields(512)
end type

type cutensorDescriptor
  integer(8) :: fields(64)
end type

type cutensorContractionDescriptor
  integer(8) :: fields(256)
end type

type cutensorContractionPlan
  integer(8) :: fields(640)
end type

type cutensorContractionFind
  integer(8) :: fields(64)
end type
```

```
! CUDA Datatype whose values are common among libraries
type, bind(c) :: cudaDataType
    integer(4) :: ctyp
end type
```

6.1.1. cutensorInit

This function initializes the cuTENSOR library and returns a handle for subsequent cuTENSOR calls.

```
type(cutensorStatus) function cutensorInit(handle)
   type(cutensorHandle) :: handle
```

6.1.2. cutensorInitTensorDescriptor

This function initializes a cuTENSOR descriptor, given the number of modes, extents, strides, and type of the data.

```
type(cutensorStatus) function cutensorInitTensorDescriptor(handle, desc,
 numModes, extent, stride, dataType, unaryOp)
    type(cutensorHandle) :: handle
    type(cutensorDescriptor) :: desc
    integer(4) :: numModes
    integer(8), dimension(*) :: extent
    integer(8), dimension(*) :: stride
    type(cudaDataType) :: dataType
    type(cutensorOperator) :: unaryOp
```

Altenatively, cuTENSOR will take a value of NULL for the stride, and assume a packed array. You can pass c_null_ptr (from the iso_c_binding module) for the stride in that case.

6.1.3. cutensorGetAlignmentRequirement

This function computes the minimal alignment requirement for a given data pointer and descriptor. The ptr argument can be any type, kind, or rank, but must be device data.

```
type(cutensorStatus) function cutensorGetAlignmentRequirement(handle, ptr, desc,
 alignment)
    type(cutensorHandle) :: handle
    real, device, dimension(*) :: ptr
    type(cutensorDescriptor) :: desc
    integer(4), intent(out) :: alignment
```

6.1.4. cutensorGetErrorString

This function returns the description string for an error code.

```
character*128 function cutensorGetErrorString(ierr)
   type(cutensorStatus) :: ierr
```

6.1.5. cutensorGetVersion

This function returns the version number of the cuTENSOR library.

```
integer(8) function cutensorGetVersion()
```

6.1.6. cutensorGetCudartVersion

This function returns the version of the CUDA runtime that the cuTENSOR library was compiled against.

```
integer(8) function cutensorGetCudartVersion()
```

6.2. CUTENSOR Element-wise Operations

This section contains interfaces for the cuTENSOR functions that perform element-wise operations between tensors.

6.2.1. cutensorPermutation

This function performs an out-of-place tensor permutation of the form

B=alpha*op(perm(A))

The type and kind of the **alpha** scalar is determined by the typeScalar argument. The arrays **A, B** can be of any supported type, kind, and rank. The permutations of **A, B** are set up using the mode arguments. The operations op(perm(A)) are set up in the call to **cutensorInitTensorDescriptor()** via the **unaryOp** argument.

```
type(cutensorStatus) function cutensorPermutation(handle, &
    alpha, A, descA, modeA, B, descB, modeB, typeScalar, stream)
    type(cutensorHandle) :: handle
    type(cutensorDescriptor) :: descA, descB
    real, device, dimension(*) :: A, B
    real :: alpha
    integer(4), dimension(*) :: modeA, modeB
    type(cudaDataType) :: typeScalar
    integer(kind=cuda_stream_kind) :: stream
```

6.2.2. cutensorElementwiseBinary

This function performs an element-wise tensor operation on two inputs of the form

D=opAC(alpha*op(perm(A)),gamma*op(perm(C)))

The **opAC** argument is an element-wise binary operator. The type and kind of the **alpha, gamma** scalars is determined by the typeScalar argument. The arrays **A, C, D** can be of any supported type, kind, and rank. The permutations of **A, C, D** are set up using the mode arguments. The operations op(perm(A)), op(perm(C)), etc. are set up in the call to **cutensorInitTensorDescriptor()** via the **unaryOp** argument.

```
type(cutensorStatus) function cutensorElementwiseBinary(handle, &
    alpha, A, descA, modeA, gamma, C, descC, modeC, &
    D, descD, modeD, opAC, typeScalar, stream)
    type(cutensorHandle) :: handle
    type(cutensorDescriptor) :: descA, descC, descD
    real, device, dimension(*) :: A, C, D
    real :: alpha, gamma
    integer(4), dimension(*) :: modeA, modeC, modeD
    type(cutensorOperator) :: opAC
    type(cudaDataType) :: typeScalar
    integer(kind=cuda_stream_kind) :: stream
```

6.2.3. cutensorElementwiseTrinary

This function performs an element-wise tensor operation on three inputs of the form

D=opABC(opAB(alpha*op(perm(A)),beta*op(perm(B))),gamma*op(perm(C)))

The **opABC, opAB** arguments are element-wise binary operators. The type and kind of the **alpha, beta, gamma** scalars is determined by the typeScalar argument. The arrays **A, B, C, D** can be of any supported type, kind, and rank. The permutations of **A, B, C, D** are set up using the mode arguments. The operations op(perm(A)), op(perm(B)), etc. are set up in the call to **cutensorInitTensorDescriptor()** via the **unaryOp** argument.

```
type(cutensorStatus) function cutensorElementwiseTrinary(handle, &
    alpha, A, descA, modeA, beta, B, descB, modeB, gamma, C, descC, modeC, &
    D, descD, modeD, opAB, opABC, typeScalar, stream)
    type(cutensorHandle) :: handle
    type(cutensorDescriptor) :: descA, descB, descC, descD
    real, device, dimension(*) :: A, B, C, D
    real :: alpha, beta, gamma
    integer(4), dimension(*) :: modeA, modeB, modeC, modeD
    type(cutensorOperator) :: opAB, opABC
    type(cudaDataType) :: typeScalar
    integer(kind=cuda_stream_kind) :: stream
```

6.3. CUTENSOR Reduction Operations

This section contains interfaces for the cuTENSOR functions that perform reduction operations on tensors.

6.3.1. cutensorReductionGetWorkspace

This function determines the workspace needed for a given tensor reduction performed by **cutensorReduction()**.

```
type(cutensorStatus) function cutensorReductionGetWorkspace(handle, &
    A, descA, modeA, C, descC, modeC, D, descD, modeD, &
    opReduce, minTypeCompute, workspaceSize)

    type(cutensorHandle) :: handle
    real, device, dimension(*) :: A, C, D
    type(cutensorDescriptor) :: descA, descC, descD
    integer(4), dimension(*) :: modeA, modeC, modeD
    type(cutensorOperator) :: opReduce
    type(cutensorComputeType) :: minTypeCompute
    integer(8), intent(out) :: workspaceSize
```

6.3.2. cutensorReduction

This function performs a tensor reduction of the form **D = alpha * opReduce(op(A)) + beta * opC(C)**. The **opReduce** argument controls what type of reduction is performed. The arrays A, C, and D can be of any supported type, kind, and rank. The alpha and beta scalars' type and kind is determined by the minTypeCompute argument.

```
type(cutensorStatus) function cutensorReduction(handle, &
    alpha, A, descA, modeA, beta, C, descC, modeC, D, descD, modeD, &
    opReduce, minTypeCompute, workspace, workspaceSize, stream)

    type(cutensorHandle) :: handle
    real, device, dimension(*) :: A, C, D
    type(cutensorDescriptor) :: descA, descC, descD
    integer(4), dimension(*) :: modeA, modeC, modeD
    real :: alpha, beta
    type(cutensorOperator) :: opReduce
    type(cutensorComputeType) :: minTypeCompute
```

```
    real, device, dimension(*) :: workspace
    integer(8) :: workspaceSize
    integer(kind=cuda_stream_kind) :: stream
```

6.4. CUTENSOR Contraction Operations

This section contains interfaces for the cuTENSOR functions that perform contraction operations between tensors.

6.4.1. cutensorInitContractionDescriptor

This function initializes the contraction descriptor for a contraction problem of the form

D = alpha*(AxB) + beta*C

The arrays **A, B, C, D** can be of any supported type, kind, and rank. The permutations of **A, B, C, D** are set up using the mode arguments. The strides and extents of the arrays**A, B, C, D** are set up in the array descriptors by calls to **cutensorInitTensorDescriptor()**. The alignment requirements for each array can be found by calls to **cutensorGetAlignmentRequirement**.

```
type(cutensorStatus) function cutensorInitContractionDescriptor(handle, desc, &
    descA, modeA, algnA, descB, modeB, algnB, &
    descC, modeC, algnC, descD, modeD, algnD, computeType)
    type(cutensorHandle) :: handle
    type(cutensorContractionDescriptor) :: desc
    integer(4), dimension(*) :: modeA, modeB, modeC, modeD
    integer(4) :: algnA, algnB, algnC, algnD
    type(cutensorComputeType) :: computeType
```

6.4.2. cutensorInitContractionFind

This function limits the search space of viable algorithms for contraction operations.

```
type(cutensorStatus) function cutensorInitContractionFind(handle, &
    find, algo)
    type(cutensorHandle) :: handle
    type(cutensorContractionFind) :: find
    type(cutensorAlgo) :: algo
```

6.4.3. cutensorInitContractionPlan

This function initializes the contraction plan for a tensor contraction operation.

```
type(cutensorStatus) function cutensorInitContractionPlan(handle, &
    plan, desc, find, workspaceize)
    type(cutensorHandle) :: handle
    type(cutensorContractionPlan) :: plan
    type(cutensorContractionDescriptor) :: desc
    type(cutensorContractionFind) :: find
    integer(8) :: workspaceSize
```

6.4.4. cutensorContractionGetWorkspace

This function determines the workspace needed for a given tensor contraction performed by **cutensorContraction()**.

```
type(cutensorStatus) function cutensorContractionGetWorkspace(handle, &
    desc, find, pref, workspaceSize)
```

```
    type(cutensorHandle) :: handle
    type(cutensorContractionDescriptor) :: desc
    type(cutensorContractionFind) :: find
    type(cutensorWorksizePreference) :: pref
    integer(8), intent(out) :: workspaceSize
```

6.4.5. cutensorContractionMaxAlgos

This function returns the maximum number of algorithms available to perform tensor contractions.

```
type(cutensorStatus) function cutensorContractionMaxAlgos(maxNumAlgos)
    integer(4), intent(out) :: maxNumAlgos
```

6.4.6. cutensorContraction

This function performs a tensor contraction of the form

`D = alpha*(AxB) + beta*C`

The arrays A, B, C, and D can be of any supported type, kind, and rank. The alpha and beta scalars type and kind is determined by the typeCompute argument given to **`cutensorInitContractionDesciptor()`** and encoded in the plan.

```
type(cutensorStatus) function cutensorContraction(handle, &
    plan, alpha, A, B, beta, C, D, workspace, workspaceSize, stream)
    type(cutensorHandle) :: handle
    type(cutensorContractionPlan) :: plan
    real, device, dimension(*) :: A, B, C, D
    real :: alpha, beta
    real, device, dimension(*) :: workspace
    integer(8) :: workspaceSize
    integer(kind=cuda_stream_kind) :: stream
```

6.5. CUTENSOR Fortran Extensions

This section contains extensions to the cuTENSOR interfaces for Fortran array intrinsic function operations, array expressions, and array assignment, built upon the cuTENSOR library. In CUDA Fortran, these operations take data with the device or managed attribute. In OpenACC, they can be invoked with a host_data construct. The interfaces to these extensions are exposed by adding the line **`use cutensorEx`** to your program unit, or can be applied to specific statements using the Fortran block feature, as shown in the next example.

```
block; use cutensorEx
  D = reshape(A,shape=[ni,nk,nj],order=[1,3,2])
end block
```

To enable the operations to take place in one underlying kernel invocation, the RHS expression is deferred until the overloaded assignment operation has both the LHS and RHS available. This guarantees performance on par with the low-level cuTENSOR API whenever possible. Generality is affected, so only specific forms of the Fortran statements are currently supported, and those are documented in the subsequent sections of this chapter.

Since the cuTENSOR library operations take a stream argument, we have added a way to set a cuTENSOR default stream that our runtime will maintain, one copy per CPU thread. That is:

```
integer function cutensorexSetStream(stream)
integer(kind=cuda_stream_kind) :: stream
```

6.5.1. Fortran Reshape

This Fortran function changes the shape of an array and possibly permutes the dimensions and layout. It is invoked as:

`D = alpha * func(reshape(A, shape=[...], order=[...]))`

The arrays A and D can be of type real(2), real(4), real(8), complex(4), or complex(8). The rank (number of dimensions) of A and D can be from 1 to 7. The alpha value is expected to be the same type as A, or as func(reshape(A)), if that differs. Accepted functions which can be applied to the result of reshape are listed at the end of this section. The pad argument to the F90 reshape function is not currently supported. This Fortran call, besides initialization and setting up cuTENSOR descriptors, maps to **`cutensorPermutation()`**.

```
! Example to switch the 2nd and 3rd dimension layout
D = reshape(a,shape=[ni,nk,nj], order=[1,3,2])
! Same example, take the absolute value and scale by 2.5
D = 2.5 * abs(reshape(a,shape=[ni,nk,nj], order=[1,3,2]))
```

6.5.2. Fortran Transpose

This Fortran function tranposes a matrix (a 2-dimensional array). It is invoked as:

`D = alpha * func(transpose(A))`

The arrays A and D can be of type real(2), real(4), real(8), complex(4), or complex(8). The rank (number of dimensions) of A and D is 2. Applying scaling (the **`alpha`** argument) or applying a function to the transpose result is optional. The alpha value is expected to be the same type as A, or as func(transpose(A)), if that differs. Accepted functions which can be applied to the result of the transpose are listed at the end of this section. This Fortran call, besides initialization and setting up cuTENSOR descriptors, maps to **`cutensorPermutation()`**.

```
! Example of transpose
D = transpose(A)
! Same example, take the absolute value and scale by 2.5
D = 2.5 * abs(tranpose(A))
```

6.5.3. Fortran Spread

This Fortran function increases the rank of an array by one across the specified dimension and broadcasts the values over the new dimension. It is invoked as:

`D = alpha * func(spread(A, dim=i, ncopies=n))`

The arrays A and D can be of type real(2), real(4), real(8), complex(4), or complex(8). The rank (number of dimensions) of A and D can be from 1 to 7. The alpha value is expected to be the same type as A. Accepted functions which can be applied to the result

of spread are listed at the end of this section. This Fortran call, besides initialization and setting up cuTENSOR descriptors, maps to **cutensorPermutation()**.

```
! Example to add and broadcast values over the new first dimension
D = spread(A, dim=1, ncopies=n1)
! Same example, take the absolute value and scale by 2.5
D = 2.5 * abs(spread(A, dim=1, ncopies=n1))
```

6.5.4. Fortran Element-wise Expressions

There is some limited support for converting expressions involving two or three source arrays into cuTENSOR calls. The first one or two operands can be a permuted array, the result of a call to reshape(), transpose(), or spread(). An elemental function can be applied to the array operands, permuted or not, and they can also be scaled. Here are some supported forms:

```
D = A + B

D = permute(A) + B

D = func(permute(A)) + permute(B)

D = alpha * func(permute(A) + beta * permute(B) + gamma * C
```

The arrays A, B, C, and D can be of type real(2), real(4), real(8), complex(4), or complex(8). The rank (number of dimensions) of A, B, C, and D can be from 1 to 7. For the three-operand case, arrays C and D must have the same shape, strides, and type. The alpha value is expected to be the same type as A. The same applies for beta and B, and gamma and C. The Fortran wrapper does no type conversion, though cuTENSOR may. Compile-time checking of array conformance is limited. Other runtime checks for unsupported combinations may come from either the Fortran wrapper or from cuTENSOR. Accepted functions which can be applied to permuted or unpermuted arrays are listed at the end of this section. These Fortran expressions, besides initialization and setting up cuTENSOR descriptors, map to either **cutensorElementwiseBinary()** or **cutensorElementwiseTrinary()**.

```
! Example to scale and add two arrays together
D = alpha * A + beta * B
! Same example, take the absolute value of A and B and add to C
D = alpha * abs(A) + beta * abs(B) + C
! Transpose the first array before adding to the second
D = alpha * abs(transpose(A)) + beta * abs(B) + C
```

6.5.5. Fortran Matmul Operations

Matrix multiplication is one instance of tensor contraction. Either operand to matmul can be a permuted array, the result of a call to reshape(), transpose(), or spread(). The cuTENSOR library does not currently support applying an elemental function to the array operands, but the result and accumlator can be scaled. Here are some supported forms:

```
D = matmul(A, B)

D = matmul(permute(A), B)

D = matmul(A, permute(B))
```

D = matmul(permute(A), permute(B))

D = C + matmul(A, B)

D = C - matmul(A, B)

D = alpha * matmul(A, B) + beta * C

The arrays A, B, C, and D can be of type real(2), real(4), real(8), complex(4), or complex(8). The rank (number of dimensions) of A, B, C, and D must be 2, after any permutations. Arrays C and D must currently have the same shape, strides, and type. The alpha value is expected to be the same type as A and B. The beta value should have the same type as C. The Fortran wrapper does no type conversion, though cuTENSOR may. Compile-time checking of array conformance is limited. Other runtime checks for unsupported combinations may come from either the Fortran wrapper or from cuTENSOR. Fortran support for **Matmul**, besides initialization and setting up cuTENSOR descriptors, maps to **cutensorContraction()**.

```
! Example to multiply two matrices together
D = matmul(A, B)
! Same example, accumulate into C
C = C +  matmul(A, B)
! Same example, transpose the first argument
C = C + matmul(transpose(A), B)
```

On GPUs which support the TF32 type, to direct a contraction to use the compute type CUTENSOR_R_MIN_TF32 rather than CUTENSOR_R_MIN_32F for real(4) (or similar for complex(4)), we have provided a way to set an internal parameter, similar to the default stream, that our runtime will maintain. The default opt level is 0. Setting the opt level to be greater than 0 will use CUTENSOR_R_MIN_TF32.

```
integer function cutensorExSetOptLevel(level)
integer(4) :: level
```

6.5.6. Supported Element-wise Functions

From the list of supported element-wise functions for the cuTENSOR definitions listed above, several are supported from the high-level interface. The cuTENSOR library does not currently support many of the functions listed for complex data; consult the cuTENSOR documentation for the latest information. Here are the functions with at least some level of Fortran interface support:

SQRT RELU CONJG RCP SIGMOID

TANH EXP LOG ABS NEG

SIN COS TAN SINH COSH

ASIN ACOS ATAN ASINH ACOSH

ATANH CEIL FLOOR

Note the C complex conjugate function **conj** is spelled **conjg** in Fortran. Also, the functions for **ceil** and **floor** differ between C and Fortran, in their return type. We have kept the C spelling and behavior (returning a real, not an integer).

Only **ABS, CEIL, CONJG, COS, SIN** can be used on bare arrays in the current implementation. All functions can be applied to the result of a permutation. Use

`reshape(A, shape=shape(A))` as a NOP to put the bare array into a form that can currently be recognized.

Users will find that the performance of binary or trinary kernels, such as:

```
D = sin(A) + cos(B) + C
```

will not perform as well as kernels written and compiled for that specific operation (using CUDA, CUDA Fortran, or OpenACC) due to overhead in the cuTENSOR kernels needed for generally applying functions to the operands. If the operation permutes the operands, such as:

```
D = sin(permute(A)) + cos(permute(B)) + C
```

users may see good performance compared to other naive implementations depending on how complex the permutations on the input arrays are.

Chapter 7. NVIDIA COLLECTIVE COMMUNICATIONS LIBRARY (NCCL) APIS

This section describes the Fortran interfaces to the NCCL library. The NCCL functions are only accessible from host code. Most of the runtime API routines, other than some utilities, are functions that return an error code; they return a value of ncclSuccess if the call was successful, or another value if there was an error. Unlike earlier Fortran modules, we have created a ncclResult derived type for the return values. We have also overloaded the **.eq.** and **.ne.** logical operators for testing the return status.

Unless a specific kind is provided, the plain integer type used in the interfaces implies integer(4) and the plain real type implies real(4).

7.1. NCCL Definitions and Helper Functions

This section contains definitions and data types used in the NCCL library and interfaces to the NCCL communicator creation and management functions.

The Fortran NCCL module contains the following derived type definitions:

```
! Definitions from nccl.h
integer, parameter :: NCCL_MAJOR = 2
integer, parameter :: NCCL_MINOR = 7
integer, parameter :: NCCL_PATCH = 3
```

```
! Types from nccl.h
! ncclUniqueId
type, bind(c) :: ncclUniqueId
  character(c_char) :: internal(NCCL_UNIQUE_ID_BYTES)
end type ncclUniqueId

! ncclComm
type, bind(c) :: ncclComm
  type(c_ptr) :: member
end type ncclComm

! ncclResult
type, bind(c) :: ncclResult
  integer(c_int) :: member
end type ncclResult

type(ncclResult), parameter :: &
```

```
    ncclSuccess             = ncclResult(0), &
    ncclUnhandledCudaError  = ncclResult(1), &
    ncclSystemError         = ncclResult(2), &
    ncclInternalError       = ncclResult(3), &
    ncclInvalidArgument     = ncclResult(4), &
    ncclInvalidUsage        = ncclResult(5), &
    ncclNumResults          = ncclResult(6)
```

```
! ncclDataType
type, bind(c) :: ncclDataType
  integer(c_int) :: member
end type ncclDataType

type(ncclDataType), parameter :: &
    ncclInt8     = ncclDataType(0), &
    ncclChar     = ncclDataType(0), &
    ncclUint8    = ncclDataType(1), &
    ncclInt32    = ncclDataType(2), &
    ncclInt      = ncclDataType(2), &
    ncclUint32   = ncclDataType(3), &
    ncclInt64    = ncclDataType(4), &
    ncclUint64   = ncclDataType(5), &
    ncclFloat16  = ncclDataType(6), &
    ncclHalf     = ncclDataType(6), &
    ncclFloat32  = ncclDataType(7), &
    ncclFloat    = ncclDataType(7), &
    ncclFloat64  = ncclDataType(8), &
    ncclDouble   = ncclDataType(8), &
    ncclNumTypes = ncclDataType(9)
```

```
! ncclRedOp
type, bind(c) :: ncclRedOp
  integer(c_int) :: member
end type ncclRedOp

type(ncclRedOp), parameter :: &
    ncclSum    = ncclRedOp(0), &
    ncclProd   = ncclRedOp(1), &
    ncclMax    = ncclRedOp(2), &
    ncclMin    = ncclRedOp(3), &
    ncclNumOps = ncclRedOp(4)
```

7.1.1. ncclGetVersion

This function returns the version number of the NCCL library.

```
type(ncclResult) function ncclGetVersion(version)
integer(4) :: version
```

7.1.2. ncclGetUniqueId

This function generates an ID to be used with ncclCommInitRank. This routine should be called once, and the generated ID should be distributed to all ranks.

```
type(ncclResult) function ncclGetUniqueId(uniqueId)
type(ncclUniqueId) :: uniqueId
```

7.1.3. ncclCommInitRank

This function generates a new NCCL communicator, of type(ncclCom). The rank argument must be between 0 and nranks-1. The uniqueId argument should be generated with ncclGetUniqueId.

```
type(ncclResult) function ncclCommInitRank(comm, nranks, uniqueId, rank)
```

```
type(ncclComm) :: comm
integer(4) :: nranks
type(ncclUniqueId) :: uniqueId
integer(4) :: rank
```

7.1.4. ncclCommInitAll

This function creates a single-process communicator clique, an array of type(ncclCom).

```
type(ncclResult) function ncclCommInitAll(comms, ndev, devlist)
type(ncclComm) :: comms(*)
integer(4) :: ndev
integer(4) :: devlist(*)
```

7.1.5. ncclCommDestroy

This function frees resources allocated to a NCCL communicator. It will wait for uncompleted operations.

```
type(ncclResult) function ncclCommDestroy(comm)
type(ncclComm) :: comm
```

7.1.6. ncclCommAbort

This function frees resources allocated to a NCCL communicator. It will abort uncompleted operations.

```
type(ncclResult) function ncclCommAbort(comm)
type(ncclComm) :: comm
```

7.1.7. ncclGetErrorString

This function returns an error string for a given ncclResult value.

```
character*128 function ncclGetErrorString(ierr)
type(ncclResult) :: ierr
```

7.1.8. ncclCommGetAsyncError

This function queries whether the communicator has encountered any asynchronous errors.

```
type(ncclResult) function ncclCommGetAsyncError(comm, asyncError)
type(ncclComm) :: comm
type(ncclResult) :: asyncError
```

7.1.9. ncclCommCount

This function sets the count argument to the number of ranks in the NCCL communicator.

```
type(ncclResult) function ncclCommCount(comm, count)
type(ncclComm) :: comm
integer(4) :: count
```

7.1.10. ncclCommCuDevice

This function sets the device argument to the CUDA device associated with a NCCL communicator.

```
type(ncclResult) function ncclCommCuDevice(comm, device)
type(ncclComm) :: comm
integer(4) :: device
```

7.1.11. ncclCommUserRank

This function sets the rank argument to the rank within a NCCL communicator.

```
type(ncclResult) function ncclCommUserRank(comm, rank)
type(ncclComm) :: comm
integer(4) :: rank
```

7.2. NCCL Collective Communication Functions

This section contains interfaces for the NCCL functions that perform collective communication operations on device data. All functions can take either CUDA Fortran device arrays, OpenACC arrays within a host_data use_device data directive, or Fortran type(c_devptr) arguments.

7.2.1. ncclAllReduce

This function performs the specified reduction on data across devices and writes the results into the receive buffer of every rank.

```
type(ncclResult) function ncclAllReduce(sendbuff, recvbuff, &
    count, datatype, op, comm, stream)
type(c_devptr) :: sendbuff, recvbuff
  ! These combinations of sendbuff, recvbuff are also accepted:
  ! integer(4), device :: sendbuff(*), recvbuff(*)
  ! integer(8), device :: sendbuff(*), recvbuff(*)
  ! real(2), device :: sendbuff(*), recvbuff(*)
  ! real(4), device :: sendbuff(*), recvbuff(*)
  ! real(8), device :: sendbuff(*), recvbuff(*)
integer(cuda_count_kind) :: count
type(ncclDataType) :: datatype
type(ncclRedOp) :: op
type(ncclComm) :: comm
integer(cuda_stream_kind) :: stream
```

7.2.2. ncclBroadcast

This function copies the send buffer on the root rank to all other ranks in the NCCL communicator. An in-place operation will happen if sendbuff and recvbuff are the same address.

```
type(ncclResult) function ncclBroadcast(sendbuff, recvbuff, &
    count, datatype, root, comm, stream)
type(c_devptr) :: sendbuff, recvbuff
  ! These combinations of sendbuff, recvbuff are also accepted:
  ! integer(4), device :: sendbuff(*), recvbuff(*)
  ! integer(8), device :: sendbuff(*), recvbuff(*)
  ! real(2), device :: sendbuff(*), recvbuff(*)
```

```
  ! real(4), device :: sendbuff(*), recvbuff(*)
  ! real(8), device :: sendbuff(*), recvbuff(*)
integer(cuda_count_kind) :: count
type(ncclDataType) :: datatype
integer(4) :: root
type(ncclComm) :: comm
integer(cuda_stream_kind) :: stream
```

7.2.3. ncclReduce

This function performs the same operation as AllReduce, but writes the results only to the receive buffers of the specified root rank.

```
type(ncclResult) function ncclReduce(sendbuff, recvbuff, &
    count, datatype, op, root, comm, stream)
type(c_devptr) :: sendbuff, recvbuff
  ! These combinations of sendbuff, recvbuff are also accepted:
  ! integer(4), device :: sendbuff(*), recvbuff(*)
  ! integer(8), device :: sendbuff(*), recvbuff(*)
  ! real(2), device :: sendbuff(*), recvbuff(*)
  ! real(4), device :: sendbuff(*), recvbuff(*)
  ! real(8), device :: sendbuff(*), recvbuff(*)
integer(cuda_count_kind) :: count
type(ncclDataType) :: datatype
type(ncclRedOp) :: op
integer(4) :: root
type(ncclComm) :: comm
integer(cuda_stream_kind) :: stream
```

7.2.4. ncclAllGather

This function gathers the send buffers from each rank and stores them in rank order in the receive buffer of all ranks.

```
type(ncclResult) function ncclAllGather(sendbuff, recvbuff, &
    sendcount, datatype, comm, stream)
type(c_devptr) :: sendbuff, recvbuff
  ! These combinations of sendbuff, recvbuff are also accepted:
  ! integer(4), device :: sendbuff(*), recvbuff(*)
  ! integer(8), device :: sendbuff(*), recvbuff(*)
  ! real(2), device :: sendbuff(*), recvbuff(*)
  ! real(4), device :: sendbuff(*), recvbuff(*)
  ! real(8), device :: sendbuff(*), recvbuff(*)
integer(cuda_count_kind) :: sendcount
type(ncclDataType) :: datatype
type(ncclComm) :: comm
integer(cuda_stream_kind) :: stream
```

7.2.5. ncclReduceScatter

This function performs the specified reduction on the data, and leaves the result scattered in equal blocks among the ranks, based on the rank index.

```
type(ncclResult) function ncclReduceScatter(sendbuff, recvbuff, &
    recvcount, datatype, op, comm, stream)
type(c_devptr) :: sendbuff, recvbuff
  ! These combinations of sendbuff, recvbuff are also accepted:
  ! integer(4), device :: sendbuff(*), recvbuff(*)
  ! integer(8), device :: sendbuff(*), recvbuff(*)
  ! real(2), device :: sendbuff(*), recvbuff(*)
  ! real(4), device :: sendbuff(*), recvbuff(*)
  ! real(8), device :: sendbuff(*), recvbuff(*)
integer(cuda_count_kind) :: recvcount
type(ncclDataType) :: datatype
```

```
type(ncclRedOp) :: op
type(ncclComm) :: comm
integer(cuda_stream_kind) :: stream
```

7.3. NCCL Point To Point Communication Functions

This section contains interfaces for the NCCL functions that perform point to point communication operations on device data. All functions can take either CUDA Fortran device arrays, OpenACC arrays within a host_data use_device data directive, or Fortran type(c_devptr) arguments. The point to point operations were added in NCCL 2.7.

7.3.1. ncclSend

This function sends data from the send buffer to a communicator peer. This operation blocks the GPU. The receiving peer must call ncclRecv, with the same datatype and count.

```
type(ncclResult) function ncclSend(sendbuff, &
    count, datatype, peer, comm, stream)
type(c_devptr) :: sendbuff
  ! These types for sendbuff are also accepted:
  ! integer(4), device :: sendbuff(*)
  ! integer(8), device :: sendbuff(*)
  ! real(2), device :: sendbuff(*)
  ! real(4), device :: sendbuff(*)
  ! real(8), device :: sendbuff(*)
integer(cuda_count_kind) :: count
type(ncclDataType) :: datatype
integer(4) :: peer
type(ncclComm) :: comm
integer(cuda_stream_kind) :: stream
```

7.3.2. ncclRecv

This function receives data from a communicator peer. This operation blocks the GPU. The sending peer must call ncclSend, with the same datatype and count.

```
type(ncclResult) function ncclRecv(recvbuff, &
    count, datatype, peer, comm, stream)
type(c_devptr) :: recvbuff
  ! These types for recvbuff are also accepted:
  ! integer(4), device :: recvbuff(*)
  ! integer(8), device :: recvbuff(*)
  ! real(2), device :: recvbuff(*)
  ! real(4), device :: recvbuff(*)
  ! real(8), device :: recvbuff(*)
integer(cuda_count_kind) :: count
type(ncclDataType) :: datatype
integer(4) :: peer
type(ncclComm) :: comm
integer(cuda_stream_kind) :: stream
```

7.4. NCCL Group Calls

This section contains interfaces for the NCCL functions that begin and end a group such that multiple calls can be merged.

7.4.1. ncclGroupStart

This function starts a group call. All subsequent calls to NCCL functions may not block due to inter-CPU synchronization.

```
type(ncclResult) function ncclGroupStart()
```

7.4.2. ncclGroupEnd

This function ends a group call. It returns when all operations since the corresponding call to ncclGroupStart have been processed, but not necessarily completed.

```
type(ncclResult) function ncclGroupEnd()
```

Chapter 8. NVSHMEM COMMUNICATION LIBRARY APIS

This section describes the Fortran interfaces to the NVSHMEM library. NVSHMEM is a software library that implements the OpenSHMEM application programming interface (API) for clusters of NVIDIA GPUs. OpenSHMEM is a community standard, one-sided communication API that provides a partitioned global address space (PGAS) parallel programming model. NVSHMEM provides an easy-to-use host-side interface for allocating symmetric memory, which can be distributed across a cluster of NVIDIA GPUs interconnected with NVLink, PCIe, and InfiniBand. The NVSHMEM communication functions are accessible from both host and device code. Most of the runtime API routines are written as C void functions, and we have implemented their Fortran wrappers as subroutines.

Unless a specific kind is provided, the plain integer type used in the interfaces implies integer(4) and the plain real type implies real(4).

8.1. NVSHMEM Definitions, Setup, Exit, and Query Functions

This section contains definitions and data types used in the NVSHMEM library and interfaces to the NVSHMEM initialization and access to the parallel environment of the PEs.

The Fortran NVSHMEM module contains the following constant and derived type definitions:

```
! These are not available to the user, internal only
! defines, from nvshmemx_api.h
#define INIT_HANDLE_BYTES 128

! defines, from nvshmem_constants.h
#define SYNC_SIZE 27648
```

```
! Constant Definitions
integer, parameter :: NVSHMEM_SYNC_VALUE = 0
integer, parameter :: NVSHMEM_SYNC_SIZE = (2 * SYNC_SIZE)
integer, parameter :: NVSHMEM_BARRIER_SYNC_SIZE = (2 * SYNC_SIZE)
```

```
integer, parameter :: NVSHMEM_BCAST_SYNC_SIZE = SYNC_SIZE
integer, parameter :: NVSHMEM_REDUCE_SYNC_SIZE = SYNC_SIZE
integer, parameter :: NVSHMEM_REDUCE_MIN_WRKDATA_SIZE = SYNC_SIZE
integer, parameter :: NVSHMEM_COLLECT_SYNC_SIZE = SYNC_SIZE
integer, parameter :: NVSHMEM_ALLTOALL_SYNC_SIZE = SYNC_SIZE

integer, parameter :: NVSHMEMX_CMP_EQ = 0
integer, parameter :: NVSHMEMX_CMP_NE = 1
integer, parameter :: NVSHMEMX_CMP_GT = 2
integer, parameter :: NVSHMEMX_CMP_LE = 3
integer, parameter :: NVSHMEMX_CMP_LT = 4
integer, parameter :: NVSHMEMX_CMP_GE = 5

integer, parameter :: NVSHMEMX_THREAD_SINGLE     = 0
integer, parameter :: NVSHMEMX_THREAD_FUNNELED   = 1
integer, parameter :: NVSHMEMX_THREAD_SERIALIZED = 2
integer, parameter :: NVSHMEMX_THREAD_MULTIPLE   = 3

integer, parameter :: NVSHMEM_TEAM_INVALID = 0
integer, parameter :: NVSHMEM_TEAM_WORLD   = 1
integer, parameter :: NVSHMEMX_TEAM_NODE   = 2

integer, parameter :: NVSHMEMX_INIT_THREAD_PES    = 1
integer, parameter :: NVSHMEMX_INIT_WITH_MPI_COMM = 2
integer, parameter :: NVSHMEMX_INIT_WITH_SHMEM    = 4
integer, parameter :: NVSHMEMX_INIT_WITH_HANDLE   = 8
```

```
! Types from nvshmemx_api.h
type, bind(c) :: nvshmemx_init_handle
  character(c_char) :: content(INIT_HANDLE_BYTES)
end type nvshmemx_init_handle
```

```
! Types from nvshmemx_api.h
type, bind(c) :: nvshmemx_init_attr_type
  integer(8)  heap_size
  integer(4)  num_threads
  integer(4)  n_pes
  integer(4)  my_pe
  type(c_ptr) mpi_comm
  type(nvshmemx_init_handle) handle
end type nvshmemx_init_attr_type
```

```
! nvshmemx_status, from nvshmem_error.h
type, bind(c) :: nvshmemx_status
  integer(c_int) :: member
end type nvshmemx_status

type(nvshmemx_status), parameter :: &
  NVSHMEMX_SUCCESS = nvshmemx_status(0), &
  NVSHMEMX_ERROR_INVALID_VALUE = nvshmemx_status(1), &
  NVSHMEMX_ERROR_OUT_OF_MEMORY = nvshmemx_status(2), &
  NVSHMEMX_ERROR_NOT_SUPPORTED = nvshmemx_status(3), &
  NVSHMEMX_ERROR_SYMMETRY = nvshmemx_status(4), &
  NVSHMEMX_ERROR_GPU_NOT_SELECTED = nvshmemx_status(5), &
  NVSHMEMX_ERROR_COLLECTIVE_LAUNCH_FAILED = nvshmemx_status(6), &
  NVSHMEMX_ERROR_INTERNAL = nvshmemx_status(7)
```

8.1.1. nvshmem_init

This subroutine allocates and initializes resources used by the NVSHMEM library.

```
subroutine nvshmem_init()
```

8.1.2. nvshmemx_init_attr

This function initializes the NVSHMEM library based on an existing MPI communicator. Since the C and Fortran mpi_comm objects differ, this function has a different argument list than the corresponding C library entry point.

```
type(nvshmemx_status) function nvshmemx_init_attr(flags, comm)
integer(4) :: flags, comm
```

Here is an example of using this function with MPI

```
use nvshmem
type(nvshmemx_status) :: nvstat
. . .
! Setup MPI
call MPI_Init(ierror)
call MPI_Comm_rank(MPI_COMM_WORLD, my_rank, ierror)
call MPI_Comm_size(MPI_COMM_WORLD, nranks, ierror)
!
nvstat = nvshmemx_init_attr(NVSHMEMX_INIT_WITH_MPI_COMM, MPI_COMM_WORLD)
```

8.1.3. nvshmem_my_pe

This function returns the PE number of the calling PE, a number between 0 and npes-1.

```
integer(4) function nvshmem_my_pe()
```

8.1.4. nvshmem_n_pes

This function returns the number of PEs running in the program.

```
integer(4) function nvshmem_n_pes()
```

8.1.5. nvshmem_team_my_pe

This function returns the PE number of the calling PE, within the specified team.

```
integer(4) function nvshmem_team_my_pe(team)
integer(4) :: team
```

8.1.6. nvshmem_team_n_pes

This function returns the number of PEs in the specified team.

```
integer(4) function nvshmem_team_n_pes(team)
integer(4) :: team
```

8.1.7. nvshmem_info_get_version

This subroutine returns the major and minor version number of the NVSHMEM library.

```
subroutine nvshmem_info_get_version(major, minor)
integer(4) :: major, minor
```

8.1.8. nvshmem_info_get_name

This subroutine returns the vendor-defined name string for the library.

```
subroutine nvshmem_info_get_version(name)
character*256, intent(out) :: name
```

8.1.9. nvshmem_finalize

This subroutine releases resources and ends the NVSHMEM portion of a program started with nvshmem_init().

```
subroutine nvshmem_finalize()
```

8.1.10. nvshmem_ptr

This function returns a local address that may be used to directly reference the destination data on the specified PE. The function nvshmem_ptr is implemented as a Fortran generic function, and can take any datatype, as long as it is a symmetric address.

```
type(c_devptr) function nvshmem_ptr(dest, pe)
dest can be of type integer, logical, real, complex, character, or a
 type(c_devptr)
integer(4) :: pe
```

The following specific functions are also supported:

```
type(c_devptr) function nvshmem_ptri(dest, pe)
integer :: dest ! Any kind and rank
integer(4) :: pe
```

```
type(c_devptr) function nvshmem_ptrl(dest, pe)
logical :: dest ! Any kind and rank
integer(4) :: pe
```

```
type(c_devptr) function nvshmem_ptrr(dest, pe)
real :: dest ! Any kind and rank
integer(4) :: pe
```

```
type(c_devptr) function nvshmem_ptrc(dest, pe)
complex :: dest ! Any kind and rank
integer(4) :: pe
```

```
type(c_devptr) function nvshmem_ptrc1(dest, pe)
character :: dest ! Any kind and rank
integer(4) :: pe
```

```
type(c_devptr) function nvshmem_ptrcd(dest, pe)
type(c_devptr) :: dest
integer(4) :: pe
```

8.2. NVSHMEM Memory Management Functions

This section contains the Fortran interfaces to NVSHMEM functions used to manage the symmetric heap.

8.2.1. nvshmem_malloc

This function allocates a block containing the specified number of bytes from the symmetric heap. This routine is a collective operation and requires participation by all PEs.

```
type(c_devptr) function nvshmem_malloc(size)
integer(8) :: size  ! Size is in bytes
```

Entities of type(c_devptr) can be cast as Fortran arrays in a few ways. Here are some examples:

```
use nvshmem
! Contiguous will avoid some runtime checks
real(8), device, pointer, contiguous :: array(:)
. . .
call c_f_pointer(nvshmem_malloc(N*8), array, [N])
```

```
use nvshmem
! Cray Pointer
real(8), device :: array(N); pointer(pa,array)
. . .
pa = transfer(nvshmem_malloc(N*8), pa)
```

8.2.2. nvshmem_free

This subroutine frees a block of symmetric data which was previously allocated.

```
subroutine nvshmem_free(ptr)
ptr can be of type(c_devptr), or other types if it was cast to a Fortran array
 using the techniques described in the nvshmem_malloc section.
```

8.2.3. nvshmem_align

This function allocates a block from the symmetric heap that has a byte alignment specified by the alignment argument.

```
type(c_devptr) function nvshmem_align(alignment, size)
integer(8) :: alignment
integer(8) :: size  ! Size is in bytes
```

8.2.4. nvshmem_calloc

This function allocates a block containing the specified number of bytes from the symmetric heap. This routine is a collective operation and requires participation by all PEs. The space is also initialized to zero.

```
type(c_devptr) function nvshmem_calloc(size)
integer(8) :: size  ! Size is in bytes
```

8.3. NVSHMEM Remote Memory Access Functions

This section contains the Fortran interfaces to NVSHMEM functions used to perform reads and writes to symmetric data objects. The CUDA C library contains a number of functions for each C type. We have tried to distill those down into a useful, but non-redundant set for Fortran programmers. In addition, we have provided the following generic interfaces which are overloaded to take multiple types:

- nvshmem_put
- nvshmem_p
- nvshmem_iput
- nvshmem_put_nbi
- nvshmemx_put_block
- nvshmemx_put_warp
- nvshmem_get

- nvshmem_g
- nvshmem_iget
- nvshmem_get_nbi
- nvshmemx_get_block
- nvshmemx_get_warp

Many of these functions are available on both the host and device. Certain programming models may not currently support generic functions on the device. Some of the functions are only available on the device (most notably, those performed by a whole block or whole warp).

8.3.1. nvshmem_put

This subroutine returns after the data has been copied out of the source array on the local PE. The subroutine **`nvshmem_put`** is overloaded to take a number of different sets of arguments. The specific names and argument lists are below.

```
subroutine nvshmem_putmem(dest, source, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_putmem_on_stream(dest, source, nelems, pe, stream)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_int8_put(dest, source, nelems, pe)
integer(1), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_int8_put_on_stream(dest, source, nelems, pe, stream)
integer(1), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_int16_put(dest, source, nelems, pe)
integer(2), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_int16_put_on_stream(dest, source, nelems, pe, stream)
integer(2), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_int32_put(dest, source, nelems, pe)
integer(4), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_int32_put_on_stream(dest, source, nelems, pe, stream)
integer(4), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_int64_put(dest, source, nelems, pe)
integer(8), device :: dest(*), source(*)
integer(8) :: nelems
```

```
integer(4) :: pe
```

```
subroutine nvshmemx_int64_put_on_stream(dest, source, nelems, pe, stream)
integer(8), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_float_put(dest, source, nelems, pe)
real(4), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_float_put_on_stream(dest, source, nelems, pe, stream)
real(4), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_double_put(dest, source, nelems, pe)
real(8), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_double_put_on_stream(dest, source, nelems, pe, stream)
real(8), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_complex_put(dest, source, nelems, pe)
complex(4), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_complex_put_on_stream(dest, source, nelems, pe, stream)
complex(4), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_dcomplex_put(dest, source, nelems, pe)
complex(8), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_dcomplex_put_on_stream(dest, source, nelems, pe, stream)
complex(8), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

The following nvshmem put subroutines are not part of the generic nvshmem_put group, but are provided for flexibility and for compatibility with the C names:

```
subroutine nvshmem_put8(dest, source, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_put8_on_stream(dest, source, nelems, pe, stream)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_put16(dest, source, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: nelems
```

```
integer(4) :: pe
```

```
subroutine nvshmemx_put16_on_stream(dest, source, nelems, pe, stream)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_put32(dest, source, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_put32_on_stream(dest, source, nelems, pe, stream)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_put64(dest, source, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_put64_on_stream(dest, source, nelems, pe, stream)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_put128(dest, source, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_put128_on_stream(dest, source, nelems, pe, stream)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

8.3.2. nvshmem_p

This subroutine returns after the data has been copied out of the source array on the local PE. The subroutine **`nvshmem_p`** is overloaded to take a number of different sets of arguments. These subroutines can be called from either the host or device, and the source is passed by value and should be host-resident or device-resident, respectively. The specific names and argument lists are below.

```
subroutine nvshmem_int8_p(dest, source, nelems, pe)
integer(1), device :: dest(*), source
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_int8_p_on_stream(dest, source, nelems, pe, stream)
integer(1), device :: dest(*), source
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_int16_p(dest, source, nelems, pe)
integer(2), device :: dest(*), source
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_int16_p_on_stream(dest, source, nelems, pe, stream)
integer(2), device :: dest(*), source
```

```
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_int32_p(dest, source, nelems, pe)
integer(4), device :: dest(*), source
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_int32_p_on_stream(dest, source, nelems, pe, stream)
integer(4), device :: dest(*), source
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_int64_p(dest, source, nelems, pe)
integer(8), device :: dest(*), source
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_int64_p_on_stream(dest, source, nelems, pe, stream)
integer(8), device :: dest(*), source
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_float_p(dest, source, nelems, pe)
real(4), device :: dest(*), source
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_float_p_on_stream(dest, source, nelems, pe, stream)
real(4), device :: dest(*), source
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_double_p(dest, source, nelems, pe)
real(8), device :: dest(*), source
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_double_p_on_stream(dest, source, nelems, pe, stream)
real(8), device :: dest(*), source
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

8.3.3. nvshmem_iput

This subroutine provides a way to copy strided data elements to a destination. This subroutine returns after the data has been copied out of the source array on the local PE. The subroutine **`nvshmem_iput`** is overloaded to take a number of different sets of arguments. These subroutines can be called from either the host or device. The specific names and argument lists are below.

```
subroutine nvshmem_int8_iput(dest, source, dst, sst, nelems, pe)
integer(1), device :: dest(*), source(*)
integer(8) :: dst, sst, nelems
integer(4) :: pe
```

```
subroutine nvshmemx_int8_iput_on_stream(dest, source, dst, sst, nelems, pe,
 stream)
integer(1), device :: dest(*), source(*)
integer(8) :: dst, sst, nelems
integer(4) :: pe
```

```
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_int16_iput(dest, source, dst, sst, nelems, pe)
integer(2), device :: dest(*), source(*)
integer(8) :: dst, sst, nelems
integer(4) :: pe
```

```
subroutine nvshmemx_int16_iput_on_stream(dest, source, dst, sst, nelems, pe,
 stream)
integer(2), device :: dest(*), source(*)
integer(8) :: dst, sst, nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_int32_iput(dest, source, dst, sst, nelems, pe)
integer(4), device :: dest(*), source(*)
integer(8) :: dst, sst, nelems
integer(4) :: pe
```

```
subroutine nvshmemx_int32_iput_on_stream(dest, source, dst, sst, nelems, pe,
 stream)
integer(4), device :: dest(*), source(*)
integer(8) :: dst, sst, nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_int64_iput(dest, source, dst, sst, nelems, pe)
integer(8), device :: dest(*), source(*)
integer(8) :: dst, sst, nelems
integer(4) :: pe
```

```
subroutine nvshmemx_int64_iput_on_stream(dest, source, dst, sst, nelems, pe,
 stream)
integer(8), device :: dest(*), source(*)
integer(8) :: dst, sst, nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_float_iput(dest, source, dst, sst, nelems, pe)
real(4), device :: dest(*), source(*)
integer(8) :: dst, sst, nelems
integer(4) :: pe
```

```
subroutine nvshmemx_float_iput_on_stream(dest, source, dst, sst, nelems, pe,
 stream)
real(4), device :: dest(*), source(*)
integer(8) :: dst, sst, nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_double_iput(dest, source, dst, sst, nelems, pe)
real(8), device :: dest(*), source(*)
integer(8) :: dst, sst, nelems
integer(4) :: pe
```

```
subroutine nvshmemx_double_iput_on_stream(dest, source, dst, sst, nelems, pe,
 stream)
real(8), device :: dest(*), source(*)
integer(8) :: dst, sst, nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_complex_iput(dest, source, dst, sst, nelems, pe)
complex(4), device :: dest(*), source(*)
integer(8) :: dst, sst, nelems
integer(4) :: pe
```

```
subroutine nvshmemx_complex_iput_on_stream(dest, source, dst, sst, nelems, pe,
 stream)
complex(4), device :: dest(*), source(*)
```

```
integer(8) :: dst, sst, nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_dcomplex_iput(dest, source, dst, sst, nelems, pe)
complex(8), device :: dest(*), source(*)
integer(8) :: dst, sst, nelems
integer(4) :: pe
```

```
subroutine nvshmemx_dcomplex_iput_on_stream(dest, source, dst, sst, nelems, pe,
 stream)
complex(8), device :: dest(*), source(*)
integer(8) :: dst, sst, nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

The following nvshmem iput subroutines are not part of the generic nvshmem_iput group, but are provided for flexibility and for compatibility with the C names:

```
subroutine nvshmem_iput8(dest, source, dst, sst, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: dst, sst, nelems
integer(4) :: pe
```

```
subroutine nvshmemx_iput8_on_stream(dest, source, dst, sst, nelems, pe, stream)
type(c_devptr) :: dest, source
integer(8) :: dst, sst, nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_iput16(dest, source, dst, sst, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: dst, sst, nelems
integer(4) :: pe
```

```
subroutine nvshmemx_iput16_on_stream(dest, source, dst, sst, nelems, pe, stream)
type(c_devptr) :: dest, source
integer(8) :: dst, sst, nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_iput32(dest, source, dst, sst, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: dst, sst, nelems
integer(4) :: pe
```

```
subroutine nvshmemx_iput32_on_stream(dest, source, dst, sst, nelems, pe, stream)
type(c_devptr) :: dest, source
integer(8) :: dst, sst, nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_iput64(dest, source, dst, sst, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: dst, sst, nelems
integer(4) :: pe
```

```
subroutine nvshmemx_iput64_on_stream(dest, source, dst, sst, nelems, pe, stream)
type(c_devptr) :: dest, source
integer(8) :: dst, sst, nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_iput128(dest, source, dst, sst, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: dst, sst, nelems
integer(4) :: pe
```

```
subroutine nvshmemx_iput128_on_stream(dest, source, dst, sst, nelems, pe,
 stream)
```

```
type(c_devptr) :: dest, source
integer(8) :: dst, sst, nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

8.3.4. nvshmem_put_nbi

This subroutine returns after initiating the put operation. The subroutine **`nvshmem_put_nbi`** is overloaded to take a number of different sets of arguments. These subroutines can be called from either the host or device. The specific names and argument lists are below.

```
subroutine nvshmem_int8_put_nbi(dest, source, nelems, pe)
integer(1), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_int8_put_nbi_on_stream(dest, source, nelems, pe, stream)
integer(1), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_int16_put_nbi(dest, source, nelems, pe)
integer(2), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_int16_put_nbi_on_stream(dest, source, nelems, pe, stream)
integer(2), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_int32_put_nbi(dest, source, nelems, pe)
integer(4), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_int32_put_nbi_on_stream(dest, source, nelems, pe, stream)
integer(4), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_int64_put_nbi(dest, source, nelems, pe)
integer(8), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_int64_put_nbi_on_stream(dest, source, nelems, pe, stream)
integer(8), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_float_put_nbi(dest, source, nelems, pe)
real(4), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_float_put_nbi_on_stream(dest, source, nelems, pe, stream)
real(4), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_double_put_nbi(dest, source, nelems, pe)
```

```
real(8), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_double_put_nbi_on_stream(dest, source, nelems, pe, stream)
real(8), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_complex_put_nbi(dest, source, nelems, pe)
complex(4), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_complex_put_nbi_on_stream(dest, source, nelems, pe, stream)
complex(4), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_dcomplex_put_nbi(dest, source, nelems, pe)
complex(8), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_dcomplex_put_nbi_on_stream(dest, source, nelems, pe, stream)
complex(8), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

The following nvshmem put_nbi subroutines are not part of the generic nvshmem_put_nbi group, but are provided for flexibility and for compatibility with the C names:

```
subroutine nvshmem_put8_nbi(dest, source, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_put8_nbi_on_stream(dest, source, nelems, pe, stream)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_put16_nbi(dest, source, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_put16_nbi_on_stream(dest, source, nelems, pe, stream)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_put32_nbi(dest, source, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_put32_nbi_on_stream(dest, source, nelems, pe, stream)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_put64_nbi(dest, source, nelems, pe)
```

```
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_put64_nbi_on_stream(dest, source, nelems, pe, stream)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_put128_nbi(dest, source, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_put128_nbi_on_stream(dest, source, nelems, pe, stream)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

8.3.5. nvshmemx_put_block

This subroutine returns after the data has been copied out of the source array on the local PE. It is only available from device code. The subroutine **`nvshmemx_put_block`** is overloaded to take a number of different sets of arguments. The specific names and argument lists are below.

```
subroutine nvshmemx_putmem_block(dest, source, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_int8_put_block(dest, source, nelems, pe)
integer(1), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_int16_put_block(dest, source, nelems, pe)
integer(2), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_int32_put_block(dest, source, nelems, pe)
integer(4), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_int64_put_block(dest, source, nelems, pe)
integer(8), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_fp16_put_block(dest, source, nelems, pe)
real(2), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_float_put_block(dest, source, nelems, pe)
real(4), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_double_put_block(dest, source, nelems, pe)
real(8), device :: dest(*), source(*)
integer(8) :: nelems
```

```
integer(4) :: pe
```

```
subroutine nvshmemx_complex_put_block(dest, source, nelems, pe)
complex(4), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_dcomplex_put_block(dest, source, nelems, pe)
complex(8), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

The following nvshmem put block subroutines are not part of the generic nvshmemx_put_block group, but are provided for flexibility and for compatibility with the C names:

```
subroutine nvshmemx_int_put_block(dest, source, nelems, pe)
integer(4), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_long_put_block(dest, source, nelems, pe)
integer(8), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_put8_block(dest, source, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_put16_block(dest, source, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_put32_block(dest, source, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_put64_block(dest, source, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_put128_block(dest, source, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
```

8.3.6. nvshmemx_put_warp

This subroutine returns after the data has been copied out of the source array on the local PE. It is only available from device code. The subroutine **`nvshmemx_put_warp`** is overloaded to take a number of different sets of arguments. The specific names and argument lists are below.

```
subroutine nvshmemx_putmem_warp(dest, source, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_int8_put_warp(dest, source, nelems, pe)
integer(1), device :: dest(*), source(*)
integer(8) :: nelems
```

```
integer(4) :: pe
```

```
subroutine nvshmemx_int16_put_warp(dest, source, nelems, pe)
integer(2), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_int32_put_warp(dest, source, nelems, pe)
integer(4), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_int64_put_warp(dest, source, nelems, pe)
integer(8), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_fp16_put_warp(dest, source, nelems, pe)
real(2), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_float_put_warp(dest, source, nelems, pe)
real(4), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_double_put_warp(dest, source, nelems, pe)
real(8), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_complex_put_warp(dest, source, nelems, pe)
complex(4), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_dcomplex_put_warp(dest, source, nelems, pe)
complex(8), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

The following nvshmem put warp subroutines are not part of the generic nvshmemx_put_warp group, but are provided for flexibility and for compatibility with the C names:

```
subroutine nvshmemx_int_put_warp(dest, source, nelems, pe)
integer(4), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_long_put_warp(dest, source, nelems, pe)
integer(8), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_put8_warp(dest, source, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_put16_warp(dest, source, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_put32_warp(dest, source, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: nelems
```

```
integer(4) :: pe
```

```
subroutine nvshmemx_put64_warp(dest, source, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_put128_warp(dest, source, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
```

8.3.7. nvshmem_get

This subroutine returns after the data has been copied out of the source array on the local PE. The subroutine **`nvshmem_get`** is overloaded to take a number of different sets of arguments. The specific names and argument lists are below.

```
subroutine nvshmem_getmem(dest, source, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_getmem_on_stream(dest, source, nelems, pe, stream)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_int8_get(dest, source, nelems, pe)
integer(1), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_int8_get_on_stream(dest, source, nelems, pe, stream)
integer(1), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_int16_get(dest, source, nelems, pe)
integer(2), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_int16_get_on_stream(dest, source, nelems, pe, stream)
integer(2), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_int32_get(dest, source, nelems, pe)
integer(4), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_int32_get_on_stream(dest, source, nelems, pe, stream)
integer(4), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_int64_get(dest, source, nelems, pe)
integer(8), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_int64_get_on_stream(dest, source, nelems, pe, stream)
integer(8), device :: dest(*), source(*)
```

```
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_float_get(dest, source, nelems, pe)
real(4), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_float_get_on_stream(dest, source, nelems, pe, stream)
real(4), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_double_get(dest, source, nelems, pe)
real(8), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_double_get_on_stream(dest, source, nelems, pe, stream)
real(8), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_complex_get(dest, source, nelems, pe)
complex(4), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_complex_get_on_stream(dest, source, nelems, pe, stream)
complex(4), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_dcomplex_get(dest, source, nelems, pe)
complex(8), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_dcomplex_get_on_stream(dest, source, nelems, pe, stream)
complex(8), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

The following nvshmem get subroutines are not part of the generic nvshmem_get group, but are provided for flexibility and for compatibility with the C names:

```
subroutine nvshmem_get8(dest, source, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_get8_on_stream(dest, source, nelems, pe, stream)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_get16(dest, source, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_get16_on_stream(dest, source, nelems, pe, stream)
type(c_devptr) :: dest, source
integer(8) :: nelems
```

```
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_get32(dest, source, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_get32_on_stream(dest, source, nelems, pe, stream)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_get64(dest, source, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_get64_on_stream(dest, source, nelems, pe, stream)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_get128(dest, source, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_get128_on_stream(dest, source, nelems, pe, stream)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

8.3.8. nvshmem_g

This subroutine returns after the data has been copied out of the source array on the local PE. The subroutine **`nvshmem_g`** is overloaded to take a number of different sets of arguments. These subroutines can be called from either the host or device, and the source is passed by value and should be host-resident or device-resident, respectively. The specific names and argument lists are below.

```
subroutine nvshmem_int8_g(dest, source, nelems, pe)
integer(1), device :: dest(*), source
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_int8_g_on_stream(dest, source, nelems, pe, stream)
integer(1), device :: dest(*), source
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_int16_g(dest, source, nelems, pe)
integer(2), device :: dest(*), source
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_int16_g_on_stream(dest, source, nelems, pe, stream)
integer(2), device :: dest(*), source
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_int32_g(dest, source, nelems, pe)
```

```
integer(4), device :: dest(*), source
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_int32_g_on_stream(dest, source, nelems, pe, stream)
integer(4), device :: dest(*), source
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_int64_g(dest, source, nelems, pe)
integer(8), device :: dest(*), source
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_int64_g_on_stream(dest, source, nelems, pe, stream)
integer(8), device :: dest(*), source
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_float_g(dest, source, nelems, pe)
real(4), device :: dest(*), source
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_float_g_on_stream(dest, source, nelems, pe, stream)
real(4), device :: dest(*), source
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_double_g(dest, source, nelems, pe)
real(8), device :: dest(*), source
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_double_g_on_stream(dest, source, nelems, pe, stream)
real(8), device :: dest(*), source
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

8.3.9. nvshmem_iget

This subroutine provides a way to copy strided data elements to a destination. This subroutine returns after the data has been copied out of the source array on the local PE. The subroutine **`nvshmem_iget`** is overloaded to take a number of different sets of arguments. These subroutines can be called from either the host or device. The specific names and argument lists are below.

```
subroutine nvshmem_int8_iget(dest, source, dst, sst, nelems, pe)
integer(1), device :: dest(*), source(*)
integer(8) :: dst, sst, nelems
integer(4) :: pe
```

```
subroutine nvshmemx_int8_iget_on_stream(dest, source, dst, sst, nelems, pe,
 stream)
integer(1), device :: dest(*), source(*)
integer(8) :: dst, sst, nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_int16_iget(dest, source, dst, sst, nelems, pe)
integer(2), device :: dest(*), source(*)
integer(8) :: dst, sst, nelems
```

```
integer(4) :: pe
```

```
subroutine nvshmemx_int16_iget_on_stream(dest, source, dst, sst, nelems, pe,
 stream)
integer(2), device :: dest(*), source(*)
integer(8) :: dst, sst, nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_int32_iget(dest, source, dst, sst, nelems, pe)
integer(4), device :: dest(*), source(*)
integer(8) :: dst, sst, nelems
integer(4) :: pe
```

```
subroutine nvshmemx_int32_iget_on_stream(dest, source, dst, sst, nelems, pe,
 stream)
integer(4), device :: dest(*), source(*)
integer(8) :: dst, sst, nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_int64_iget(dest, source, dst, sst, nelems, pe)
integer(8), device :: dest(*), source(*)
integer(8) :: dst, sst, nelems
integer(4) :: pe
```

```
subroutine nvshmemx_int64_iget_on_stream(dest, source, dst, sst, nelems, pe,
 stream)
integer(8), device :: dest(*), source(*)
integer(8) :: dst, sst, nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_float_iget(dest, source, dst, sst, nelems, pe)
real(4), device :: dest(*), source(*)
integer(8) :: dst, sst, nelems
integer(4) :: pe
```

```
subroutine nvshmemx_float_iget_on_stream(dest, source, dst, sst, nelems, pe,
 stream)
real(4), device :: dest(*), source(*)
integer(8) :: dst, sst, nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_double_iget(dest, source, dst, sst, nelems, pe)
real(8), device :: dest(*), source(*)
integer(8) :: dst, sst, nelems
integer(4) :: pe
```

```
subroutine nvshmemx_double_iget_on_stream(dest, source, dst, sst, nelems, pe,
 stream)
real(8), device :: dest(*), source(*)
integer(8) :: dst, sst, nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_complex_iget(dest, source, dst, sst, nelems, pe)
complex(4), device :: dest(*), source(*)
integer(8) :: dst, sst, nelems
integer(4) :: pe
```

```
subroutine nvshmemx_complex_iget_on_stream(dest, source, dst, sst, nelems, pe,
 stream)
complex(4), device :: dest(*), source(*)
integer(8) :: dst, sst, nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_dcomplex_iget(dest, source, dst, sst, nelems, pe)
```

```
complex(8), device :: dest(*), source(*)
integer(8) :: dst, sst, nelems
integer(4) :: pe
```

```
subroutine nvshmemx_dcomplex_iget_on_stream(dest, source, dst, sst, nelems, pe,
 stream)
complex(8), device :: dest(*), source(*)
integer(8) :: dst, sst, nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

The following nvshmem iget subroutines are not part of the generic nvshmem_iget group, but are provided for flexibility and for compatibility with the C names:

```
subroutine nvshmem_iget8(dest, source, dst, sst, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: dst, sst, nelems
integer(4) :: pe
```

```
subroutine nvshmemx_iget8_on_stream(dest, source, dst, sst, nelems, pe, stream)
type(c_devptr) :: dest, source
integer(8) :: dst, sst, nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_iget16(dest, source, dst, sst, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: dst, sst, nelems
integer(4) :: pe
```

```
subroutine nvshmemx_iget16_on_stream(dest, source, dst, sst, nelems, pe, stream)
type(c_devptr) :: dest, source
integer(8) :: dst, sst, nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_iget32(dest, source, dst, sst, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: dst, sst, nelems
integer(4) :: pe
```

```
subroutine nvshmemx_iget32_on_stream(dest, source, dst, sst, nelems, pe, stream)
type(c_devptr) :: dest, source
integer(8) :: dst, sst, nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_iget64(dest, source, dst, sst, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: dst, sst, nelems
integer(4) :: pe
```

```
subroutine nvshmemx_iget64_on_stream(dest, source, dst, sst, nelems, pe, stream)
type(c_devptr) :: dest, source
integer(8) :: dst, sst, nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_iget128(dest, source, dst, sst, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: dst, sst, nelems
integer(4) :: pe
```

```
subroutine nvshmemx_iget128_on_stream(dest, source, dst, sst, nelems, pe,
 stream)
type(c_devptr) :: dest, source
integer(8) :: dst, sst, nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

8.3.10. nvshmem_get_nbi

This subroutine returns after initiating the get operation. The subroutine **nvshmem_get_nbi** is overloaded to take a number of different sets of arguments. These subroutines can be called from either the host or device. The specific names and argument lists are below.

```
subroutine nvshmem_int8_get_nbi(dest, source, nelems, pe)
integer(1), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_int8_get_nbi_on_stream(dest, source, nelems, pe, stream)
integer(1), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_int16_get_nbi(dest, source, nelems, pe)
integer(2), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_int16_get_nbi_on_stream(dest, source, nelems, pe, stream)
integer(2), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_int32_get_nbi(dest, source, nelems, pe)
integer(4), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_int32_get_nbi_on_stream(dest, source, nelems, pe, stream)
integer(4), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_int64_get_nbi(dest, source, nelems, pe)
integer(8), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_int64_get_nbi_on_stream(dest, source, nelems, pe, stream)
integer(8), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_float_get_nbi(dest, source, nelems, pe)
real(4), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_float_get_nbi_on_stream(dest, source, nelems, pe, stream)
real(4), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_double_get_nbi(dest, source, nelems, pe)
real(8), device :: dest(*), source(*)
integer(8) :: nelems
```

```
integer(4) :: pe
```

```
subroutine nvshmemx_double_get_nbi_on_stream(dest, source, nelems, pe, stream)
real(8), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_complex_get_nbi(dest, source, nelems, pe)
complex(4), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_complex_get_nbi_on_stream(dest, source, nelems, pe, stream)
complex(4), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_dcomplex_get_nbi(dest, source, nelems, pe)
complex(8), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_dcomplex_get_nbi_on_stream(dest, source, nelems, pe, stream)
complex(8), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

The following nvshmem get_nbi subroutines are not part of the generic nvshmem_get_nbi group, but are provided for flexibility and for compatibility with the C names:

```
subroutine nvshmem_get8_nbi(dest, source, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_get8_nbi_on_stream(dest, source, nelems, pe, stream)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_get16_nbi(dest, source, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_get16_nbi_on_stream(dest, source, nelems, pe, stream)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_get32_nbi(dest, source, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_get32_nbi_on_stream(dest, source, nelems, pe, stream)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_get64_nbi(dest, source, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: nelems
```

```
integer(4) :: pe
```

```
subroutine nvshmemx_get64_nbi_on_stream(dest, source, nelems, pe, stream)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_get128_nbi(dest, source, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_get128_nbi_on_stream(dest, source, nelems, pe, stream)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
integer(cuda_stream_kind) :: stream
```

8.3.11. nvshmemx_get_block

This subroutine returns after the data has been copied out of the source array on the local PE. It is only available from device code. The subroutine **`nvshmemx_get_block`** is overloaded to take a number of different sets of arguments. The specific names and argument lists are below.

```
subroutine nvshmemx_getmem_block(dest, source, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_int8_get_block(dest, source, nelems, pe)
integer(1), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_int16_get_block(dest, source, nelems, pe)
integer(2), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_int32_get_block(dest, source, nelems, pe)
integer(4), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_int64_get_block(dest, source, nelems, pe)
integer(8), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_fp16_get_block(dest, source, nelems, pe)
real(2), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_float_get_block(dest, source, nelems, pe)
real(4), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_double_get_block(dest, source, nelems, pe)
real(8), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_complex_get_block(dest, source, nelems, pe)
complex(4), device :: dest(*), source(*)
```

```
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_dcomplex_get_block(dest, source, nelems, pe)
complex(8), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

The following nvshmem get block subroutines are not part of the generic nvshmemx_get_block group, but are provided for flexibility and for compatibility with the C names:

```
subroutine nvshmemx_int_get_block(dest, source, nelems, pe)
integer(4), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_long_get_block(dest, source, nelems, pe)
integer(8), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_get8_block(dest, source, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_get16_block(dest, source, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_get32_block(dest, source, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_get64_block(dest, source, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_get128_block(dest, source, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
```

8.3.12. nvshmemx_get_warp

This subroutine returns after the data has been copied out of the source array on the local PE. It is only available from device code. The subroutine **`nvshmemx_get_warp`** is overloaded to take a number of different sets of arguments. The specific names and argument lists are below.

```
subroutine nvshmemx_getmem_warp(dest, source, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_int8_get_warp(dest, source, nelems, pe)
integer(1), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_int16_get_warp(dest, source, nelems, pe)
integer(2), device :: dest(*), source(*)
integer(8) :: nelems
```

```
integer(4) :: pe
```

```
subroutine nvshmemx_int32_get_warp(dest, source, nelems, pe)
integer(4), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_int64_get_warp(dest, source, nelems, pe)
integer(8), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_fp16_get_warp(dest, source, nelems, pe)
real(2), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_float_get_warp(dest, source, nelems, pe)
real(4), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_double_get_warp(dest, source, nelems, pe)
real(8), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_complex_get_warp(dest, source, nelems, pe)
complex(4), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_dcomplex_get_warp(dest, source, nelems, pe)
complex(8), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

The following nvshmem get warp subroutines are not part of the generic nvshmemx_get_warp group, but are provided for flexibility and for compatibility with the C names:

```
subroutine nvshmemx_int_get_warp(dest, source, nelems, pe)
integer(4), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_long_get_warp(dest, source, nelems, pe)
integer(8), device :: dest(*), source(*)
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_get8_warp(dest, source, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_get16_warp(dest, source, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_get32_warp(dest, source, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
```

```
subroutine nvshmemx_get64_warp(dest, source, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: nelems
```

```
integer(4) :: pe
```

```
subroutine nvshmemx_get128_warp(dest, source, nelems, pe)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe
```

8.4. NVSHMEM Collective Communication Functions

This section contains the Fortran interfaces to NVSHMEM functions that perform coordinated communication or synchronization operations within a group of PEs. The section can be further divided between barrier and sync functions, all-to-all, broadcast, and collect functions, and reductions.

8.4.1. nvshmem_barrier, nvshmem_barrier_all

These subroutines perform a collective synchronization over all (**`nvshmem_barrier_all`**) or a provided subset (**`nvshmem_barrier`**) of PEs. Ordering APIs initiated on the CPU only order communication operations that were issued from the CPU. Use cudaDeviceSynchronize() or something similar to ensure GPU operations have completed. The list of subroutine names and argument lists are below.

```
subroutine nvshmem_barrier_all()
```

```
subroutine nvshmemx_barrier_all_on_stream(stream)
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_barrier(pe_start, pe_stride, pe_size, psync)
integer(4) :: pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
```

```
subroutine nvshmemx_barrier_on_stream(pe_start, pe_stride, pe_size, psync,
 stream)
integer(4) :: pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream
```

8.4.2. nvshmem_sync, nvshmem_sync_all

These subroutines perform a collective synchronization over all (**`nvshmem_sync_all`**) or a provided subset (**`nvshmem_sync`**) of PEs. Unlike the barrier routines, these subroutines only ensure completion and visibility of previously issued memory stores and does not ensure completion of remote memory updates. The list of subroutine names and argument lists are below.

```
subroutine nvshmem_sync_all()
```

```
subroutine nvshmemx_sync_all_on_stream(stream)
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_sync(pe_start, pe_stride, pe_size, psync)
integer(4) :: pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
```

```
subroutine nvshmemx_sync_on_stream(pe_start, pe_stride, pe_size, psync, stream)
integer(4) :: pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream
```

8.4.3. nvshmem_alltoall

These subroutines perform a collective all-to-all operation, exchanging the specified number of data elements with all other PEs. The routines operate on generic 32-bit and 64-bit datatypes. The list of subroutine names and argument lists are below.

```
subroutine nvshmem_alltoall32(dest, source, nelems, pe_start, pe_stride,
 pe_size, psync)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
```

```
subroutine nvshmemx_alltoall32_on_stream(dest, source, nelems, pe_start,
 pe_stride, pe_size, psync, stream)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_alltoall64(dest, source, nelems, pe_start, pe_stride,
 pe_size, psync)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
```

```
subroutine nvshmemx_alltoall64_on_stream(dest, source, nelems, pe_start,
 pe_stride, pe_size, psync, stream)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream
```

8.4.4. nvshmem_broadcast

These subroutines perform a collective broadcast operation, sending the source data from the specified root to all of the other PEs. The routines operate on generic 32-bit and 64-bit datatypes. The list of subroutine names and argument lists are below.

```
subroutine nvshmem_broadcast32(dest, source, nelems, pe_root, pe_start,
 pe_stride, pe_size, psync)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe_root, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
```

```
subroutine nvshmemx_broadcast32_on_stream(dest, source, nelems, pe_root,
 pe_start, pe_stride, pe_size, psync, stream)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe_root, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_broadcast64(dest, source, nelems, pe_root, pe_start,
 pe_stride, pe_size, psync)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe_root, pe_start, pe_stride, pe_size
```

```
integer(8), device :: psync(*)
```

```
subroutine nvshmemx_broadcast64_on_stream(dest, source, nelems, pe_root,
 pe_start, pe_stride, pe_size, psync, stream)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe_root, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream
```

8.4.5. nvshmem_collect

These subroutines perform a collective operation to concatenate the specified number of data elements from each source array into the dest array, over the set of PEs, and in processor number order. The routines operate on generic 32-bit and 64-bit datatypes. The list of subroutine names and argument lists are below.

```
subroutine nvshmem_collect32(dest, source, nelems, pe_start, pe_stride, pe_size,
 psync)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
```

```
subroutine nvshmemx_collect32_on_stream(dest, source, nelems, pe_start,
 pe_stride, pe_size, psync, stream)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_collect64(dest, source, nelems, pe_start, pe_stride, pe_size,
 psync)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
```

```
subroutine nvshmemx_collect64_on_stream(dest, source, nelems, pe_start,
 pe_stride, pe_size, psync, stream)
type(c_devptr) :: dest, source
integer(8) :: nelems
integer(4) :: pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream
```

8.4.6. NVSHMEM Reductions

This section contains the Fortran interfaces to NVSHMEM functions that perform reductions, which are synchronization operations within a group of PEs performing a bitwise or arithmetic operation, reducing a set of values down to one.

8.4.6.1. nvshmem_and_to_all

This subroutine performs a bitwise AND reduction across a set of PEs. The subroutine **`nvshmem_and_to_all`** is overloaded to take a number of different sets of arguments. The specific names and argument lists are below.

```
subroutine nvshmem_integer4_and_to_all(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
integer(4), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
```

```
integer(8), device :: psync(*)
```

```
subroutine nvshmemx_integer4_and_to_all_on_stream(dest, source, nreduce,
 pe_start, pe_stride, pe_size, pwrk, psync, stream)
integer(4), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_integer8_and_to_all(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
integer(8), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
```

```
subroutine nvshmemx_integer8_and_to_all_on_stream(dest, source, nreduce,
 pe_start, pe_stride, pe_size, pwrk, psync, stream)
integer(8), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream
```

The following nvshmem AND reduction subroutines are not part of the generic nvshmem_and_to_all group, but are provided for flexibility and for compatibility with the C names. When an overloaded name exists, it is included as a comment above the declaration:

```
! Can be called as nvshmem_int_and_to_all
subroutine nvshmem_int4_and_to_all(dest, source, nreduce, pe_start, pe_stride,
 pe_size, pwrk, psync)
integer(4), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)

! Can also be called as nvshmem_int_and_to_all
subroutine nvshmem_intcd_and_to_all(dest, source, nreduce, pe_start, pe_stride,
 pe_size, pwrk, psync)
type(c_devptr) :: dest, source, pwrk, psync
integer(4) :: nreduce, pe_start, pe_stride, pe_size
```

```
! Can be called as nvshmemx_int_and_to_all_on_stream
subroutine nvshmemx_int4_and_to_all_on_stream(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
integer(4), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream

! Can also be called as nvshmemx_int_and_to_all_on_stream
subroutine nvshmemx_intcd_and_to_all_on_stream(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
type(c_devptr) :: dest, source, pwrk, psync
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(cuda_stream_kind) :: stream
```

```
! Can be called as nvshmem_longlong_and_to_all
subroutine nvshmem_int8_and_to_all(dest, source, nreduce, pe_start, pe_stride,
 pe_size, pwrk, psync)
integer(8), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)

! Can also be called as nvshmem_longlong_and_to_all
subroutine nvshmem_longlongcd_and_to_all(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
type(c_devptr) :: dest, source, pwrk, psync
```

```
integer(4) :: nreduce, pe_start, pe_stride, pe_size
```

```
! Can be called as nvshmemx_longlong_and_to_all_on_stream
subroutine nvshmemx_int8_and_to_all_on_stream(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync, stream)
integer(8), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream

! Can also be called as nvshmemx_longlong_and_to_all_on_stream
subroutine nvshmemx_longlongcd_and_to_all_on_stream(dest, source, nreduce,
 pe_start, pe_stride, pe_size, pwrk, psync)
type(c_devptr) :: dest, source, pwrk, psync
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(cuda_stream_kind) :: stream
```

8.4.6.2. nvshmem_or_to_all

This subroutine performs a bitwise OR reduction across a set of PEs. The subroutine **`nvshmem_or_to_all`** is overloaded to take a number of different sets of arguments. The specific names and argument lists are below.

```
subroutine nvshmem_integer4_or_to_all(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
integer(4), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
```

```
subroutine nvshmemx_integer4_or_to_all_on_stream(dest, source, nreduce,
 pe_start, pe_stride, pe_size, pwrk, psync, stream)
integer(4), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_integer8_or_to_all(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
integer(8), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
```

```
subroutine nvshmemx_integer8_or_to_all_on_stream(dest, source, nreduce,
 pe_start, pe_stride, pe_size, pwrk, psync, stream)
integer(8), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream
```

The following nvshmem OR reduction subroutines are not part of the generic nvshmem_or_to_all group, but are provided for flexibility and for compatibility with the C names. When an overloaded name exists, it is included as a comment above the declaration:

```
! Can be called as nvshmem_int_or_to_all
subroutine nvshmem_int4_or_to_all(dest, source, nreduce, pe_start, pe_stride,
 pe_size, pwrk, psync)
integer(4), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)

! Can also be called as nvshmem_int_or_to_all
subroutine nvshmem_intcd_or_to_all(dest, source, nreduce, pe_start, pe_stride,
 pe_size, pwrk, psync)
type(c_devptr) :: dest, source, pwrk, psync
```

```
integer(4) :: nreduce, pe_start, pe_stride, pe_size
```

```
! Can be called as nvshmemx_int_or_to_all_on_stream
subroutine nvshmemx_int4_or_to_all_on_stream(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
integer(4), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream

! Can also be called as nvshmemx_int_or_to_all_on_stream
subroutine nvshmemx_intcd_or_to_all_on_stream(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
type(c_devptr) :: dest, source, pwrk, psync
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(cuda_stream_kind) :: stream
```

```
! Can be called as nvshmem_longlong_or_to_all
subroutine nvshmem_int8_or_to_all(dest, source, nreduce, pe_start, pe_stride,
 pe_size, pwrk, psync)
integer(8), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)

! Can also be called as nvshmem_longlong_or_to_all
subroutine nvshmem_longlongcd_or_to_all(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
type(c_devptr) :: dest, source, pwrk, psync
integer(4) :: nreduce, pe_start, pe_stride, pe_size
```

```
! Can be called as nvshmemx_longlong_or_to_all_on_stream
subroutine nvshmemx_int8_or_to_all_on_stream(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync, stream)
integer(8), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream

! Can also be called as nvshmemx_longlong_or_to_all_on_stream
subroutine nvshmemx_longlongcd_or_to_all_on_stream(dest, source, nreduce,
 pe_start, pe_stride, pe_size, pwrk, psync)
type(c_devptr) :: dest, source, pwrk, psync
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(cuda_stream_kind) :: stream
```

8.4.6.3. nvshmem_xor_to_all

This subroutine performs a bitwise XOR reduction across a set of PEs. The subroutine **`nvshmem_xor_to_all`** is overloaded to take a number of different sets of arguments. The specific names and argument lists are below.

```
subroutine nvshmem_integer4_xor_to_all(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
integer(4), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
```

```
subroutine nvshmemx_integer4_xor_to_all_on_stream(dest, source, nreduce,
 pe_start, pe_stride, pe_size, pwrk, psync, stream)
integer(4), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_integer8_xor_to_all(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
integer(8), device :: dest(*), source(*), pwrk(*)
```

```
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
```

```
subroutine nvshmemx_integer8_xor_to_all_on_stream(dest, source, nreduce,
 pe_start, pe_stride, pe_size, pwrk, psync, stream)
integer(8), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream
```

The following nvshmem XOR reduction subroutines are not part of the generic nvshmem_xor_to_all group, but are provided for flexibility and for compatibility with the C names. When an overloaded name exists, it is included as a comment above the declaration:

```
! Can be called as nvshmem_int_xor_to_all
subroutine nvshmem_int4_xor_to_all(dest, source, nreduce, pe_start, pe_stride,
 pe_size, pwrk, psync)
integer(4), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)

! Can also be called as nvshmem_int_xor_to_all
subroutine nvshmem_intcd_xor_to_all(dest, source, nreduce, pe_start, pe_stride,
 pe_size, pwrk, psync)
type(c_devptr) :: dest, source, pwrk, psync
integer(4) :: nreduce, pe_start, pe_stride, pe_size
```

```
! Can be called as nvshmemx_int_xor_to_all_on_stream
subroutine nvshmemx_int4_xor_to_all_on_stream(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
integer(4), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream

! Can also be called as nvshmemx_int_xor_to_all_on_stream
subroutine nvshmemx_intcd_xor_to_all_on_stream(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
type(c_devptr) :: dest, source, pwrk, psync
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(cuda_stream_kind) :: stream
```

```
! Can be called as nvshmem_longlong_xor_to_all
subroutine nvshmem_int8_xor_to_all(dest, source, nreduce, pe_start, pe_stride,
 pe_size, pwrk, psync)
integer(8), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)

! Can also be called as nvshmem_longlong_xor_to_all
subroutine nvshmem_longlongcd_xor_to_all(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
type(c_devptr) :: dest, source, pwrk, psync
integer(4) :: nreduce, pe_start, pe_stride, pe_size
```

```
! Can be called as nvshmemx_longlong_xor_to_all_on_stream
subroutine nvshmemx_int8_xor_to_all_on_stream(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync, stream)
integer(8), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream

! Can also be called as nvshmemx_longlong_xor_to_all_on_stream
subroutine nvshmemx_longlongcd_xor_to_all_on_stream(dest, source, nreduce,
 pe_start, pe_stride, pe_size, pwrk, psync)
type(c_devptr) :: dest, source, pwrk, psync
```

```
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(cuda_stream_kind) :: stream
```

8.4.6.4. nvshmem_max_to_all

This subroutine performs a maximum value, MAX, reduction across a set of PEs. The subroutine **`nvshmem_max_to_all`** is overloaded to take a number of different sets of arguments. The specific names and argument lists are below.

```
subroutine nvshmem_integer4_max_to_all(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
integer(4), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
```

```
subroutine nvshmemx_integer4_max_to_all_on_stream(dest, source, nreduce,
 pe_start, pe_stride, pe_size, pwrk, psync, stream)
integer(4), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_integer8_max_to_all(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
integer(8), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
```

```
subroutine nvshmemx_integer8_max_to_all_on_stream(dest, source, nreduce,
 pe_start, pe_stride, pe_size, pwrk, psync, stream)
integer(8), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_real4_max_to_all(dest, source, nreduce, pe_start, pe_stride,
 pe_size, pwrk, psync)
real(4), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
```

```
subroutine nvshmemx_real4_max_to_all_on_stream(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync, stream)
real(4), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_real8_max_to_all(dest, source, nreduce, pe_start, pe_stride,
 pe_size, pwrk, psync)
real(8), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
```

```
subroutine nvshmemx_real8_max_to_all_on_stream(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync, stream)
real(8), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream
```

The following nvshmem MAX reduction subroutines are not part of the generic nvshmem_max_to_all group, but are provided for flexibility and for compatibility with the C names. When an overloaded name exists, it is included as a comment above the declaration:

```
! Can be called as nvshmem_int_max_to_all
```

```
subroutine nvshmem_int4_max_to_all(dest, source, nreduce, pe_start, pe_stride,
 pe_size, pwrk, psync)
integer(4), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)

! Can also be called as nvshmem_int_max_to_all
subroutine nvshmem_intcd_max_to_all(dest, source, nreduce, pe_start, pe_stride,
 pe_size, pwrk, psync)
type(c_devptr) :: dest, source, pwrk, psync
integer(4) :: nreduce, pe_start, pe_stride, pe_size
```

```
! Can be called as nvshmemx_int_max_to_all_on_stream
subroutine nvshmemx_int4_max_to_all_on_stream(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
integer(4), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream

! Can also be called as nvshmemx_int_max_to_all_on_stream
subroutine nvshmemx_intcd_max_to_all_on_stream(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
type(c_devptr) :: dest, source, pwrk, psync
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(cuda_stream_kind) :: stream
```

```
! Can be called as nvshmem_longlong_max_to_all
subroutine nvshmem_int8_max_to_all(dest, source, nreduce, pe_start, pe_stride,
 pe_size, pwrk, psync)
integer(8), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)

! Can also be called as nvshmem_longlong_max_to_all
subroutine nvshmem_longlongcd_max_to_all(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
type(c_devptr) :: dest, source, pwrk, psync
integer(4) :: nreduce, pe_start, pe_stride, pe_size
```

```
! Can be called as nvshmemx_longlong_max_to_all_on_stream
subroutine nvshmemx_int8_max_to_all_on_stream(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync, stream)
integer(8), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream

! Can also be called as nvshmemx_longlong_max_to_all_on_stream
subroutine nvshmemx_longlongcd_max_to_all_on_stream(dest, source, nreduce,
 pe_start, pe_stride, pe_size, pwrk, psync)
type(c_devptr) :: dest, source, pwrk, psync
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(cuda_stream_kind) :: stream
```

```
! Can be called as nvshmem_float_max_to_all
subroutine nvshmem_float_max_to_all(dest, source, nreduce, pe_start, pe_stride,
 pe_size, pwrk, psync)
real(4), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)

! Can also be called as nvshmem_float_max_to_all
subroutine nvshmem_floatcd_max_to_all(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
type(c_devptr) :: dest, source, pwrk, psync
integer(4) :: nreduce, pe_start, pe_stride, pe_size
```

```
! Can be called as nvshmemx_float_max_to_all_on_stream
```

```
subroutine nvshmemx_float_max_to_all_on_stream(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
real(4), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream

! Can also be called as nvshmemx_float_max_to_all_on_stream
subroutine nvshmemx_floatcd_max_to_all_on_stream(dest, source, nreduce,
 pe_start, pe_stride, pe_size, pwrk, psync)
type(c_devptr) :: dest, source, pwrk, psync
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(cuda_stream_kind) :: stream
```

```
! Can be called as nvshmem_double_max_to_all
subroutine nvshmem_double_max_to_all(dest, source, nreduce, pe_start, pe_stride,
 pe_size, pwrk, psync)
real(8), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)

! Can also be called as nvshmem_double_max_to_all
subroutine nvshmem_doublecd_max_to_all(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
type(c_devptr) :: dest, source, pwrk, psync
integer(4) :: nreduce, pe_start, pe_stride, pe_size
```

```
! Can be called as nvshmemx_double_max_to_all_on_stream
subroutine nvshmemx_double_max_to_all_on_stream(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
real(8), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream

! Can also be called as nvshmemx_double_max_to_all_on_stream
subroutine nvshmemx_doublecd_max_to_all_on_stream(dest, source, nreduce,
 pe_start, pe_stride, pe_size, pwrk, psync)
type(c_devptr) :: dest, source, pwrk, psync
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(cuda_stream_kind) :: stream
```

8.4.6.5. nvshmem_min_to_all

This subroutine performs a minimum value, MIN, reduction across a set of PEs. The subroutine **`nvshmem_min_to_all`** is overloaded to take a number of different sets of arguments. The specific names and argument lists are below.

```
subroutine nvshmem_integer4_min_to_all(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
integer(4), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
```

```
subroutine nvshmemx_integer4_min_to_all_on_stream(dest, source, nreduce,
 pe_start, pe_stride, pe_size, pwrk, psync, stream)
integer(4), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_integer8_min_to_all(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
integer(8), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
```

```
integer(8), device :: psync(*)
```

```
subroutine nvshmemx_integer8_min_to_all_on_stream(dest, source, nreduce,
 pe_start, pe_stride, pe_size, pwrk, psync, stream)
integer(8), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_real4_min_to_all(dest, source, nreduce, pe_start, pe_stride,
 pe_size, pwrk, psync)
real(4), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
```

```
subroutine nvshmemx_real4_min_to_all_on_stream(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync, stream)
real(4), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_real8_min_to_all(dest, source, nreduce, pe_start, pe_stride,
 pe_size, pwrk, psync)
real(8), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
```

```
subroutine nvshmemx_real8_min_to_all_on_stream(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync, stream)
real(8), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream
```

The following nvshmem MIN reduction subroutines are not part of the generic nvshmem_min_to_all group, but are provided for flexibility and for compatibility with the C names. When an overloaded name exists, it is included as a comment above the declaration:

```
! Can be called as nvshmem_int_min_to_all
subroutine nvshmem_int4_min_to_all(dest, source, nreduce, pe_start, pe_stride,
 pe_size, pwrk, psync)
integer(4), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)

! Can also be called as nvshmem_int_min_to_all
subroutine nvshmem_intcd_min_to_all(dest, source, nreduce, pe_start, pe_stride,
 pe_size, pwrk, psync)
type(c_devptr) :: dest, source, pwrk, psync
integer(4) :: nreduce, pe_start, pe_stride, pe_size
```

```
! Can be called as nvshmemx_int_min_to_all_on_stream
subroutine nvshmemx_int4_min_to_all_on_stream(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
integer(4), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream

! Can also be called as nvshmemx_int_min_to_all_on_stream
subroutine nvshmemx_intcd_min_to_all_on_stream(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
type(c_devptr) :: dest, source, pwrk, psync
integer(4) :: nreduce, pe_start, pe_stride, pe_size
```

```
integer(cuda_stream_kind) :: stream
```

```
! Can be called as nvshmem_longlong_min_to_all
subroutine nvshmem_int8_min_to_all(dest, source, nreduce, pe_start, pe_stride,
 pe_size, pwrk, psync)
integer(8), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)

! Can also be called as nvshmem_longlong_min_to_all
subroutine nvshmem_longlongcd_min_to_all(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
type(c_devptr) :: dest, source, pwrk, psync
integer(4) :: nreduce, pe_start, pe_stride, pe_size
```

```
! Can be called as nvshmemx_longlong_min_to_all_on_stream
subroutine nvshmemx_int8_min_to_all_on_stream(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync, stream)
integer(8), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream

! Can also be called as nvshmemx_longlong_min_to_all_on_stream
subroutine nvshmemx_longlongcd_min_to_all_on_stream(dest, source, nreduce,
 pe_start, pe_stride, pe_size, pwrk, psync)
type(c_devptr) :: dest, source, pwrk, psync
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(cuda_stream_kind) :: stream
```

```
! Can be called as nvshmem_float_min_to_all
subroutine nvshmem_float_min_to_all(dest, source, nreduce, pe_start, pe_stride,
 pe_size, pwrk, psync)
real(4), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)

! Can also be called as nvshmem_float_min_to_all
subroutine nvshmem_floatcd_min_to_all(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
type(c_devptr) :: dest, source, pwrk, psync
integer(4) :: nreduce, pe_start, pe_stride, pe_size
```

```
! Can be called as nvshmemx_float_min_to_all_on_stream
subroutine nvshmemx_float_min_to_all_on_stream(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
real(4), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream

! Can also be called as nvshmemx_float_min_to_all_on_stream
subroutine nvshmemx_floatcd_min_to_all_on_stream(dest, source, nreduce,
 pe_start, pe_stride, pe_size, pwrk, psync)
type(c_devptr) :: dest, source, pwrk, psync
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(cuda_stream_kind) :: stream
```

```
! Can be called as nvshmem_double_min_to_all
subroutine nvshmem_double_min_to_all(dest, source, nreduce, pe_start, pe_stride,
 pe_size, pwrk, psync)
real(8), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)

! Can also be called as nvshmem_double_min_to_all
subroutine nvshmem_doublecd_min_to_all(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
type(c_devptr) :: dest, source, pwrk, psync
```

```
integer(4) :: nreduce, pe_start, pe_stride, pe_size
```

```
! Can be called as nvshmemx_double_min_to_all_on_stream
subroutine nvshmemx_double_min_to_all_on_stream(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
real(8), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream

! Can also be called as nvshmemx_double_min_to_all_on_stream
subroutine nvshmemx_doublecd_min_to_all_on_stream(dest, source, nreduce,
 pe_start, pe_stride, pe_size, pwrk, psync)
type(c_devptr) :: dest, source, pwrk, psync
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(cuda_stream_kind) :: stream
```

8.4.6.6. nvshmem_sum_to_all

This subroutine performs a summation, or SUM, reduction across a set of PEs. The subroutine **`nvshmem_sum_to_all`** is overloaded to take a number of different sets of arguments. The specific names and argument lists are below.

```
subroutine nvshmem_integer4_sum_to_all(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
integer(4), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
```

```
subroutine nvshmemx_integer4_sum_to_all_on_stream(dest, source, nreduce,
 pe_start, pe_stride, pe_size, pwrk, psync, stream)
integer(4), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_integer8_sum_to_all(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
integer(8), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
```

```
subroutine nvshmemx_integer8_sum_to_all_on_stream(dest, source, nreduce,
 pe_start, pe_stride, pe_size, pwrk, psync, stream)
integer(8), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_real4_sum_to_all(dest, source, nreduce, pe_start, pe_stride,
 pe_size, pwrk, psync)
real(4), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
```

```
subroutine nvshmemx_real4_sum_to_all_on_stream(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync, stream)
real(4), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_real8_sum_to_all(dest, source, nreduce, pe_start, pe_stride,
 pe_size, pwrk, psync)
real(8), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
```

```
integer(8), device :: psync(*)
```

```
subroutine nvshmemx_real8_sum_to_all_on_stream(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync, stream)
real(8), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_complex4_sum_to_all(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
complex(4), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
```

```
subroutine nvshmemx_complex4_sum_to_all_on_stream(dest, source, nreduce,
 pe_start, pe_stride, pe_size, pwrk, psync, stream)
complex(8), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_complex8_sum_to_all(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
complex(4), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
```

```
subroutine nvshmemx_complex8_sum_to_all_on_stream(dest, source, nreduce,
 pe_start, pe_stride, pe_size, pwrk, psync, stream)
complex(8), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream
```

The following nvshmem SUM reduction subroutines are not part of the generic nvshmem_sum_to_all group, but are provided for flexibility and for compatibility with the C names. When an overloaded name exists, it is included as a comment above the declaration:

```
! Can be called as nvshmem_int_sum_to_all
subroutine nvshmem_int4_sum_to_all(dest, source, nreduce, pe_start, pe_stride,
 pe_size, pwrk, psync)
integer(4), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)

! Can also be called as nvshmem_int_sum_to_all
subroutine nvshmem_intcd_sum_to_all(dest, source, nreduce, pe_start, pe_stride,
 pe_size, pwrk, psync)
type(c_devptr) :: dest, source, pwrk, psync
integer(4) :: nreduce, pe_start, pe_stride, pe_size
```

```
! Can be called as nvshmemx_int_sum_to_all_on_stream
subroutine nvshmemx_int4_sum_to_all_on_stream(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
integer(4), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream

! Can also be called as nvshmemx_int_sum_to_all_on_stream
subroutine nvshmemx_intcd_sum_to_all_on_stream(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
type(c_devptr) :: dest, source, pwrk, psync
integer(4) :: nreduce, pe_start, pe_stride, pe_size
```

```
integer(cuda_stream_kind) :: stream
```

```
! Can be called as nvshmem_longlong_sum_to_all
subroutine nvshmem_int8_sum_to_all(dest, source, nreduce, pe_start, pe_stride,
 pe_size, pwrk, psync)
integer(8), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)

! Can also be called as nvshmem_longlong_sum_to_all
subroutine nvshmem_longlongcd_sum_to_all(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
type(c_devptr) :: dest, source, pwrk, psync
integer(4) :: nreduce, pe_start, pe_stride, pe_size
```

```
! Can be called as nvshmemx_longlong_sum_to_all_on_stream
subroutine nvshmemx_int8_sum_to_all_on_stream(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync, stream)
integer(8), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream

! Can also be called as nvshmemx_longlong_sum_to_all_on_stream
subroutine nvshmemx_longlongcd_sum_to_all_on_stream(dest, source, nreduce,
 pe_start, pe_stride, pe_size, pwrk, psync)
type(c_devptr) :: dest, source, pwrk, psync
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(cuda_stream_kind) :: stream
```

```
! Can be called as nvshmem_float_sum_to_all
subroutine nvshmem_float_sum_to_all(dest, source, nreduce, pe_start, pe_stride,
 pe_size, pwrk, psync)
real(4), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)

! Can also be called as nvshmem_float_sum_to_all
subroutine nvshmem_floatcd_sum_to_all(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
type(c_devptr) :: dest, source, pwrk, psync
integer(4) :: nreduce, pe_start, pe_stride, pe_size
```

```
! Can be called as nvshmemx_float_sum_to_all_on_stream
subroutine nvshmemx_float_sum_to_all_on_stream(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
real(4), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream

! Can also be called as nvshmemx_float_sum_to_all_on_stream
subroutine nvshmemx_floatcd_sum_to_all_on_stream(dest, source, nreduce,
 pe_start, pe_stride, pe_size, pwrk, psync)
type(c_devptr) :: dest, source, pwrk, psync
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(cuda_stream_kind) :: stream
```

```
! Can be called as nvshmem_double_sum_to_all
subroutine nvshmem_double_sum_to_all(dest, source, nreduce, pe_start, pe_stride,
 pe_size, pwrk, psync)
real(8), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)

! Can also be called as nvshmem_double_sum_to_all
subroutine nvshmem_doublecd_sum_to_all(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
type(c_devptr) :: dest, source, pwrk, psync
```

```
integer(4) :: nreduce, pe_start, pe_stride, pe_size
```

```
! Can be called as nvshmemx_double_sum_to_all_on_stream
subroutine nvshmemx_double_sum_to_all_on_stream(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
real(8), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream

! Can also be called as nvshmemx_double_sum_to_all_on_stream
subroutine nvshmemx_doublecd_sum_to_all_on_stream(dest, source, nreduce,
 pe_start, pe_stride, pe_size, pwrk, psync)
type(c_devptr) :: dest, source, pwrk, psync
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(cuda_stream_kind) :: stream
```

```
! Can be called as nvshmem_complexf_sum_to_all
subroutine nvshmem_complexf_sum_to_all(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
complex(4), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)

! Can also be called as nvshmem_complexf_sum_to_all
subroutine nvshmem_complexfcd_sum_to_all(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
type(c_devptr) :: dest, source, pwrk, psync
integer(4) :: nreduce, pe_start, pe_stride, pe_size
```

```
! Can be called as nvshmemx_complexf_sum_to_all_on_stream
subroutine nvshmemx_complexf_sum_to_all_on_stream(dest, source, nreduce,
 pe_start, pe_stride, pe_size, pwrk, psync)
complex(4), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream

! Can also be called as nvshmemx_complexf_sum_to_all_on_stream
subroutine nvshmemx_complexfcd_sum_to_all_on_stream(dest, source, nreduce,
 pe_start, pe_stride, pe_size, pwrk, psync)
type(c_devptr) :: dest, source, pwrk, psync
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(cuda_stream_kind) :: stream
```

```
! Can be called as nvshmem_complexd_sum_to_all
subroutine nvshmem_complexd_sum_to_all(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
complex(8), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)

! Can also be called as nvshmem_complexd_sum_to_all
subroutine nvshmem_complexdcd_sum_to_all(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
type(c_devptr) :: dest, source, pwrk, psync
integer(4) :: nreduce, pe_start, pe_stride, pe_size
```

```
! Can be called as nvshmemx_complexd_sum_to_all_on_stream
subroutine nvshmemx_complexd_sum_to_all_on_stream(dest, source, nreduce,
 pe_start, pe_stride, pe_size, pwrk, psync)
complex(8), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream

! Can also be called as nvshmemx_complexd_sum_to_all_on_stream
subroutine nvshmemx_complexdcd_sum_to_all_on_stream(dest, source, nreduce,
 pe_start, pe_stride, pe_size, pwrk, psync)
```

```
type(c_devptr) :: dest, source, pwrk, psync
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(cuda_stream_kind) :: stream
```

8.4.6.7. nvshmem_prod_to_all

This subroutine performs a product reduction across a set of PEs. The subroutine **`nvshmem_prod_to_all`** is overloaded to take a number of different sets of arguments. The specific names and argument lists are below.

```
subroutine nvshmem_integer4_prod_to_all(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
integer(4), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
```

```
subroutine nvshmemx_integer4_prod_to_all_on_stream(dest, source, nreduce,
 pe_start, pe_stride, pe_size, pwrk, psync, stream)
integer(4), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_integer8_prod_to_all(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
integer(8), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
```

```
subroutine nvshmemx_integer8_prod_to_all_on_stream(dest, source, nreduce,
 pe_start, pe_stride, pe_size, pwrk, psync, stream)
integer(8), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_real4_prod_to_all(dest, source, nreduce, pe_start, pe_stride,
 pe_size, pwrk, psync)
real(4), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
```

```
subroutine nvshmemx_real4_prod_to_all_on_stream(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync, stream)
real(4), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_real8_prod_to_all(dest, source, nreduce, pe_start, pe_stride,
 pe_size, pwrk, psync)
real(8), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
```

```
subroutine nvshmemx_real8_prod_to_all_on_stream(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync, stream)
real(8), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_complex4_prod_to_all(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
complex(4), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
```

```
integer(8), device :: psync(*)
```

```
subroutine nvshmemx_complex4_prod_to_all_on_stream(dest, source, nreduce,
 pe_start, pe_stride, pe_size, pwrk, psync, stream)
complex(8), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_complex8_prod_to_all(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
complex(4), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
```

```
subroutine nvshmemx_complex8_prod_to_all_on_stream(dest, source, nreduce,
 pe_start, pe_stride, pe_size, pwrk, psync, stream)
complex(8), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream
```

The following nvshmem product reduction subroutines are not part of the generic nvshmem_prod_to_all group, but are provided for flexibility and for compatibility with the C names. When an overloaded name exists, it is included as a comment above the declaration:

```
! Can be called as nvshmem_int_prod_to_all
subroutine nvshmem_int4_prod_to_all(dest, source, nreduce, pe_start, pe_stride,
 pe_size, pwrk, psync)
integer(4), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)

! Can also be called as nvshmem_int_prod_to_all
subroutine nvshmem_intcd_prod_to_all(dest, source, nreduce, pe_start, pe_stride,
 pe_size, pwrk, psync)
type(c_devptr) :: dest, source, pwrk, psync
integer(4) :: nreduce, pe_start, pe_stride, pe_size
```

```
! Can be called as nvshmemx_int_prod_to_all_on_stream
subroutine nvshmemx_int4_prod_to_all_on_stream(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
integer(4), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream

! Can also be called as nvshmemx_int_prod_to_all_on_stream
subroutine nvshmemx_intcd_prod_to_all_on_stream(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
type(c_devptr) :: dest, source, pwrk, psync
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(cuda_stream_kind) :: stream
```

```
! Can be called as nvshmem_longlong_prod_to_all
subroutine nvshmem_int8_prod_to_all(dest, source, nreduce, pe_start, pe_stride,
 pe_size, pwrk, psync)
integer(8), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)

! Can also be called as nvshmem_longlong_prod_to_all
subroutine nvshmem_longlongcd_prod_to_all(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
type(c_devptr) :: dest, source, pwrk, psync
```

```
integer(4) :: nreduce, pe_start, pe_stride, pe_size
```

```
! Can be called as nvshmemx_longlong_prod_to_all_on_stream
subroutine nvshmemx_int8_prod_to_all_on_stream(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync, stream)
integer(8), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream

! Can also be called as nvshmemx_longlong_prod_to_all_on_stream
subroutine nvshmemx_longlongcd_prod_to_all_on_stream(dest, source, nreduce,
 pe_start, pe_stride, pe_size, pwrk, psync)
type(c_devptr) :: dest, source, pwrk, psync
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(cuda_stream_kind) :: stream
```

```
! Can be called as nvshmem_float_prod_to_all
subroutine nvshmem_float_prod_to_all(dest, source, nreduce, pe_start, pe_stride,
 pe_size, pwrk, psync)
real(4), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)

! Can also be called as nvshmem_float_prod_to_all
subroutine nvshmem_floatcd_prod_to_all(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
type(c_devptr) :: dest, source, pwrk, psync
integer(4) :: nreduce, pe_start, pe_stride, pe_size
```

```
! Can be called as nvshmemx_float_prod_to_all_on_stream
subroutine nvshmemx_float_prod_to_all_on_stream(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
real(4), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream

! Can also be called as nvshmemx_float_prod_to_all_on_stream
subroutine nvshmemx_floatcd_prod_to_all_on_stream(dest, source, nreduce,
 pe_start, pe_stride, pe_size, pwrk, psync)
type(c_devptr) :: dest, source, pwrk, psync
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(cuda_stream_kind) :: stream
```

```
! Can be called as nvshmem_double_prod_to_all
subroutine nvshmem_double_prod_to_all(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
real(8), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)

! Can also be called as nvshmem_double_prod_to_all
subroutine nvshmem_doublecd_prod_to_all(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
type(c_devptr) :: dest, source, pwrk, psync
integer(4) :: nreduce, pe_start, pe_stride, pe_size
```

```
! Can be called as nvshmemx_double_prod_to_all_on_stream
subroutine nvshmemx_double_prod_to_all_on_stream(dest, source, nreduce,
 pe_start, pe_stride, pe_size, pwrk, psync)
real(8), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream

! Can also be called as nvshmemx_double_prod_to_all_on_stream
subroutine nvshmemx_doublecd_prod_to_all_on_stream(dest, source, nreduce,
 pe_start, pe_stride, pe_size, pwrk, psync)
```

```
type(c_devptr) :: dest, source, pwrk, psync
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(cuda_stream_kind) :: stream
```

```
! Can be called as nvshmem_complexf_prod_to_all
subroutine nvshmem_complexf_prod_to_all(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
complex(4), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)

! Can also be called as nvshmem_complexf_prod_to_all
subroutine nvshmem_complexfcd_prod_to_all(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
type(c_devptr) :: dest, source, pwrk, psync
integer(4) :: nreduce, pe_start, pe_stride, pe_size
```

```
! Can be called as nvshmemx_complexf_prod_to_all_on_stream
subroutine nvshmemx_complexf_prod_to_all_on_stream(dest, source, nreduce,
 pe_start, pe_stride, pe_size, pwrk, psync)
complex(4), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream

! Can also be called as nvshmemx_complexf_prod_to_all_on_stream
subroutine nvshmemx_complexfcd_prod_to_all_on_stream(dest, source, nreduce,
 pe_start, pe_stride, pe_size, pwrk, psync)
type(c_devptr) :: dest, source, pwrk, psync
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(cuda_stream_kind) :: stream
```

```
! Can be called as nvshmem_complexd_prod_to_all
subroutine nvshmem_complexd_prod_to_all(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
complex(8), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)

! Can also be called as nvshmem_complexd_prod_to_all
subroutine nvshmem_complexdcd_prod_to_all(dest, source, nreduce, pe_start,
 pe_stride, pe_size, pwrk, psync)
type(c_devptr) :: dest, source, pwrk, psync
integer(4) :: nreduce, pe_start, pe_stride, pe_size
```

```
! Can be called as nvshmemx_complexd_prod_to_all_on_stream
subroutine nvshmemx_complexd_prod_to_all_on_stream(dest, source, nreduce,
 pe_start, pe_stride, pe_size, pwrk, psync)
complex(8), device :: dest(*), source(*), pwrk(*)
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(8), device :: psync(*)
integer(cuda_stream_kind) :: stream

! Can also be called as nvshmemx_complexd_prod_to_all_on_stream
subroutine nvshmemx_complexdcd_prod_to_all_on_stream(dest, source, nreduce,
 pe_start, pe_stride, pe_size, pwrk, psync)
type(c_devptr) :: dest, source, pwrk, psync
integer(4) :: nreduce, pe_start, pe_stride, pe_size
integer(cuda_stream_kind) :: stream
```

8.5. NVSHMEM Point to Point Synchronization Functions

This section contains the Fortran interfaces to NVSHMEM functions that provide a mechanism for synchronization between two PEs based on a value in the symmetric memory.

8.5.1. nvshmem_wait_until

This subroutine blocks until the value contained in the symmetric data object at the calling PE satisfies the condition specified by the comparison operator and the comparison value. The subroutine **`nvshmem_wait_until`** is overloaded to take a number of different sets of arguments. The specific names and argument lists are below.

```
subroutine nvshmem_int32_wait_until(ivar, cmp, value)
integer(4), device :: ivar
integer(4) :: cmp, value
```

```
subroutine nvshmemx_int32_wait_until_on_stream(ivar, cmp, value, stream)
integer(4), device :: ivar
integer(4) :: cmp, value
integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_int64_wait_until(ivar, cmp, value)
integer(8), device :: ivar
integer(4) :: cmp
integer(8) :: value
```

```
subroutine nvshmemx_int64_wait_until_on_stream(ivar, cmp, value, stream)
integer(8), device :: ivar
integer(4) :: cmp
integer(8) :: value
integer(cuda_stream_kind) :: stream
```

8.6. NVSHMEM Memory Ordering Functions

This section contains the Fortran interfaces to NVSHMEM functions that provide mechanisms to ensure ordering and/or delivery of completion on NVSHMEM operations.

8.6.1. nvshmem_fence

This subroutine ensures the ordering of delivery of operations on symmetric data objects.

```
subroutine nvshmem_fence()
```

8.6.2. nvshmem_quiet

These subroutines ensure completion of all operations on symmetric data objects issued by the calling PE.

```
subroutine nvshmem_quiet()
```

```
subroutine nvshmemx_quiet_on_stream(stream)
integer(cuda_stream_kind) :: stream
```

Chapter 9. EXAMPLES

This section contains examples with source code.

9.1. Using cuBLAS from OpenACC Host Code

This example demonstrates the use of the cublas module, the `cublasHandle` type, and several forms of blas calls from OpenACC data regions.

Simple OpenACC BLAS Test

```
program testcublas
! compile with pgfortran -ta=tesla -Mcudalib=cublas -Mcuda testcublas.f90
call testcu1(1000)
call testcu2(1000)
end
!
subroutine testcu1(n)
use openacc
use cublas
integer :: a(n), b(n)
type(cublasHandle) :: h
istat = cublasCreate(h)
! Force OpenACC kernels and cuBLAS to use the OpenACC stream.
istat = cublasSetStream(h, acc_get_cuda_stream(acc_async_sync))
!$acc data copyout(a, b)
!$acc kernels
a = 1
b = 2
!$acc end kernels
! No host_data, we are lexically inside a data region
! sswap will accept any kind(4) data type
call sswap(n, a, 1, b, 1)
call cublasSswap(n, a, 1, b, 1)
!$acc end data
if (all(a.eq.1).and.all(b.eq.2)) then
  print *,"Test PASSED"
else
  print *,"Test FAILED"
endif
end
!
subroutine testcu2(n)
use openacc
use cublas
real(8) :: a(n), b(n)
```

```
a = 1.0d0
b = 2.0d0
!$acc data copy(a, b)
!$acc host_data use_device(a,b)
call dswap(n, a, 1, b, 1)
call cublasDswap(n, a, 1, b, 1)
!$acc end host_data
!$acc end data
if (all(a.eq.1.0d0).and.all(b.eq.2.0d0)) then
  print *,"Test PASSED"
else
  print *,"Test FAILED"
endif
end
```

CUBLASXT BLAS Test

This example demonstrates the use of the cublasxt module

```
program testcublasxt
call testxt1(1000)
call testxt2(1000)
end
!
subroutine testxt1(n)
use cublasxt
real(4) :: a(n,n), b(n,n), c(n,n), alpha, beta
type(cublasXtHandle) :: h
integer ndevices(1)
a = 1.0
b = 2.0
c = -1.0
alpha = 1.0
beta = 0.0
istat = cublasXtCreate(h)
if (istat .ne. CUBLAS_STATUS_SUCCESS) print *,istat
ndevices(1) = 0
istat = cublasXtDeviceSelect(h, 1, ndevices)
if (istat .ne. CUBLAS_STATUS_SUCCESS) print *,istat
istat = cublasXtSgemm(h, CUBLAS_OP_N, CUBLAS_OP_N, &
                      n, n, n, &
                      alpha, A, n, B, n, beta, C, n)
if (istat .ne. CUBLAS_STATUS_SUCCESS) print *,istat
istat = cublasXtDestroy(h)
if (istat .ne. CUBLAS_STATUS_SUCCESS) print *,istat
if (all(c.eq.2.0*n)) then
  print *,"Test PASSED"
else
  print *,"Test FAILED"
endif
print *,c(1,1),c(n,n)
end
!
subroutine testxt2(n)
use cublasxt
real(8) :: a(n,n), b(n,n), c(n,n), alpha, beta
type(cublasXtHandle) :: h
integer ndevices(1)
a = 1.0d0
b = 2.0d0
c = -1.0d0
alpha = 1.0d0
beta = 0.0
istat = cublasXtCreate(h)
if (istat .ne. CUBLAS_STATUS_SUCCESS) print *,istat
ndevices(1) = 0
istat = cublasXtDeviceSelect(h, 1, ndevices)
```

```
if (istat .ne. CUBLAS_STATUS_SUCCESS) print *,istat
istat = cublasXtDgemm(h, CUBLAS_OP_N, CUBLAS_OP_N, &
                      n, n, n, &
                      alpha, A, n, B, n, beta, C, n)
if (istat .ne. CUBLAS_STATUS_SUCCESS) print *,istat
istat = cublasXtDestroy(h)
if (istat .ne. CUBLAS_STATUS_SUCCESS) print *,istat
if (all(c.eq.2.0d0*n)) then
  print *,"Test PASSED"
else
  print *,"Test FAILED"
endif
print *,c(1,1),c(n,n)
end
```

9.2. Using cuBLAS from OpenACC Device Code

This example demonstrates the use of the openacc_cublas module from within an OpenACC kernel.

Simple BLAS Test from Device Code

```
module mtests
  integer, parameter :: n = 100
  contains
    subroutine testcu( a, b )
    use openacc_cublas
    real :: a(n), b(n)
    type(cublasHandle) :: h
    !$acc parallel num_gangs(1) copy(a,b,h)
    j = cublasCreate(h)
    j = cublasSswap(h,n,a,1,b,1)
    j = cublasDestroy(h)
    !$acc end parallel
    return
    end subroutine
end module mtests

program t
! compile with pgfortran -ta=tesla:cc35 -Mcuda t.f90 -lcublas_device
use mtests
real :: a(n), b(n), c(n)
logical passing
a = 1.0
b = 2.0
passing = .true.
call testcu(a,b)
print *,"Should all be 2.0"
print *,a
passing = passing .and. all(a.eq.2.0)
print *,"Should all be 1.0"
print *,b
passing = passing .and. all(b.eq.1.0)
if (passing) then
  print *,"Test PASSED"
else
  print *,"Test FAILED"
endif
end program
```

9.3. Using cuBLAS from CUDA Fortran Host Code

This example demonstrates the use of the cublas module, the `cublasHandle` type, and several forms of blas calls.

Simple BLAS Test

```
program testisamax
! Compile with "pgfortran testisamax.cuf -Mcudalib=cublas -lblas"
! Use the NVIDIA cudafor and cublas modules
use cudafor
use cublas
!
real*4, device, allocatable :: xd(:)
real*4 x(1000)
integer, device :: kd
type(cublasHandle) :: h

call random_number(x)

! F90 way
i = maxloc(x,dim=1)
print *,i
print *,x(i-1),x(i),x(i+1)

! Host way
j = isamax(1000,x,1)
print *,j
print *,x(j-1),x(j),x(j+1)

! CUDA Generic BLAS way
allocate(xd(1000))
xd = x
k = isamax(1000,xd,1)
print *,k
print *,x(k-1),x(k),x(k+1)

! CUDA Specific BLAS way
k = cublasIsamax(1000,xd,1)
print *,k
print *,x(k-1),x(k),x(k+1)

! CUDA V2 Host Specific BLAS way
istat = cublasCreate(h)
if (istat .ne. 0) print *,"cublasCreate returned ",istat
k = 0
istat = cublasIsamax_v2(h, 1000, xd, 1, k)
if (istat .ne. 0) print *,"cublasIsamax 1 returned ",istat
print *,k
print *,x(k-1),x(k),x(k+1)

! CUDA V2 Device Specific BLAS way
k = 0
istat = cublasIsamax_v2(h, 1000, xd, 1, kd)
if (istat .ne. 0) print *,"cublasIsamax 2 returned ",istat
k = kd
print *,k
print *,x(k-1),x(k),x(k+1)

istat = cublasDestroy(h)
if (istat .ne. 0) print *,"cublasDestroy returned ",istat

end program
```

Multi-threaded BLAS Test

This example demonstrates the use of the cublas module in a multi-threaded code. Each thread will attach to a different GPU, create a context, and combine the results at the end.

```
program tsgemm
!
! Multi-threaded example calling sgemm from cuda fortran.
! Compile with "pgfortran -mp tsgemm.cuf -Mcudalib=cublas"
! Set OMP_NUM_THREADS=number of GPUs in your system.
!
use cublas
use cudafor
use omp_lib
!
! Size this according to number of GPUs
!
! Small
!integer, parameter :: K = 2500
!integer, parameter :: M = 2000
!integer, parameter :: N = 2000
! Large
integer, parameter :: K = 10000
integer, parameter :: M = 10000
integer, parameter :: N = 10000
integer, parameter :: NTIMES = 10
!
real*4, device, allocatable :: a_d(:,:), b_d(:,:), c_d(:,:)
!$omp THREADPRIVATE(a_d,b_d,c_d)
real*4 a(m,k), b(k,n), c(m,n)
real*4 alpha, beta
integer, allocatable :: offs(:)
type(cudaEvent) :: start, stop

a = 1.0; b = 0.0; c = 0.0
do i = 1, N
  b(i,i) = 1.0
end do
alpha = 1.0; beta  = 1.0

! Break up the B and C array into sections
nthr = omp_get_max_threads()
nsec = N / nthr
print *,"Running with ",nthr," threads, each section = ",nsec
allocate(offs(nthr))
offs = (/ (i*nsec,i=0,nthr-1) /)

! Allocate and initialize the arrays
! Each thread connects to a device and creates a CUDA context.
!$omp PARALLEL private(i,istat)
    i = omp_get_thread_num() + 1
    istat = cudaSetDevice(i-1)
    allocate(a_d(M,K), b_d(K,nsec), c_d(M,nsec))
    a_d = a
    b_d = b(:,offs(i)+1:offs(i)+nsec)
    c_d = c(:,offs(i)+1:offs(i)+nsec)
!$omp end parallel

istat = cudaEventCreate(start)
istat = cudaEventCreate(stop)

time = 0.0
istat = cudaEventRecord(start, 0)
! Run the traditional blas kernel
!$omp PARALLEL private(j,istat)
```

```
    do j = 1, NTIMES
      call sgemm('n','n', M, N/nthr, K, alpha, a_d, M, b_d, K, beta, c_d, M)
    end do
    istat = cudaDeviceSynchronize()
!$omp end parallel

istat = cudaEventRecord(stop, 0)
istat = cudaEventElapsedTime(time, start, stop)
time = time / (NTIMES*1.0e3)
!$omp PARALLEL private(i)
    i = omp_get_thread_num() + 1
    c(:,offs(i)+1:offs(i)+nsec) = c_d
!$omp end parallel
nerrors = 0
do j = 1, N
    do i = 1, M
        if (c(i,j) .ne. NTIMES) nerrors = nerrors + 1
    end do
end do
print *,"Number of Errors:",nerrors
gflops = 2.0 * N * M * K/time/1e9
write (*,901) m,k,k,N,time*1.0e3,gflops
901 format(i0,'x',i0,' * ',i0,'x',i0,':\t',f8.3,' ms\t',f12.3,' GFlops/s')
end
```

9.4. Using cuBLAS from CUDA Fortran Device Code

This example demonstrates the use of the cublas_device module from a CUDA Fortran global subroutines.

Simple BLAS Test from Device Code

```
module mtests
  use cublas_device
  contains
    attributes(global) subroutine testb( a, b, n )
    real, device :: a(*), b(*)
    integer, value :: n
    type(cublasHandle) :: h
    i = threadIdx%x
    if (i.eq.1) then
      j = cublasCreate(h)
      j = cublasSswap(h,n,a,1,b,1)
      j = cublasDestroy(h)
    end if
    return
    end subroutine
end module mtests

program tdev
! Compile and link with "pgfortran -Mcuda=cc35 tdev.cuf -lcublas_device"
use mtests
integer, parameter :: nt = 128
real, device :: a(nt), b(nt)
real c(nt)
a = 1.0
b = 2.0
call testb<<<1,nt>>> (a,b,nt)
c = a
print *,"Should be 2.0",all(c.eq.2.0)
c = b
print *,"Should be 1.0",all(c.eq.1.0)
end
end program
```

9.5. Using cuFFT from OpenACC Host Code

This example demonstrates the use of the cufft module, the `cufftHandle` type, and several cuFFT library calls.

Simple cuFFT Test

```
program cufft2dTest
  use cufft
  use openacc
  integer, parameter :: m=768, n=512
  complex, allocatable  :: a(:,:),b(:,:),c(:,:)
  real, allocatable     :: r(:,:),q(:,:)
  integer :: iplan1, iplan2, iplan3, ierr

  allocate(a(m,n),b(m,n),c(m,n))
  allocate(r(m,n),q(m,n))

  a = 1; r = 1
  xmx = -99.0

  ierr = cufftPlan2D(iplan1,m,n,CUFFT_C2C)
  ierr = ierr + cufftSetStream(iplan1,acc_get_cuda_stream(acc_async_sync))
  !$acc host_data use_device(a,b,c)
  ierr = ierr + cufftExecC2C(iplan1,a,b,CUFFT_FORWARD)
  ierr = ierr + cufftExecC2C(iplan1,b,c,CUFFT_INVERSE)
  !$acc end host_data

  ! scale c
  !$acc kernels
  c = c / (m*n)
  !$acc end kernels

  ! Check forward answer
  write(*,*) 'Max error C2C FWD: ', cmplx(maxval(real(b)) - sum(real(b)), &
                                          maxval(imag(b)))
  ! Check inverse answer
  write(*,*) 'Max error C2C INV: ', maxval(abs(a-c))

  ! Real transform
  ierr = ierr + cufftPlan2D(iplan2,m,n,CUFFT_R2C)
  ierr = ierr + cufftPlan2D(iplan3,m,n,CUFFT_C2R)
  ierr = ierr + cufftSetStream(iplan2,acc_get_cuda_stream(acc_async_sync))
  ierr = ierr + cufftSetStream(iplan3,acc_get_cuda_stream(acc_async_sync))

  !$acc host_data use_device(r,b,q)
  ierr = ierr + cufftExecR2C(iplan2,r,b)
  ierr = ierr + cufftExecC2R(iplan3,b,q)
  !$acc end host_data

  !$acc kernels
  xmx = maxval(abs(r-q/(m*n)))
  !$acc end kernels

  ! Check R2C + C2R answer
  write(*,*) 'Max error R2C/C2R: ', xmx

  ierr = ierr + cufftDestroy(iplan1)
  ierr = ierr + cufftDestroy(iplan2)
  ierr = ierr + cufftDestroy(iplan3)

  if (ierr.eq.0) then
    print *,"test PASSED"
```

```
  else
    print *,"test FAILED"
  endif

end program cufft2dTest
```

9.6. Using cuFFT from CUDA Fortran Host Code

This example demonstrates the use of the cuFFT module, the `cufftHandle` type, and several cuFFT library calls.

Simple cuFFT Test

```
program cufft2dTest
  use cudafor
  use cufft
  implicit none
  integer, parameter :: m=768, n=512
  complex, managed :: a(m,n),b(m,n)
  real, managed :: ar(m,n),br(m,n)
  real    x
  integer plan, ierr
  logical passing

  a = 1; ar = 1

  ierr = cufftPlan2D(plan,m,n,CUFFT_C2C)
  ierr = ierr + cufftExecC2C(plan,a,b,CUFFT_FORWARD)
  ierr = ierr + cufftExecC2C(plan,b,b,CUFFT_INVERSE)
  ierr = ierr + cudaDeviceSynchronize()
  x = maxval(abs(a-b/(m*n)))
  write(*,*) 'Max error C2C: ', x
  passing = x .le. 1.0e-5

  ierr = ierr + cufftPlan2D(plan,m,n,CUFFT_R2C)
  ierr = ierr + cufftExecR2C(plan,ar,b)
  ierr = ierr + cufftPlan2D(plan,m,n,CUFFT_C2R)
  ierr = ierr + cufftExecC2R(plan,b,br)
  ierr = ierr + cudaDeviceSynchronize()
  x = maxval(abs(ar-br/(m*n)))
  write(*,*) 'Max error R2C/C2R: ', x
  passing = passing .and. (x .le. 1.0e-5)

  ierr = ierr + cufftDestroy(plan)
  print *,ierr
  passing = passing .and. (ierr .eq. 0)
  if (passing) then
    print *,"Test PASSED"
  else
    print *,"Test FAILED"
  endif
end program cufft2dTest
```

9.7. Using cuRAND from OpenACC Host Code

This example demonstrates the use of the curand module, the `curandHandle` type, and several forms of rand calls.

Simple cuRAND Tests

```
program testcurand
```

```
! compile with the flags -ta=tesla -Mcuda -Mcudalib=curand
call cur1(1000, .true.); call cur1(1000, .false.)
call cur2(1000, .true.); call cur2(1000, .false.)
call cur3(1000, .true.); call cur3(1000, .false.)
end
!
subroutine cur1(n, onhost)
use curand
integer :: a(n)
type(curandGenerator) :: g
integer(8) nbits
logical onhost, passing
a = 0
passing = .true.
if (onhost) then
  istat = curandCreateGeneratorHost(g,CURAND_RNG_PSEUDO_XORWOW)
  istat = curandGenerate(g, a, n)
  istat = curandDestroyGenerator(g)
else
  !$acc data copy(a)
  istat = curandCreateGenerator(g,CURAND_RNG_PSEUDO_XORWOW)
  !$acc host_data use_device(a)
  istat = curandGenerate(g, a, n)
  !$acc end host_data
  istat = curandDestroyGenerator(g)
  !$acc end data
endif
nbits = 0
do i = 1, n
  if (i.lt.10) print *,i,a(i)
  nbits = nbits + popcnt(a(i))
end do
print *,"Should be roughly half the bits set"
nbits = nbits / n
if ((nbits .lt. 12) .or. (nbits .gt. 20)) then
  passing = .false.
else
  print *,"nbits is ",nbits," which passes"
endif
if (passing) then
  print *,"Test PASSED"
else
  print *,"Test FAILED"
endif
end
!
subroutine cur2(n, onhost)
use curand
real :: a(n)
type(curandGenerator) :: g
logical onhost, passing
a = 0.0
passing = .true.
if (onhost) then
  istat = curandCreateGeneratorHost(g,CURAND_RNG_PSEUDO_XORWOW)
  istat = curandGenerate(g, a, n)
  istat = curandDestroyGenerator(g)
else
  !$acc data copy(a)
  istat = curandCreateGenerator(g,CURAND_RNG_PSEUDO_XORWOW)
  !$acc host_data use_device(a)
  istat = curandGenerate(g, a, n)
  !$acc end host_data
  istat = curandDestroyGenerator(g)
  !$acc end data
endif
print *,"Should be uniform around 0.5"
```

```
do i = 1, n
  if (i.lt.10) print *,i,a(i)
  if ((a(i).lt.0.0) .or. (a(i).gt.1.0)) passing = .false.
end do
rmean = sum(a)/n
if ((rmean .lt. 0.4) .or. (rmean .gt. 0.6)) then
  passing = .false.
else
  print *,"Mean is ",rmean," which passes"
endif
if (passing) then
  print *,"Test PASSED"
else
  print *,"Test FAILED"
endif
end
!
subroutine cur3(n, onhost)
use curand
real(8) :: a(n)
type(curandGenerator) :: g
logical onhost, passing
a = 0.0d0
passing = .true.
if (onhost) then
  istat = curandCreateGeneratorHost(g,CURAND_RNG_PSEUDO_XORWOW)
  istat = curandGenerate(g, a, n)
  istat = curandDestroyGenerator(g)
else
  !$acc data copy(a)
  istat = curandCreateGenerator(g,CURAND_RNG_PSEUDO_XORWOW)
  !$acc host_data use_device(a)
  istat = curandGenerate(g, a, n)
  !$acc end host_data
  istat = curandDestroyGenerator(g)
  !$acc end data
endif
do i = 1, n
  if (i.lt.10) print *,i,a(i)
  if ((a(i).lt.0.0d0) .or. (a(i).gt.1.0d0)) passing = .false.
end do
rmean = sum(a)/n
if ((rmean .lt. 0.4d0) .or. (rmean .gt. 0.6d0)) then
  passing = .false.
else
  print *,"Mean is ",rmean," which passes"
endif
if (passing) then
  print *,"Test PASSED"
else
  print *,"Test FAILED"
endif
end
```

9.8. Using cuRAND from OpenACC Device Code

This example demonstrates the use of the curand_device module from a CUDA Fortran global subroutines.

Simple cuRAND Test from OpenACC Device Code

```
module mtests
  integer, parameter :: n = 1000
  contains
```

```
    subroutine testrand( a, b )
    use openacc_curand
    real :: a(n), b(n)
    type(curandStateXORWOW) :: h
    integer(8) :: seed, seq, offset

    !$acc parallel num_gangs(1) vector_length(1) copy(a,b) private(h)
    seed = 12345
    seq = 0
    offset = 0
    call curand_init(seed, seq, offset, h)
    !$acc loop seq
    do i = 1, n
      a(i) = curand_uniform(h)
      b(i) = curand_normal(h)
    end do
    !$acc end parallel
    return
    end subroutine
end module mtests

program t
use mtests
real :: a(n), b(n), c(n)
logical passing
a = 1.0
b = 2.0
passing = .true.
call testrand(a,b)
c = a
print *,"Should be uniform around 0.5"
do i = 1, n
  if (i.lt.10) print *,i,c(i)
  if ((c(i).lt.0.0) .or. (c(i).gt.1.0)) passing = .false.
end do
rmean = sum(c)/n
if ((rmean .lt. 0.4) .or. (rmean .gt. 0.6)) then
  passing = .false.
else
  print *,"Mean is ",rmean," which passes"
endif
c = b
print *,"Should be normal around 0.0"
nc1 = 0;
nc2 = 0;
do i = 1, n
  if (i.lt.10) print *,i,c(i)
  if ((c(i) .gt. -4.0) .and. (c(i) .lt. 0.0)) nc1 = nc1 + 1
  if ((c(i) .gt.  0.0) .and. (c(i) .lt. 4.0)) nc2 = nc2 + 1
end do
print *,"Found on each side of zero ",nc1,nc2
if (abs(nc1-nc2) .gt. (n/10)) npassing = .false.
rmean = sum(c,mask=abs(c).lt.4.0)/n
if ((rmean .lt. -0.1) .or. (rmean .gt. 0.1)) then
  passing = .false.
else
  print *,"Mean is ",rmean," which passes"
endif

if (passing) then
  print *,"Test PASSED"
else
  print *,"Test FAILED"
endif
end program
```

9.9. Using cuRAND from CUDA Fortran Host Code

This example demonstrates the use of the curand module, the `curandHandle` type, and several forms of rand calls.

Simple cuRAND Test

```
program testcurand1
call testr1(1000)
call testr2(1000)
call testr3(1000)
end
!
subroutine testr1(n)
use cudafor
use curand
integer, managed :: a(n)
type(curandGenerator) :: g
integer(8) nbits
logical passing
a = 0
passing = .true.
istat = curandCreateGenerator(g,CURAND_RNG_PSEUDO_XORWOW)
istat = curandGenerate(g, a, n)
istat = cudaDeviceSynchronize()
istat = curandDestroyGenerator(g)
nbits = 0
do i = 1, n
  if (i.lt.10) print *,i,a(i)
  nbits = nbits + popcnt(a(i))
end do
print *,"Should be roughly half the bits set"
nbits = nbits / n
if ((nbits .lt. 12) .or. (nbits .gt. 20)) then
  passing = .false.
else
  print *,"nbits is ",nbits," which passes"
endif
if (passing) then
  print *,"Test PASSED"
else
  print *,"Test FAILED"
endif
end
!
subroutine testr2(n)
use cudafor
use curand
real, managed :: a(n)
type(curandGenerator) :: g
logical passing
a = 0.0
passing = .true.
istat = curandCreateGenerator(g,CURAND_RNG_PSEUDO_XORWOW)
istat = curandGenerate(g, a, n)
istat = cudaDeviceSynchronize()
istat = curandDestroyGenerator(g)
print *,"Should be uniform around 0.5"
do i = 1, n
  if (i.lt.10) print *,i,a(i)
  if ((a(i).lt.0.0) .or. (a(i).gt.1.0)) passing = .false.
end do
rmean = sum(a)/n
```

```
if ((rmean .lt. 0.4) .or. (rmean .gt. 0.6)) then
  passing = .false.
else
  print *,"Mean is ",rmean," which passes"
endif
if (passing) then
  print *,"Test PASSED"
else
  print *,"Test FAILED"
endif
end
!
subroutine testr3(n)
use cudafor
use curand
real(8), managed :: a(n)
type(curandGenerator) :: g
logical passing
a = 0.0d0
passing = .true.
istat = curandCreateGenerator(g,CURAND_RNG_PSEUDO_XORWOW)
istat = curandGenerate(g, a, n)
istat = cudaDeviceSynchronize()
istat = curandDestroyGenerator(g)
do i = 1, n
  if (i.lt.10) print *,i,a(i)
  if ((a(i).lt.0.0d0) .or. (a(i).gt.1.0d0)) passing = .false.
end do
rmean = sum(a)/n
if ((rmean .lt. 0.4d0) .or. (rmean .gt. 0.6d0)) then
  passing = .false.
else
  print *,"Mean is ",rmean," which passes"
endif
if (passing) then
  print *,"Test PASSED"
else
  print *,"Test FAILED"
endif
end
end program
```

9.10. Using cuRAND from CUDA Fortran Device Code

This example demonstrates the use of the curand_device module from a CUDA Fortran global subroutines.

Simple cuRAND Test from Device Code

```
module mtests
  use curand_device
  integer, parameter :: n = 10000
  contains
    attributes(global) subroutine testr( a, b )
    real, device :: a(n), b(n)
    type(curandStateXORWOW) :: h
    integer(8) :: seed, seq, offset
    integer :: iam
    iam = threadIdx%x
    seed = iam*64 + 12345
    seq = 0
    offset = 0
```

```
    call curand_init(seed, seq, offset, h)
    do i = iam, n, blockdim%x
      a(i) = curand_uniform(h)
      b(i) = curand_normal(h)
    end do
    return
    end subroutine
end module mtests

program t
use mtests
real, allocatable, device :: a(:), b(:)
real c(n), rmean
logical passing
allocate(a(n))
allocate(b(n))
a = 0.0
b = 0.0
passing = .true.
call testr<<<1,32>>> (a,b)
c = a
print *,"Should be uniform around 0.5"
do i = 1, n
  if (i.lt.10) print *,i,c(i)
  if ((c(i).lt.0.0) .or. (c(i).gt.1.0)) passing = .false.
end do
rmean = sum(c)/n
if ((rmean .lt. 0.4) .or. (rmean .gt. 0.6)) then
  passing = .false.
else
  print *,"Mean is ",rmean," which passes"
endif

c = b
print *,"Should be normal around 0.0"
nc1 = 0;
nc2 = 0;
do i = 1, n
  if (i.lt.10) print *,i,c(i)
  if ((c(i) .gt. -4.0) .and. (c(i) .lt. 0.0)) nc1 = nc1 + 1
  if ((c(i) .gt.  0.0) .and. (c(i) .lt. 4.0)) nc2 = nc2 + 1
end do
print *,"Found on each side of zero ",nc1,nc2
if (abs(nc1-nc2) .gt. (n/10)) npassing = .false.
rmean = sum(c,mask=abs(c).lt.4.0)/n
if ((rmean .lt. -0.1) .or. (rmean .gt. 0.1)) then
  passing = .false.
else
  print *,"Mean is ",rmean," which passes"
endif

if (passing) then
  print *,"Test PASSED"
else
  print *,"Test FAILED"
endif
end
end program
```

9.11. Using cuSPARSE from OpenACC Host Code

This example demonstrates the use of the cuSPARSE module, the `cusparseHandle` type, and several calls to the cuSPARSE library.

Simple BLAS Test

```
program sparseMatVec
    integer n
    n = 25 ! # rows/cols in dense matrix
    call sparseMatVecSub1(n)
    n = 45 ! # rows/cols in dense matrix
    call sparseMatVecSub1(n)
end program

subroutine sparseMatVecSub1(n)
  use openacc
  use cusparse

  implicit none

  integer n

  ! dense data
  real(4), allocatable :: Ade(:,:), x(:), y(:)

  ! sparse CSR arrays
  real(4), allocatable :: csrValA(:)
  integer, allocatable :: nnzPerRowA(:), csrRowPtrA(:), csrColIndA(:)

  allocate(Ade(n,n), x(n), y(n))
  allocate(csrValA(n))
  allocate(nnzPerRowA(n), csrRowPtrA(n+1), csrColIndA(n))

  call sparseMatVecSub2(Ade, x, y, csrValA, nnzPerRowA, csrRowPtrA, &
                                                        csrColIndA, n)
  deallocate(Ade)
  deallocate(x)
  deallocate(y)
  deallocate(csrValA)
  deallocate(nnzPerRowA)
  deallocate(csrRowPtrA)
  deallocate(csrColIndA)
end subroutine

subroutine sparseMatVecSub2(Ade, x, y, csrValA, nnzPerRowA, csrRowPtrA, &
                                                          csrColIndA, n)
  use openacc
  use cusparse

  implicit none

  ! dense data
  real(4) :: Ade(n,n), x(n), y(n)

  ! sparse CSR arrays
  real(4) :: csrValA(n)
  integer :: nnzPerRowA(n), csrRowPtrA(n+1), csrColIndA(n)

  integer :: n, nnz, status, i
  type(cusparseHandle) :: h
  type(cusparseMatDescr) :: descrA

  ! parameters
  real(4) :: alpha, beta

  ! result
  real(4) :: xerr

  ! initalize CUSPARSE and matrix descriptor
  status = cusparseCreate(h)
  if (status /= CUSPARSE_STATUS_SUCCESS) &
```

```
     write(*,*) 'cusparseCreate error: ', status
status = cusparseCreateMatDescr(descrA)
status = cusparseSetMatType(descrA, &
     CUSPARSE_MATRIX_TYPE_GENERAL)
status = cusparseSetMatIndexBase(descrA, &
     CUSPARSE_INDEX_BASE_ONE)
status = cusparseSetStream(h, acc_get_cuda_stream(acc_async_sync))

!$acc data create(Ade, x, y, csrValA, nnzPerRowA, csrRowPtrA, csrColIndA)

! Initialize matrix (upper circular shift matrix)
!$acc kernels
Ade = 0.0
do i = 1, n-1
   Ade(i,i+1) = 1.0
end do
Ade(n,1) = 1.0

! Initialize vectors and constants
do i = 1, n
   x(i) = i
enddo
y = 0.0
!$acc end kernels

!$acc update host(x)
write(*,*) 'Original vector:'
write(*,'(5(1x,f7.2))') x

! convert matrix from dense to csr format
!$acc host_data use_device(Ade, nnzPerRowA, csrValA, csrRowPtrA, csrColIndA)
status = cusparseSnnz_v2(h, CUSPARSE_DIRECTION_ROW, &
     n, n, descrA, Ade, n, nnzPerRowA, nnz)
status = cusparseSdense2csr(h, n, n, descrA, Ade, n, &
     nnzPerRowA, csrValA, csrRowPtrA, csrColIndA)
!$acc end host_data

! A is upper circular shift matrix
! y = alpha*A*x + beta*y
alpha = 1.0
beta = 0.0
!$acc host_data use_device(csrValA, csrRowPtrA, csrColIndA, x, y)
status = cusparseScsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
     n, n, n, alpha, descrA, csrValA, csrRowPtrA, &
     csrColIndA, x, beta, y)
!$acc end host_data

!$acc wait
write(*,*) 'Shifted vector:'
write(*,'(5(1x,f7.2))') y

! shift-down y and add original x
! A' is lower circular shift matrix
! x = alpha*A'*y + beta*x
beta = -1.0
!$acc host_data use_device(csrValA, csrRowPtrA, csrColIndA, x, y)
status = cusparseScsrmv(h, CUSPARSE_OPERATION_TRANSPOSE, &
     n, n, n, alpha, descrA, csrValA, csrRowPtrA, &
     csrColIndA, y, beta, x)
!$acc end host_data

!$acc kernels
xerr = maxval(abs(x))
!$acc end kernels
!$acc end data

write(*,*) 'Max error = ', xerr
```

```
  if (xerr.le.1.e-5) then
    write(*,*) 'Test PASSED'
  else
    write(*,*) 'Test FAILED'
  endif

end subroutine
```

9.12. Using cuSPARSE from CUDA Fortran Host Code

This example demonstrates the use of the cuSPARSE module, the `cusparseHandle` type, and several forms of cusparse calls.

Simple BLAS Test

```
program sparseMatVec
  use cudafor
  use cusparse

  implicit none

  integer, parameter :: n = 25 ! # rows/cols in dense matrix

  type(cusparseHandle) :: h
  type(cusparseMatDescr) :: descrA

  ! dense data
  real(4), managed :: Ade(n,n), x(n), y(n)

  ! sparse CSR arrays
  real(4), managed :: csrValA(n)
  integer, managed :: nnzPerRowA(n), &
       csrRowPtrA(n+1), csrColIndA(n)
  integer :: nnz, status, i

  ! parameters
  real(4) :: alpha, beta

  ! initalize CUSPARSE and matrix descriptor
  status = cusparseCreate(h)
  if (status /= CUSPARSE_STATUS_SUCCESS) &
       write(*,*) 'cusparseCreate error: ', status
  status = cusparseCreateMatDescr(descrA)
  status = cusparseSetMatType(descrA, &
       CUSPARSE_MATRIX_TYPE_GENERAL)
  status = cusparseSetMatIndexBase(descrA, &
       CUSPARSE_INDEX_BASE_ONE)

  ! Initialize matrix (upper circular shift matrix)
  Ade = 0.0
  do i = 1, n-1
     Ade(i,i+1) = 1.0
  end do
  Ade(n,1) = 1.0

  ! Initialize vectors and constants
  x = [(i,i=1,n)]
  y = 0.0

  write(*,*) 'Original vector:'
  write(*,'(5(1x,f7.2))') x
```

```
  ! convert matrix from dense to csr format
  status = cusparseSnnz_v2(h, CUSPARSE_DIRECTION_ROW, &
       n, n, descrA, Ade, n, nnzPerRowA, nnz)
  status = cusparseSdense2csr(h, n, n, descrA, Ade, n, &
       nnzPerRowA, csrValA, csrRowPtrA, csrColIndA)

  ! A is upper circular shift matrix
  ! y = alpha*A*x + beta*y
  alpha = 1.0
  beta = 0.0
  status = cusparseScsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
       n, n, n, alpha, descrA, csrValA, csrRowPtrA, &
       csrColIndA, x, beta, y)

  ! shift-down y and add original x
  ! A' is lower circular shift matrix
  ! x = alpha*A'*y + beta*x
  beta = -1.0
  status = cusparseScsrmv(h, CUSPARSE_OPERATION_TRANSPOSE, &
       n, n, n, alpha, descrA, csrValA, csrRowPtrA, &
       csrColIndA, y, beta, x)

  status = cudaDeviceSynchronize()

  write(*,*) 'Shifted vector:'
  write(*,'(5(1x,f7.2))') y

  write(*,*) 'Max error = ', maxval(abs(x))

  if (maxval(abs(x)).le.1.e-5) then
    write(*,*) 'Test PASSED'
  else
    write(*,*) 'Test FAILED'
  endif

end program sparseMatVec
```

9.13. Using cuTENSOR from CUDA Fortran Host Code

This example demonstrates the use of the low-level cuTENSOR module from CUDA Fortran to perform a reshape permutation and a sum reduction across one dimension..

This example can be compiled and linked as a normal CUDA Fortran subprogram by adding the "-Mcudalib=cutensor" option to the link line.

Simple cuTENSOR Test from CUDA Fortran

```
program testcutensorcuf1
use cudafor
use cutensor

integer, parameter :: ndim = 3
real, managed, allocatable :: dA(:,:,:), dC(:,:)
real, allocatable :: hA(:,:,:), hC(:,:)
real, device, allocatable :: workbuf(:)
real :: alpha, beta
integer(4) :: modesA, modesC
integer(8), dimension(ndim) :: extA, strA, extC, strC
integer(4), dimension(ndim) :: modeA, modeC
integer(8) :: workbufsize
```

```
type(cutensorStatus) :: cstat
type(cutensorHandle) :: handle
type(cutensorDescriptor) :: descA, descC

! Init
cstat = cutensorInit(handle)
if (cstat.ne.CUTENSOR_STATUS_SUCCESS) print *,cutensorGetErrorString(cstat)

allocate(workbuf(2500))
workbufsize = 2500 * 4

allocate(dA(100,60,80))
!
! This is the operation we are going to perform
! dC = sum(reshape(dA,shape=[80,60,100],order=[3,2,1]),dim=2)
!
call random_number(dA); dA = real(int(dA * 10.0))
extA = shape(dA)
strA(1) =  1; strA(2) = 100; strA(3) = 6000
modeA(1) = 3; modeA(2) = 2; modeA(3) = 1  ! Reshape Order Values
modesA = ndim

cstat = cutensorInitTensorDescriptor(handle, descA, modesA, extA, strA, &
             cudaDataType(CUDA_R_32F), CUTENSOR_OP_IDENTITY)
if (cstat.ne.CUTENSOR_STATUS_SUCCESS) print *,cutensorGetErrorString(cstat)

allocate(dC(80,100))
dC = 0.0
extC(1:ndim-1) = shape(dC)
strC(1) = 1; strC(2) = 80
modeC(1) = 1; modeC(2) = 3  ! Skip mode 2 for reduction across that dim
modesC = ndim-1

cstat = cutensorInitTensorDescriptor(handle, descC, modesC, extC, strC, &
             cudaDataType(CUDA_R_32F), CUTENSOR_OP_IDENTITY)
if (cstat.ne.CUTENSOR_STATUS_SUCCESS) print *,cutensorGetErrorString(cstat)

alpha = 1.0; beta = 0.0
cstat = cutensorReduction(handle, alpha, dA, descA, modeA, &
             beta, dC, descC, modeC, dC, descC, modeC, &
             CUTENSOR_OP_ADD, CUTENSOR_R_MIN_32F, &
             workbuf, workbufsize, 0)
if (cstat.ne.CUTENSOR_STATUS_SUCCESS) print *,cutensorGetErrorString(cstat)

hA = dA
hC = sum(reshape(hA,[80,60,100],order=[3,2,1]), dim=2)
istat = cudaDeviceSynchronize()  ! Managed memory, to be sure

if (all(hC.eq.dC)) then
  print *,"test PASSED"
else
  print *,"test FAILED"
end if
end program
```

9.14. Using cuTENSOREX from CUDA Fortran Host Code

This example demonstrates the use of the higher-level cuTENSOREX module from CUDA Fortran to perform a large matrix multiplication using multiple OpenMP threads.

This example can be compiled and linked as a normal CUDA Fortran subprogram by adding the "-mp -Mcudalib=cutensor" options to the compile and link line.

Multi-threaded cuTENSOREX Example from CUDA Fortran

```
! Test cuTensor + cuda Fortran + OMP multi-stream matmul
program testCuCufMsMatmul
use cudafor
use cutensorex

integer, parameter :: m=8192, k=1280, n=1024
integer, parameter :: mblksize = 128
integer, parameter :: mslices = m / mblksize
integer, parameter :: nstreams = 4
integer, parameter :: numtimes = mslices / nstreams

real(8), allocatable, dimension(:,:), device :: a_d, d_d
real(8), allocatable, dimension(:,:,:), pinned :: ha
real(8), dimension(k,n), device :: b_d
real(8), allocatable, dimension(:,:), pinned :: d
real(8) :: alpha
integer(kind=cuda_stream_kind) :: mystream
!$OMP THREADPRIVATE(a_d, d_d, mystream)

allocate( ha(k,mblksize,nstreams))
allocate(  d(1:m,1:n))

b_d   = 1.0d0
alpha = 1.0d0

!$OMP PARALLEL NUM_THREADS(nstreams) PRIVATE(istat)
istat = cudaStreamCreate(mystream)
istat = cudaforSetDefaultStream(mystream)
istat = cutensorExSetStream(mystream)
! At this point, all new allocations will pick up default stream
allocate(a_d(k,mblksize))
allocate(d_d(mblksize,n))
!$OMP END PARALLEL

! Test matmul
!$OMP PARALLEL DO NUM_THREADS(nstreams) PRIVATE(jlcl,jgbl,jend)
do ns = 1, nstreams
    do nt = 1, numtimes
        jgbl = 1 + ((ns-1) + (nt-1)*nstreams)*mblksize
        jend = jgbl + mblksize - 1
        ! Do some host work
        do jlcl = 1, mblksize
            ha(:,jlcl,ns) = dble(jlcl+jgbl-1)
        end do
        ! Move data to the device on default stream
        a_d = ha(:,:,ns)
        ! Matrix multiply on my thread cutensorEx stream
        d_d = alpha * matmul(transpose(a_d),b_d)
        ! Move result back to host on default stream
        d(jgbl:jend,:) = d_d
    end do
end do
! Wait for all threads to finish GPU work
istat = cudaDeviceSynchronize()
nfailed = 0
do j = 1, n
    do i = 1, m
        if (d(i,j) .ne. i*k) then
            if (nfailed .lt. 100) print *,i,j,d(i,j)
            nfailed = nfailed + 1
        end if
    end do
```

```
end do
if (nfailed .eq. 0) then
    print *,"test PASSED"
else
    print *,"test FAILED"
endif
end program
```

9.15. Using cuTENSOR from OpenACC Host Code

This example demonstrates the use of the cuTENSOREX module, calling Matmul() using OpenACC device data, and setting the cuTENSOR library stream to be consistent with the OpenACC default stream.

This example can be compiled and run with or without OpenACC. To compile with OpenACC, the options are "-ta=tesla -Mcuda -Mcudalib=cutensor". To run on the CPU, leave off those options.

Simple cuTENSOREX Test from OpenACC

```
program testcutensorOpenACC
!@acc use openacc
!@acc use cutensorex
integer, parameter :: ni=1280, nj=1024, nk=960, ntimes=1
real(8) :: a(ni,nk), b(nk,nj), c(ni,nj), d(ni,nj)

call random_number(a)
call random_number(b)
a = dble(int(4.0d0*a - 2.0d0))
b = dble(int(8.0d0*b - 4.0d0))
c = 2.0; d = 0.0

!$acc enter data copyin(a,b,c) create(d)
!@acc istat = cutensorExSetStream(acc_get_cuda_stream(acc_async_sync))
!$acc host_data use_device(a,b,c,d)
do nt = 1, ntimes
  d = c + matmul(a,b)
end do
!$acc end host_data

!$acc update host(d)
print *,sum(d)

do nt = 1, ntimes
!$acc kernels
  do j = 1, nj
    do i = 1, ni
      d(i,j) = c(i,j)
      do k = 1, nk
        d(i,j) = d(i,j) + a(i,k) * b(k,j)
      end do
    end do
  end do
!$acc end kernels
end do
!$acc exit data copyout(d)

print *,sum(d)
end program
```