
Python

Advanced Micro Devices Disclaimer and License

Jul 05, 2023

CONTENTS

1	Introduction	1
1.1	Documentation Roadmap	1
1.2	Use of Tensile	1
2	Installation and Building for Linux	3
2.1	Prerequisites	3
2.2	Installing Prebuilt Packages	3
2.3	Building and Installing rocBLAS	4
3	Installation and Building for Windows	7
3.1	Prerequisites	7
3.2	Installing Prebuilt Packages	7
3.3	Building and Installing rocBLAS	8
4	API Reference Guide	11
4.1	Introduction	11
4.2	rocBLAS API and Legacy BLAS Functions	11
4.3	Deprecations by version	18
5	Using rocBLAS API	21
5.1	rocBLAS Datatypes	21
5.2	rocBLAS Enumeration	22
5.3	rocBLAS Helper functions	28
5.4	rocBLAS Level-1 functions	33
5.5	rocBLAS Level-2 functions	62
5.6	rocBLAS Level-3 functions	147
5.7	rocBLAS Extension	206
5.8	rocBLAS Beta Features	243
5.9	Graph Support for rocBLAS	248
5.10	Device Memory Allocation in rocBLAS	249
5.11	Logging in rocBLAS	251
6	Programmer's Guide	253
6.1	Library Source Code Organization	253
6.2	Handle, Stream, and Device Management	255
6.3	Device Memory Allocation	257
6.4	Thread Safe Logging	261
6.5	rocBLAS Numerical Checking	262
6.6	rocBLAS Order of Argument Checking and Logging	263
6.7	rocBLAS Benchmarking and Testing	270

7 Contributor's Guide	289
7.1 Pull-request guidelines	289
7.2 Coding Guidelines	289
7.3 Static Code Analysis	301
8 Acknowledgement	303
Bibliography	305
Index	307

INTRODUCTION

This document contains instructions for installing, using, and contributing to rocBLAS. The quickest way to install is from prebuilt packages. Alternatively, there are instructions to build from source. The document also contains an API Reference Guide, Programmer's Guide, and Contributor's Guide.

1.1 Documentation Roadmap

The following is a list of rocBLAS documents in the suggested reading order:

- [Installation Guide](#) (either [Linux](#) or [Windows](#)): Describes how to install and configure the rocBLAS library; designed to get users up and running quickly with the library
- [API Reference Guide](#): Provides detailed information about rocBLAS functions, data types and other programming constructs
- [Programmer's Guide](#): Describes the code organization, Design implementation detail, Optimizations used in the library, and those that should be considered for new development and Testing & Benchmarking detail
- [Contributor's Guide](#): Describes coding guidelines for contributors

1.2 Use of Tensile

The rocBLAS library internally uses [Tensile](#), which supplies the high-performance implementation of xGEMM. It requires no separate installation as it is installed as part of the rocBLAS package. If building rocBLAS library CMake downloads Tensile during library configuration and automatically configures it as part of the build, so no further action is required by the user to set it up. No external facing API for Tensile is provided.

INSTALLATION AND BUILDING FOR LINUX

2.1 Prerequisites

- A ROCm enabled platform. More information in [ROCm Documentation](#). To build ROCm from the source, follow the previous link and navigate to How to install ROCm. This page will provide steps to install ROCm for specific operating systems. After following these steps to download the installer amdgpu-install, use the below command to install ROCm.

```
sudo amdgpu-install --usecase=rocm
```

- rocBLAS is supported on the same Linux versions that are supported by ROCm

2.2 Installing Prebuilt Packages

rocBLAS can be installed on Ubuntu(R) or Debian using:

```
sudo apt-get update
sudo apt-get install rocblas
```

rocBLAS can be installed on CentOS using:

```
sudo yum update
sudo yum install rocblas
```

rocBLAS can be installed on SLES using:

```
sudo dnf upgrade
sudo dnf install rocblas
```

Once installed, rocBLAS can be used just like any other library with a C API. The rocblas.h header file must be included in the user code to make calls into rocBLAS, and the rocBLAS shared library will become link-time and run-time dependent for the user application.

Once installed, find rocblas.h and rocblas_module.f90 in the /opt/rocm/include directory. Only use these two installed files when needed in user code. Find other rocBLAS files in /opt/rocm/include/internal, However, do not include these files directly.

2.3 Building and Installing rocBLAS

For most users, building from source is not necessary, as rocBLAS can be used after installing the prebuilt packages as described above. If desired, users can use following instructions to build rocBLAS from source. Note the change in the `--usecase` command argument below to install the base ROCm developer stack in-order to build rocBLAS from the source.

```
sudo amdgpu-install --usecase=rocmdev
```

2.3.1 Requirements

As a rule, 64GB of system memory is required for a full rocBLAS build. This value can be lower if rocBLAS is built with a different Tensile logic target (see the `--logic` command for `.install.sh`). This value may also increase in the future as more functions are added to rocBLAS and dependencies such as Tensile grow.

2.3.2 Download rocBLAS

The rocBLAS source code is available at the [rocBLAS github page](#). Check the ROCm version on your system. For Ubuntu(R), use:

```
apt show rocm-libs -a
```

For Centos, use:

```
yum info rocm-libs
```

The ROCm version has major, minor, and patch fields, possibly followed by a build specific identifier. For example, ROCm version could be 4.0.0.40000-23; this corresponds to major = 4, minor = 0, patch = 0, build identifier 40000-23. There are GitHub branches at the rocBLAS site with names `rocm-major.minor.x` where major and minor are the same as in the ROCm version. For ROCm version 4.0.0.40000-23, you must use the following to download rocBLAS:

```
git clone -b release/rocm-rel-x.y https://github.com/ROCmSoftwarePlatform/rocBLAS.git
cd rocBLAS
```

Replace `x.y` in the above command with the version of ROCm installed on your machine. For example, if you have ROCm 5.0 installed, then replace `release/rocm-rel-x.y` with `release/rocm-rel-5.0`.

Below are steps to build using `install.sh` script. The user can build either:

- dependencies + library
- dependencies + library + client

You only need (dependencies + library) if you call rocBLAS from your code. The client contains the test and benchmark code.

2.3.3 Library Dependencies

Dependencies are listed in the script `install.sh`. The `-d` flag to `install.sh` installs dependencies.

CMake has a minimum version requirement listed in the file `install.sh`. See `-cmake_install` flag in `install.sh` to upgrade automatically.

2.3.4 Build Library dependencies + Library

Common uses of `install.sh` to build (library dependencies + library) are in the table below:

Command	Description
<code>./install.sh -h</code>	Help information.
<code>./install.sh -d</code>	Build library dependencies and library in your local directory. The <code>-d</code> flag only needs to be used once. For subsequent invocations of <code>install.sh</code> it is not necessary to rebuild the dependencies.
<code>./install.sh</code>	Build library in your local directory. It is assumed dependencies have been built.
<code>./install.sh -i</code>	Build library, then build and install rocBLAS package in <code>/opt/rocm/roclblas</code> . You will be prompted for sudo access. This will install for all users. If you want to keep rocBLAS in your local directory, you do not need the <code>-i</code> flag.

2.3.5 Build Library Dependencies + Client Dependencies + Library + Client

Some client executables are listed in the table below:

executable name	description
<code>roclblas-test</code>	runs Google Tests to test the library
<code>roclblas-bench</code>	executable to benchmark or test functions
<code>roclblas-example-sscal</code>	example C code calling <code>roclblas_sscal</code> function

Common uses of `install.sh` to build (dependencies + library + client) are in the table below:

Command	Description
<code>./install.sh -h</code>	Help information.
<code>./install.sh -dc</code>	Build library dependencies, client dependencies, library, and client in your local directory. The <code>-d</code> flag only needs to be used once. For subsequent invocations of <code>install.sh</code> it is not necessary to rebuild the dependencies.
<code>./install.sh -c</code>	Build library and client in your local directory. It is assumed the dependencies have been built.
<code>./install.sh -idc</code>	Build library dependencies, client dependencies, library, client, then build and install the rocBLAS package. You will be prompted for sudo access. It is expected that if you want to install for all users you use the <code>-i</code> flag. If you want to keep rocBLAS in your local directory, you do not need the <code>-i</code> flag.
<code>./install.sh -ic</code>	Build and install rocBLAS package, and build the client. You will be prompted for sudo access. This will install for all users. If you want to keep rocBLAS in your local directory, you do not need the <code>-i</code> flag.

2.3.6 Build Clients without Library

The rocBLAS clients can be built on their own using *install.sh* with a preexisting rocBLAS library.

Note that the version of the rocBLAS clients being built should match the version of the installed rocBLAS. Find the version of the installed rocBLAS in the installed rocBLAS directory in the file `include/internal/rocbblas-version.h`. Find the version of rocBLAS being built by running `grep "VERSION_STRING" CMakeLists.txt` in the rocBLAS directory being built.

Command	Description
<code>./install.sh --clients-only</code>	Build rocBLAS clients and use an installed rocBLAS library at <code>ROCM_PATH</code> (<code>/opt/rocm</code> if not specified).
<code>./install.sh --clients-only --library-path / path/to/rocbblas</code>	Build rocBLAS clients and use a rocBLAS library at the specified location.

INSTALLATION AND BUILDING FOR WINDOWS

3.1 Prerequisites

- An AMD HIP SDK enabled platform. More information can be found [here](#).
- rocBLAS is supported on the same Windows versions and toolchains that are supported by the HIP SDK.
- As the AMD HIP SDK is new and quickly evolving it will have more up to date information regarding the SDK's internal contents. Thus it may overrule statements found in this section on installing and building for Windows.

3.2 Installing Prebuilt Packages

rocBLAS can be installed on Windows 11 or Windows 10 using the AMD HIP SDK installer.

The simplest way to use rocBLAS in your code would be using CMake for which you would add the SDK installation location to your *CMAKE_PREFIX_PATH*. Note you need to use quotes as the path contains a space, e.g.,

```
-DCMAKE_PREFIX_PATH="C:\Program Files\AMD\ROCm\5.5"
```

in your CMake configure step and then in your CMakeLists.txt use

```
find_package(rocblas)

target_link_libraries( your_exe PRIVATE roc::rocblas )
```

Example code of consuming rocBLAS on windows with CMake can be found at [rocBLAS-Examples github page](#).

Otherwise once installed, rocBLAS can be used just like any other library with a C API. The rocblas.h header file must be included in the user code to make calls into rocBLAS, and the rocBLAS import library and dynamic link library will become respective link-time and run-time dependencies for the user application. Note an additional runtime dependency beyond the dynamic link library (.dll) file is the entire rocblas/ subdirectory found in the HIP SDK bin folder. This must be kept in the same directory as the rocblas.dll or can be located elsewhere if setting the environment variable *ROCBLAS_TENSILE_LIBPATH* to the non-standard location. The contents are read at execution time much like additional DLL files.

Once installed, find rocblas.h in the HIP SDK *\include\rocblas* directory. Only use these two installed files when needed in user code. Find other rocBLAS included files in HIP SDK *\include\rocblas\internal*, however, do not include these files directly into source code.

3.3 Building and Installing rocBLAS

For most users, building from source is not necessary, as rocBLAS can be used after installing the prebuilt packages as described above. If desired, users can use the following instructions to build rocBLAS from source. The codebase used for rocBLAS for the HIP SDK is the same as used for linux ROCm distribution. However as these two distributions have different stacks the code and build process have subtle variations.

3.3.1 Requirements

As a rough estimate, 64GB of system memory is required for a full rocBLAS build. This value can be lower if rocBLAS is built with a different Tensile logic target (see the `-logic` command from `rmake.py --help`). This value may also increase in the future as more functions are added to rocBLAS and dependencies such as Tensile grow.

3.3.2 Download rocBLAS

The rocBLAS source code, which is the same as for the ROCm linux distributions, is available at the [rocBLAS github page](#). The version of the ROCm HIP SDK may be shown in the path of default installation, but you can run the HIP SDK compiler to report the version from the `bin/` folder with:

```
hipcc --version
```

The HIP version has major, minor, and patch fields, possibly followed by a build specific identifier. For example, HIP version could be 5.4.22880-135e1ab4; this corresponds to major = 5, minor = 4, patch = 22880, build identifier 135e1ab4. There are GitHub branches at the rocBLAS site with names `release/rocm-rel-major.minor` where major and minor are the same as in the HIP version. For example for you can use the following to download rocBLAS:

```
git clone -b release/rocm-rel-x.y https://github.com/ROCmSoftwarePlatform/rocBLAS.git
cd rocBLAS
```

Replace `x.y` in the above command with the version of HIP SDK installed on your machine. For example, if you have HIP 5.5 installed, then use `-b release/rocm-rel-5.5`. You can add the SDK tools to your path with an entry like:

```
%HIP_PATH%\bin
```

3.3.3 Building

Below are steps to build using the `rmake.py` script. The user can install dependencies and build either:

- dependencies + library
- dependencies + library + client

You only need (dependencies + library) if you call rocBLAS from your code and only want the library built. The client contains testing and benchmark tools. `rmake.py` will print to the screen the full `cmake` command being used to configure rocBLAS based on your `rmake` command line options. This full `cmake` command can be used in your own build scripts if you want to bypass the python helper script for a fixed set of build options.

3.3.4 Library Dependencies

Dependencies installed by the python script rdeps.py are listed in the rdeps.xml configuration file. The -d flag passed to rmake.py installs dependencies the same as if running rdeps.py directly. Currently rdeps.py uses vcpkg and pip to install the build dependencies, with vcpkg being cloned into environment variable `VCPKG_PATH` or defaults into `C:\github\vcpkg`. pip will install into your current python3 environment.

The minimum version requirement for CMake is listed in the top level CMakeLists.txt file. CMake installed with Visual Studio 2022 meets this requirement. The vcpkg version tag is specified at the top of the rdeps.py file.

3.3.5 Build Library dependencies + Library

Common uses of rmake.py to build (library dependencies + library) are in the table below:

Command	Description
<code>./rmake.py -h</code>	Help information.
<code>./rmake.py -d</code>	Build library dependencies and library in your local directory. The -d flag only needs to be used once.
<code>./rmake.py</code>	Build library. It is assumed dependencies have been built.
<code>./rmake.py -i</code>	Build library, then build and install rocBLAS package. If you want to keep rocBLAS in your local tree, you do not need the -i flag.

3.3.6 Build Library Dependencies + Client Dependencies + Library + Client

Some client executables (.exe) are listed in the table below:

executable name	description
rocblas-test	runs Google Tests to test the library
rocblas-bench	executable to benchmark or test functions
rocblas-example-sscal	example C code calling rocblas_sscal function

Common uses of rmake.py to build (dependencies + library + client) are in the table below:

Command	Description
<code>./rmake.py -h</code>	Help information.
<code>./rmake.py -dc</code>	Build library dependencies, client dependencies, library, and client in your local directory. The d flag only needs to be used once. For subsequent invocations of rmake.py it is not necessary to rebuild the dependencies.
<code>./rmake.py -c</code>	Build library and client in your local directory. It is assumed the dependencies have been installed.
<code>./rmake.py -idc</code>	Build library dependencies, client dependencies, library, client, then build and install the rocBLAS package. If you want to keep rocBLAS in your local directory, you do not need the -i flag.
<code>./rmake.py -ic</code>	Build and install rocBLAS package, and build the client. If you want to keep rocBLAS in your local directory, you do not need the -i flag.

3.3.7 Build Clients without Library

The rocBLAS clients can be built on their own using *rmake.py* with a pre-existing rocBLAS library.

Note that the version of the rocBLAS clients being built should match the version of the installed rocBLAS. You can determine the version of the installed rocBLAS in the HIP SDK directory from the file `include\rocbblas\internal\rocbblas-version.h`. Find the version of rocBLAS being built if you have `grep` (e.g. in a git bash) with command `grep "VERSION_STRING" CMakeLists.txt` in the rocBLAS directory where you are building the clients.

Command	Description
<code>./rmake.py --clients-only</code>	Build rocBLAS clients and use an installed rocBLAS library at HIP_PATH if no --library-path specified
<code>./rmake.py --clients-only --library-path / path/to/rocbblas</code>	Build rocBLAS clients and use a rocBLAS library at the specified location.

API REFERENCE GUIDE

4.1 Introduction

rocBLAS is the AMD library for Basic Linear Algebra Subprograms (BLAS) on the [ROCm platform](#). It is implemented in the [HIP programming language](#) and optimized for AMD GPUs.

The aim of rocBLAS is to provide:

- Functionality similar to Legacy BLAS, adapted to run on GPUs
- High-performance robust implementation

rocBLAS is written in C++17 and HIP. It uses the AMD ROCm runtime to run on GPU devices.

The rocBLAS API is a thin C99 API using the Hourglass Pattern. It contains:

- [\[Level1\]](#), [\[Level2\]](#), and [\[Level3\]](#) BLAS functions, with batched and strided_batched versions
- Extensions to Legacy BLAS, including functions for mixed precision
- Auxiliary functions
- Device Memory functions

Note:

- The official rocBLAS API is the C99 API defined in `rocblas.h`. Therefore the use of any other public symbols is discouraged. All other C/C++ interfaces may not follow a deprecation model and so can change without warning from one release to the next.
 - rocBLAS array storage format is column major and one based. This is to maintain compatibility with the Legacy BLAS code, which is written in Fortran.
 - rocBLAS calls the AMD library [Tensile](#) for Level 3 BLAS matrix multiplication.
-

4.2 rocBLAS API and Legacy BLAS Functions

rocBLAS is initialized by calling `rocblas_create_handle`, and it is terminated by calling `rocblas_destroy_handle`. The `rocblas_handle` is persistent, and it contains:

- HIP stream
- Temporary device work space
- Mode for enabling or disabling logging (default is logging disabled)

rocBLAS functions run on the host, and they call HIP to launch rocBLAS kernels that run on the device in a HIP stream. The kernels are asynchronous unless:

- The function returns a scalar result from device to host
- Temporary device memory is allocated

In both cases above, the launch can be made asynchronous by:

- Use `rocblas_pointer_mode_device` to keep the scalar result on the device. Note that it is only the following Level1 BLAS functions that return a scalar result: `Xdot`, `Xdotu`, `Xnrm2`, `Xasum`, `iXamax`, `iXamin`.
- Use the provided device memory functions to allocate device memory that persists in the handle. Note that most rocBLAS functions do not allocate temporary device memory.

Before calling a rocBLAS function, arrays must be copied to the device. Integer scalars like `m`, `n`, `k` are stored on the host. Floating point scalars like `alpha` and `beta` can be on host or device.

Error handling is by returning a `rocblas_status`. Functions conform to the Legacy BLAS argument checking.

4.2.1 Rules for Obtaining rocBLAS API from Legacy BLAS

1. The Legacy BLAS routine name is changed to lowercase and prefixed by `rocblas_`. For example: Legacy BLAS routine `SSCAL`, scales a vector by a constant, is converted to `rocblas_sscal`.
2. A first argument `rocblas_handle` `handle` is added to all rocBLAS functions.
3. Input arguments are declared with the `const` modifier.
4. Character arguments are replaced with enumerated types defined in `rocblas_types.h`. They are passed by value on the host.
5. Array arguments are passed by reference on the device.
6. Scalar arguments are passed by value on the host with the following exceptions. See the section `Pointer Mode` for more information on these exceptions:
 - Scalar values `alpha` and `beta` are passed by reference on either the host or the device.
 - Where Legacy BLAS functions have return values, the return value is instead added as the last function argument. It is returned by reference on either the host or the device. This applies to the following functions: `xDOT`, `xDOTU`, `xNRM2`, `xASUM`, `IxAMAX`, `IxAMIN`.
7. The return value of all functions is `rocblas_status`, defined in `rocblas_types.h`. It is used to check for errors.

4.2.2 Example Code

Below is a simple example code for calling function `rocblas_sscal`:

```
#include <iostream>
#include <vector>
#include "hip/hip_runtime_api.h"
#include "rocblas.h"

using namespace std;

int main()
{
    rocblas_int n = 10240;
```

(continues on next page)

(continued from previous page)

```

float alpha = 10.0;

vector<float> hx(n);
vector<float> hz(n);
float* dx;

roclblas_handle handle;
roclblas_create_handle(&handle);

// allocate memory on device
hipMalloc(&dx, n * sizeof(float));

// Initial Data on CPU,
srand(1);
for( int i = 0; i < n; ++i )
{
    hx[i] = rand() % 10 + 1; //generate a integer number between [1, 10]
}

// copy array from host memory to device memory
hipMemcpy(dx, hx.data(), sizeof(float) * n, hipMemcpyHostToDevice);

// call rocBLAS function
roclblas_status status = roclblas_sscal(handle, n, &alpha, dx, 1);

// check status for errors
if(status == roclblas_status_success)
{
    cout << "status == roclblas_status_success" << endl;
}
else
{
    cout << "roclblas failure: status = " << status << endl;
}

// copy output from device memory to host memory
hipMemcpy(hx.data(), dx, sizeof(float) * n, hipMemcpyDeviceToHost);

hipFree(dx);
roclblas_destroy_handle(handle);
return 0;
}

```

4.2.3 LP64 Interface

The rocBLAS library is LP64, so rocblas_int arguments are 32 bit and rocblas_long arguments are 64 bit.

4.2.4 Column-major Storage and 1 Based Indexing

rocBLAS uses column-major storage for 2D arrays, and 1-based indexing for the functions xMAX and xMIN. This is the same as Legacy BLAS and cuBLAS.

If you need row-major and 0-based indexing (used in C language arrays), download the file cblas.tgz from the Netlib Repository. Look at the CBLAS functions that provide a thin interface to Legacy BLAS. They convert from row-major, 0 based, to column-major, 1 based. This is done by swapping the order of function arguments. It is not necessary to transpose matrices.

4.2.5 Pointer Mode

The auxiliary functions rocblas_set_pointer and rocblas_get_pointer are used to set and get the value of the state variable rocblas_pointer_mode. This variable is stored in rocblas_handle. If rocblas_pointer_mode == rocblas_pointer_mode_host, then scalar parameters must be allocated on the host. If rocblas_pointer_mode == rocblas_pointer_mode_device, then scalar parameters must be allocated on the device.

There are two types of scalar parameter:

- Scaling parameters like alpha and beta used in functions like axpy, gemv, gemm 2
- Scalar results from functions amax, amin, asum, dot, nrm2

For scalar parameters like alpha and beta when rocblas_pointer_mode == rocblas_pointer_mode_host, they can be allocated on the host heap or stack. The kernel launch is asynchronous, and if they are on the heap, they can be freed after the return from the kernel launch. When rocblas_pointer_mode == rocblas_pointer_mode_device they must not be changed till the kernel completes.

For scalar results, when rocblas_pointer_mode == rocblas_pointer_mode_host, then the function blocks the CPU till the GPU has copied the result back to the host. When rocblas_pointer_mode == rocblas_pointer_mode_device the function will return after the asynchronous launch. Similarly to vector and matrix results, the scalar result is only available when the kernel has completed execution.

4.2.6 Asynchronous API

rocBLAS functions will be asynchronous unless:

- The function needs to allocate device memory
- The function returns a scalar result from GPU to CPU

The order of operations in the asynchronous functions is as in the figure below. The argument checking, calculation of process grid, and kernel launch take very little time. The asynchronous kernel running on the GPU does not block the CPU. After the kernel launch, the CPU keeps processing the next instructions.

The above order of operations will change if there is logging or the function is synchronous. Logging requires system calls, and the program must wait for them to complete before executing the next instruction. See the Logging section for more information.

Note: The default is no logging.

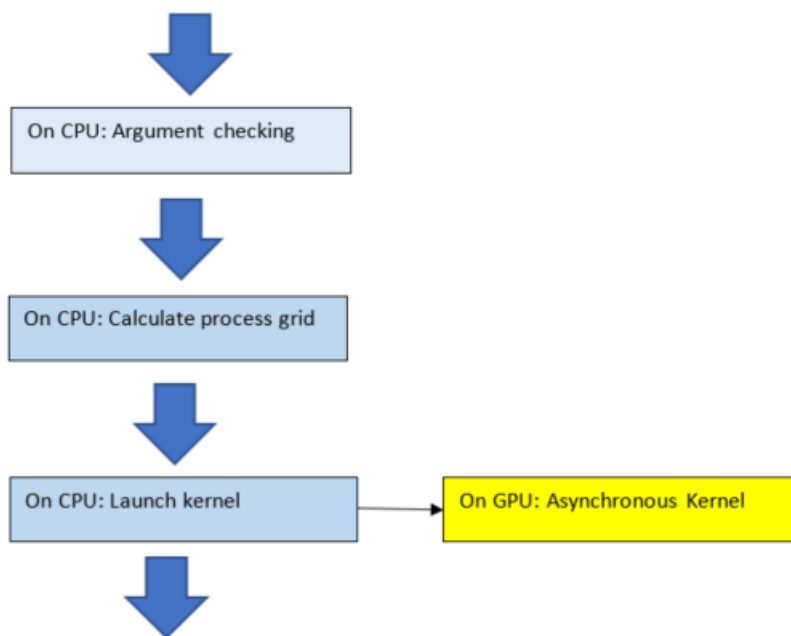


Fig. 4.1: Order of operations in asynchronous functions

If the cpu needs to allocate device memory, it must wait till this is complete before executing the next instruction. See the Device Memory Allocation section for more information.

Note: Memory can be preallocated. This will make the function asynchronous, as it removes the need for the function to allocate memory.

The following functions copy a scalar result from GPU to CPU if `rocblas_pointer_mode == rocblas_pointer_mode_host`: `asum`, `dot`, `max`, `min`, `nrm2`.

This makes the function synchronous, as the program must wait for the copy before executing the next instruction. See the section on Pointer Mode for more information.

Note: Set `rocblas_pointer_mode` to `rocblas_pointer_mode_device` makes the function asynchronous by keeping the result on the GPU.

The order of operations with logging, device memory allocation, and return of a scalar result is as in the figure below:

4.2.7 Complex Number Data Types

Data types for rocBLAS complex numbers in the API are a special case. For C compiler users, gcc, and other non-hipcc compiler users, these types are exposed as a struct with x and y components and identical memory layout to `std::complex` for float and double precision. Internally a templated C++ class is defined, but it should be considered deprecated for external use. For simplified usage with Hipified code there is an option to interpret the API as using `hipFloatComplex` and `hipDoubleComplex` types (i.e. `typedef hipFloatComplex rocblas_float_complex`). This is provided for users to avoid casting when using the hip complex types in their code. As the memory layout is consistent across all three types, it is safe to cast arguments to API calls between the 3 types: `hipFloatComplex`, `std::complex<float>`, and `rocblas_float_complex`, as well as for the double precision variants. To expose the API as using the hip defined complex types, user can use either a compiler define or inlined `#define ROCM_MATHLIBS_API_USE_HIP_COMPLEX`

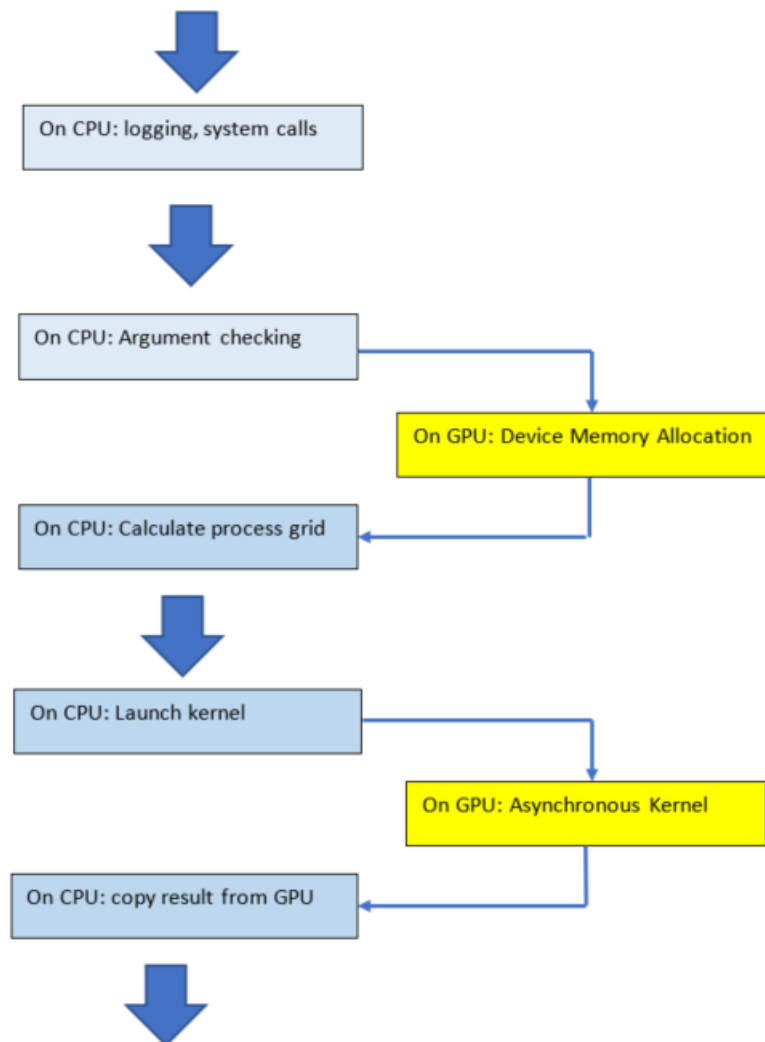


Fig. 4.2: Code blocks in synchronous function call

before including the header file `<roclblas.h>`. Thus the API is compatible with both forms, but recompilation is required to avoid casting if switching to pass in the hip complex types. Most device memory pointers are passed with `void*` types to hip utility functions (e.g. `hipMemcpy`), so uploading memory from `std::complex` arrays or `hipFloatComplex` arrays requires no changes regardless of complex data type API choice.

4.2.8 MI100 (gfx908) Considerations

On nodes with the MI100 (gfx908), MFMA (Matrix-Fused-Multiply-Add) instructions are available to substantially speed up matrix operations. This hardware feature is used in all gemm and gemm-based functions in rocBLAS with 32-bit or shorter base datatypes with an associated 32-bit `compute_type` (`f32_r`, `i32_r`, or `f32_c` as appropriate).

Specifically, rocBLAS takes advantage of MI100's MFMA instructions for three real base types `f16_r`, `bf16_r`, and `f32_r` with `compute_type f32_r`, one integral base type `i8_r` with `compute_type i32_r`, and one complex base type `f32_c` with `compute_type f32_c`. In summary, all GEMM APIs and APIs for GEMM-based functions using these five base types and their associated `compute_type` (explicit or implicit) take advantage of MI100's MFMA instructions.

Note: The use of MI100's MFMA instructions is automatic. There is no user control for on/off.

Not all problem sizes may select MFMA-based kernels; additional tuning may be needed to get good performance.

4.2.9 MI200 (gfx90a) Considerations

On nodes with the MI200 (gfx90a), MFMA_F64 instructions are available to substantially speed up double precision matrix operations. This hardware feature is used in all GEMM and GEMM-based functions in rocBLAS with 64-bit floating-point datatype, namely DGEMM, ZGEMM, DTRSM, ZTRSM, DTRMM, ZTRMM, DSYRKX, and ZSYRKX.

The MI200 MFMA_F16, MFMA_BF16 and MFMA_BF16_1K instructions flush subnormal input/output data ("denorms") to zero. It is observed that certain use cases utilizing the HPA (High Precision Accumulate) HGEMM kernels where `a_type=b_type=c_type=d_type=f16_r` and `compute_type=f32_r` do not tolerate the MI200's flush-denorms-to-zero behavior well due to F16's limited exponent range. An alternate implementation of the HPA HGEMM kernel utilizing the MFMA_BF16_1K instruction is provided which, takes advantage of BF16's much larger exponent range, albeit with reduced accuracy. To select the alternate implementation of HPA HGEMM with the `gemm_ex/gemm_strided_batched_ex` functions, for the `flags` argument, use the enum value of `roclblas_gemm_flags_fp16_alt_impl`.

Note: The use of MI200's MFMA instructions (including MFMA_F64) is automatic. There is no user control for on/off.

Not all problem sizes may select MFMA-based kernels; additional tuning may be needed to get good performance.

4.3 Deprecations by version

4.3.1 Announced in rocBLAS 2.45

Replace `is_complex` by `rocblas_is_complex`

From rocBLAS 3.0 the trait `is_complex` for rocblas complex types has been removed. Replace with `rocblas_is_complex`

Replace `truncate` with `rocblas_truncate`

From rocBLAS 3.0 enum `truncate_t` and the value `truncate` has been removed and replaced by `rocblas_truncate_t` and `rocblas_truncate`, respectively.

4.3.2 Announced in rocBLAS 2.46

Remove ability for hipBLAS to set `rocblas_int8_type_for_hipblas`

From rocBLAS 3.0 remove enum `rocblas_int8_type_for_hipblas` and the functions `rocblas_get_int8_type_for_hipblas` and `rocblas_set_int8_type_for_hipblas`. These are used by hipBLAS to select either `int8_t` or `packed_int8x4` datatype. In hipBLAS the option to use `packed_int8x4` will be removed, only `int8_t` will be available.

4.3.3 Announced in rocBLAS 3.0

Replace Legacy BLAS in-place trmm functions with trmm functions that support both in-place and out-of-place functionality

Use of the deprecated Legacy BLAS in-place trmm functions will give deprecation warnings telling you to compile with `-DROCBLAS_V3` and use the new in-place and out-of-place trmm functions.

Note that there are no deprecation warnings for the rocBLAS Fortran API.

The Legacy BLAS in-place trmm calculates $B \leftarrow \alpha * \text{op}(A) * B$. Matrix B is replaced in-place by triangular matrix A multiplied by matrix B. The prototype in the include file `rocblas-functions.h` is:

```
rocblas_status rocblas_strmm(rocblas_handle    handle,
                             rocblas_side      side,
                             rocblas_fill      uplo,
                             rocblas_operation transA,
                             rocblas_diagonal  diag,
                             rocblas_int       m,
                             rocblas_int       n,
                             const float*      alpha,
                             const float*      A,
                             rocblas_int       lda,
                             float*           B,
                             rocblas_int       ldb);
```

rocBLAS 3.0 deprecates the legacy BLAS trmm functionality and replaces it with $C \leftarrow \alpha * \text{op}(A) * B$. The prototype is:

```

rocblas_status rocblas_strmm(rocblas_handle    handle,
                             rocblas_side     side,
                             rocblas_fill     uplo,
                             rocblas_operation transA,
                             rocblas_diagonal diag,
                             rocblas_int      m,
                             rocblas_int      n,
                             const float*     alpha,
                             const float*     A,
                             rocblas_int      lda,
                             const float*     B,
                             rocblas_int      ldb,
                             float*          C,
                             rocblas_int      ldc);

```

The new API provides the legacy BLAS in-place functionality if you set pointer C equal to pointer B and ldc equal to ldb.

There are similar deprecations for the `_batched` and `_strided_batched` versions of `trmm`.

Remove rocblas_gemm_ext2

`rocblas_gemm_ext2` is deprecated and it will be removed in the next major release of rocBLAS.

Removal of rocblas_query_int8_layout_flag

`rocblas_query_int8_layout_flag` will be removed and support will end for the `rocblas_gemm_flags_pack_int8x4` enum in `rocblas_gemm_flags` in a future release. `rocblas_int8_type_for_hipblas` will remain until `rocblas_query_int8_layout_flag` is removed.

Remove user_managed mode from rocblas_handle

From rocBLAS 4.0, the schemes for allocating temporary device memory would be reduced to two from four.

Existing four schemes are:

- `rocblas_managed`
- `user_managed, preallocate`
- `user_managed, manual`
- `user_owned`

From rocBLAS 4.0, the two schemes would be `rocblas_managed` and `user_owned`. The functionality of `user_managed` (both `preallocate` and `manual`) would be combined into `rocblas_managed` scheme.

Due to this the following APIs would be affected:

- `rocblas_is_user_managing_device_memory()` will be removed.
- `rocblas_set_device_memory_size()` will be replaced by a future function `rocblas_increase_device_memory_size()`, this new API would allow users to increase the device memory pool size at runtime.

4.3.4 Announced in rocBLAS 3.1

Removal of `__STDC_WANT_IEC_60559_TYPES_EXT__` define

Prior to rocBLAS 4.0, `__STDC_WANT_IEC_60559_TYPES_EXT__` was defined in `rocblas.h`, or more specifically `rocblas-types.h`, before including `float.h`. From rocBLAS 4.0, this define will be removed. Users who want ISO/IEC TS 18661-3:2015 functionality must define `__STDC_WANT_IEC_60559_TYPES_EXT__` before including `float.h` and `rocblas.h`.

4.3.5 Removed in rocBLAS 4.0

`rocblas_gemm_ext2` removed

As announced earlier, `rocblas_gemm_ext2` was removed in 4.0.

USING ROCBLAS API

This section describes how to use the rocBLAS library API.

5.1 rocBLAS Datatypes

5.1.1 rocblas_handle

typedef struct _rocblas_handle **rocblas_handle**

rocblas_handle is a structure holding the rocblas library context. It must be initialized using *rocblas_create_handle()*, and the returned handle must be passed to all subsequent library function calls. It should be destroyed at the end using *rocblas_destroy_handle()*.

5.1.2 rocblas_int

typedef int32_t **rocblas_int**

To specify whether int32 is used for LP64 or int64 is used for ILP64.

5.1.3 rocblas_stride

typedef int64_t **rocblas_stride**

Stride between matrices or vectors in strided_batched functions.

5.1.4 rocblas_half

struct **rocblas_half**

Structure definition for *rocblas_half*.

5.1.5 rocblas_bfloat16

struct **rocblas_bfloat16**

Struct to represent a 16 bit Brain floating-point number.

5.1.6 rocblas_float_complex

struct **rocblas_float_complex**

Struct to represent a complex number with single precision real and imaginary parts.

5.1.7 rocblas_double_complex

struct **rocblas_double_complex**

Struct to represent a complex number with double precision real and imaginary parts.

5.2 rocBLAS Enumeration

Enumeration constants have numbering that is consistent with CBLAS, ACML, most standard C BLAS libraries

5.2.1 rocblas_operation

enum **rocblas_operation**

Used to specify whether the matrix is to be transposed or not.

Parameter constants. numbering is consistent with CBLAS, ACML and most standard C BLAS libraries

Values:

enumerator **rocblas_operation_none**

Operate with the matrix.

enumerator **rocblas_operation_transpose**

Operate with the transpose of the matrix.

enumerator **rocblas_operation_conjugate_transpose**

Operate with the conjugate transpose of the matrix.

5.2.2 rocblas_fill

enum **rocblas_fill**

Used by the Hermitian, symmetric and triangular matrix routines to specify whether the upper, or lower triangle is being referenced.

Values:

enumerator **rocblas_fill_upper**

Upper triangle.

enumerator **rocblas_fill_lower**

Lower triangle.

enumerator **rocblas_fill_full**

5.2.3 rocblas_diagonal

enum **rocblas_diagonal**

It is used by the triangular matrix routines to specify whether the matrix is unit triangular.

Values:

enumerator **rocblas_diagonal_non_unit**

Non-unit triangular.

enumerator **rocblas_diagonal_unit**

Unit triangular.

5.2.4 rocblas_side

enum **rocblas_side**

Indicates the side matrix A is located relative to matrix B during multiplication.

Values:

enumerator **rocblas_side_left**

Multiply general matrix by symmetric, Hermitian, or triangular matrix on the left.

enumerator **rocblas_side_right**

Multiply general matrix by symmetric, Hermitian, or triangular matrix on the right.

enumerator **rocblas_side_both**

5.2.5 rocblas_status

enum **rocblas_status**

rocblas status codes definition

Values:

enumerator **rocblas_status_success**

Success

enumerator **rocblas_status_invalid_handle**

Handle not initialized, invalid or null

enumerator **rocblas_status_not_implemented**

Function is not implemented

enumerator **rocblas_status_invalid_pointer**

Invalid pointer argument

enumerator **rocblas_status_invalid_size**

Invalid size argument

enumerator **rocblas_status_memory_error**

Failed internal memory allocation, copy or dealloc

enumerator **rocblas_status_internal_error**

Other internal library failure

enumerator **rocblas_status_perf_degraded**

Performance degraded due to low device memory

enumerator **rocblas_status_size_query_mismatch**

Unmatched start/stop size query

enumerator **rocblas_status_size_increased**

Queried device memory size increased

enumerator **rocblas_status_size_unchanged**

Queried device memory size unchanged

enumerator **rocblas_status_invalid_value**

Passed argument not valid

enumerator **rocblas_status_continue**

Nothing preventing function to proceed

enumerator **rocblas_status_check_numerics_fail**

Will be set if the vector/matrix has a NaN/Infinity/denormal value

5.2.6 rocblas_datatype

enum **rocblas_datatype**

Indicates the precision width of data stored in a blas type.

Parameter constants. Numbering continues into next free decimal range but not shared with other BLAS libraries

Values:

enumerator **rocblas_datatype_f16_r**

16-bit floating point, real

enumerator **rocblas_datatype_f32_r**

32-bit floating point, real

enumerator **rocblas_datatype_f64_r**

64-bit floating point, real

enumerator **rocblas_datatype_f16_c**

16-bit floating point, complex

enumerator **rocblas_datatype_f32_c**

32-bit floating point, complex

enumerator **rocblas_datatype_f64_c**

64-bit floating point, complex

enumerator **rocblas_datatype_i8_r**

8-bit signed integer, real

enumerator **rocblas_datatype_u8_r**

8-bit unsigned integer, real

enumerator **rocblas_datatype_i32_r**

32-bit signed integer, real

enumerator **rocblas_datatype_u32_r**

32-bit unsigned integer, real

enumerator **rocblas_datatype_i8_c**

8-bit signed integer, complex

enumerator **rocblas_datatype_u8_c**

8-bit unsigned integer, complex

enumerator **rocblas_datatype_i32_c**

32-bit signed integer, complex

enumerator **rocblas_datatype_u32_c**

32-bit unsigned integer, complex

enumerator **rocblas_datatype_bf16_r**

16-bit bfloat, real

enumerator **rocblas_datatype_bf16_c**

16-bit bfloat, complex

enumerator **rocblas_datatype_invalid**

Invalid datatype value, do not use

5.2.7 rocblas_pointer_mode

enum **rocblas_pointer_mode**

Indicates if scalar pointers are on host or device. This is used for scalars alpha and beta and for scalar function return values.

Values:

enumerator **rocblas_pointer_mode_host**

Scalar values affected by this variable are located on the host.

enumerator **rocblas_pointer_mode_device**

Scalar values affected by this variable are located on the device.

5.2.8 rocblas_atomics_mode

enum **rocblas_atomics_mode**

Indicates if atomics operations are allowed. Not allowing atomic operations may generally improve determinism and repeatability of results at a cost of performance.

Values:

enumerator **rocblas_atomics_not_allowed**

Algorithms will refrain from atomics where applicable.

enumerator **rocblas_atomics_allowed**

Algorithms will take advantage of atomics where applicable.

5.2.9 rocblas_layer_mode

enum **rocblas_layer_mode**

Indicates if layer is active with bitmask.

Values:

enumerator **rocblas_layer_mode_none**

No logging will take place.

enumerator **rocblas_layer_mode_log_trace**

A line containing the function name and value of arguments passed will be printed with each rocBLAS function call.

enumerator **rocblas_layer_mode_log_bench**

Outputs a line each time a rocBLAS function is called, this line can be used with rocblas-bench to make the same call again.

enumerator **rocblas_layer_mode_log_profile**

Outputs a YAML description of each rocBLAS function called, along with its arguments and number of times it was called.

5.2.10 rocblas_gemm_algo

enum **rocblas_gemm_algo**

Indicates if layer is active with bitmask.

Values:

enumerator **rocblas_gemm_algo_standard**

enumerator **rocblas_gemm_algo_solution_index**

5.2.11 rocblas_gemm_flags

enum **rocblas_gemm_flags**

Control flags passed into gemm algorithms invoked by Tensile Host.

Values:

enumerator **rocblas_gemm_flags_none**

Default empty flags.

enumerator **rocblas_gemm_flags_pack_int8x4**

Before ROCm 4.2, this flag is not implemented and rocblas uses packed-Int8x4 by default. After ROCm 4.2, set flag is necessary if we want packed-Int8x4. Default (0x0) uses unpacked. As of rocBLAS 3.0

in ROCm 5.6, `rocbblas_gemm_flags_pack_int8x4` is deprecated and support will be removed in a future release.

enumerator **`rocbblas_gemm_flags_use_cu_efficiency`**

Select the gemm problem with the highest efficiency per compute unit used. Useful for running multiple smaller problems simultaneously. This takes precedence over the performance metric set in `rocbblas_handle` and currently only works for `gemm_*_ex` problems.

enumerator **`rocbblas_gemm_flags_fp16_alt_impl`**

Select an alternate implementation for the MI200 FP16 HPA (High Precision Accumulate) GEMM kernel utilizing the BF16 matrix instructions with reduced accuracy in cases where computation cannot tolerate the FP16 matrix instructions flushing subnormal FP16 input/output data to zero. See the “MI200 (gfx90a) Considerations” section for more details.

enumerator **`rocbblas_gemm_flags_check_solution_index`**

enumerator **`rocbblas_gemm_flags_fp16_alt_impl_rnz`**

5.3 rocBLAS Helper functions

5.3.1 Auxiliary Functions

rocbblas_status **`rocbblas_create_handle`**(*rocbblas_handle* *handle)

Create handle.

rocbblas_status **`rocbblas_destroy_handle`**(*rocbblas_handle* handle)

Destroy handle.

rocbblas_status **`rocbblas_set_stream`**(*rocbblas_handle* handle, hipStream_t stream)

Set stream for handle.

rocbblas_status **`rocbblas_get_stream`**(*rocbblas_handle* handle, hipStream_t *stream)

Get stream [0] from handle.

rocbblas_status **`rocbblas_set_pointer_mode`**(*rocbblas_handle* handle, *rocbblas_pointer_mode* pointer_mode)

Set `rocbblas_pointer_mode`.

rocbblas_status **`rocbblas_get_pointer_mode`**(*rocbblas_handle* handle, *rocbblas_pointer_mode* *pointer_mode)

Get `rocbblas_pointer_mode`.

rocbblas_status **`rocbblas_set_atomics_mode`**(*rocbblas_handle* handle, *rocbblas_atomics_mode* atomics_mode)

Set `rocbblas_atomics_mode`.

rocbblas_status **`rocbblas_get_atomics_mode`**(*rocbblas_handle* handle, *rocbblas_atomics_mode* *atomics_mode)

Get `rocbblas_atomics_mode`.

rocbblas_status **`rocbblas_query_int8_layout_flag`**(*rocbblas_handle* handle, *rocbblas_gemm_flags* *flag)

Query the preferable supported int8 input layout for gemm.

Indicates the supported int8 input layout for gemm according to the device. If the device supports packed-int8x4 (1) only, output flag is `rocbblas_gemm_flags_pack_int8x4` and users must bitwise-or your flag with `rocbblas_gemm_flags_pack_int8x4`. If output flag is `rocbblas_gemm_flags_none` (0), then unpacked int8 is preferable and suggested.

Parameters

- **handle** – [in] [roclblas_handle] the handle of device
- **flag** – [out] pointer to roclblas_gemm_flags

roclblas_pointer_mode **roclblas_pointer_to_mode**(void *ptr)

Indicates whether the pointer is on the host or device.

roclblas_status **roclblas_set_vector**(*roclblas_int* n, *roclblas_int* elem_size, const void *x, *roclblas_int* incx, void *y, *roclblas_int* incy)

Copy vector from host to device.

Parameters

- **n** – [in] [roclblas_int] number of elements in the vector
- **elem_size** – [in] [roclblas_int] number of bytes per element in the matrix
- **x** – [in] pointer to vector on the host
- **incx** – [in] [roclblas_int] specifies the increment for the elements of the vector
- **y** – [out] pointer to vector on the device
- **incy** – [in] [roclblas_int] specifies the increment for the elements of the vector

roclblas_status **roclblas_get_vector**(*roclblas_int* n, *roclblas_int* elem_size, const void *x, *roclblas_int* incx, void *y, *roclblas_int* incy)

Copy vector from device to host.

Parameters

- **n** – [in] [roclblas_int] number of elements in the vector
- **elem_size** – [in] [roclblas_int] number of bytes per element in the matrix
- **x** – [in] pointer to vector on the device
- **incx** – [in] [roclblas_int] specifies the increment for the elements of the vector
- **y** – [out] pointer to vector on the host
- **incy** – [in] [roclblas_int] specifies the increment for the elements of the vector

roclblas_status **roclblas_set_matrix**(*roclblas_int* rows, *roclblas_int* cols, *roclblas_int* elem_size, const void *a, *roclblas_int* lda, void *b, *roclblas_int* ldb)

Copy matrix from host to device.

Parameters

- **rows** – [in] [roclblas_int] number of rows in matrices
- **cols** – [in] [roclblas_int] number of columns in matrices
- **elem_size** – [in] [roclblas_int] number of bytes per element in the matrix
- **a** – [in] pointer to matrix on the host
- **lda** – [in] [roclblas_int] specifies the leading dimension of A, lda >= rows
- **b** – [out] pointer to matrix on the GPU
- **ldb** – [in] [roclblas_int] specifies the leading dimension of B, ldb >= rows

rocbblas_status **rocbblas_get_matrix**(*rocbblas_int* rows, *rocbblas_int* cols, *rocbblas_int* elem_size, const void *a, *rocbblas_int* lda, void *b, *rocbblas_int* ldb)

Copy matrix from device to host.

Parameters

- **rows** – [in] [*rocbblas_int*] number of rows in matrices
- **cols** – [in] [*rocbblas_int*] number of columns in matrices
- **elem_size** – [in] [*rocbblas_int*] number of bytes per element in the matrix
- **a** – [in] pointer to matrix on the GPU
- **lda** – [in] [*rocbblas_int*] specifies the leading dimension of A, lda >= rows
- **b** – [out] pointer to matrix on the host
- **ldb** – [in] [*rocbblas_int*] specifies the leading dimension of B, ldb >= rows

rocbblas_status **rocbblas_set_vector_async**(*rocbblas_int* n, *rocbblas_int* elem_size, const void *x, *rocbblas_int* incx, void *y, *rocbblas_int* incy, hipStream_t stream)

Asynchronously copy vector from host to device.

rocbblas_set_vector_async copies a vector from pinned host memory to device memory asynchronously. Memory on the host must be allocated with hipHostMalloc or the transfer will be synchronous.

Parameters

- **n** – [in] [*rocbblas_int*] number of elements in the vector
- **elem_size** – [in] [*rocbblas_int*] number of bytes per element in the matrix
- **x** – [in] pointer to vector on the host
- **incx** – [in] [*rocbblas_int*] specifies the increment for the elements of the vector
- **y** – [out] pointer to vector on the device
- **incy** – [in] [*rocbblas_int*] specifies the increment for the elements of the vector
- **stream** – [in] specifies the stream into which this transfer request is queued

rocbblas_status **rocbblas_set_matrix_async**(*rocbblas_int* rows, *rocbblas_int* cols, *rocbblas_int* elem_size, const void *a, *rocbblas_int* lda, void *b, *rocbblas_int* ldb, hipStream_t stream)

Asynchronously copy matrix from host to device.

rocbblas_set_matrix_async copies a matrix from pinned host memory to device memory asynchronously. Memory on the host must be allocated with hipHostMalloc or the transfer will be synchronous.

Parameters

- **rows** – [in] [*rocbblas_int*] number of rows in matrices
- **cols** – [in] [*rocbblas_int*] number of columns in matrices
- **elem_size** – [in] [*rocbblas_int*] number of bytes per element in the matrix
- **a** – [in] pointer to matrix on the host
- **lda** – [in] [*rocbblas_int*] specifies the leading dimension of A, lda >= rows
- **b** – [out] pointer to matrix on the GPU
- **ldb** – [in] [*rocbblas_int*] specifies the leading dimension of B, ldb >= rows
- **stream** – [in] specifies the stream into which this transfer request is queued

rocbblas_status **rocbblas_get_matrix_async**(*rocbblas_int* rows, *rocbblas_int* cols, *rocbblas_int* elem_size, const void *a, *rocbblas_int* lda, void *b, *rocbblas_int* ldb, hipStream_t stream)

asynchronously copy matrix from device to host

rocbblas_get_matrix_async copies a matrix from device memory to pinned host memory asynchronously. Memory on the host must be allocated with hipHostMalloc or the transfer will be synchronous.

Parameters

- **rows** – [in] [rocbblas_int] number of rows in matrices
- **cols** – [in] [rocbblas_int] number of columns in matrices
- **elem_size** – [in] [rocbblas_int] number of bytes per element in the matrix
- **a** – [in] pointer to matrix on the GPU
- **lda** – [in] [rocbblas_int] specifies the leading dimension of A, lda >= rows
- **b** – [out] pointer to matrix on the host
- **ldb** – [in] [rocbblas_int] specifies the leading dimension of B, ldb >= rows
- **stream** – [in] specifies the stream into which this transfer request is queued

void **rocbblas_initialize**(void)

Initialize rocBLAS on the current HIP device, to avoid costly startup time at the first call on that device.

Calling *rocbblas_initialize()* allows upfront initialization including device specific kernel setup. Otherwise this function is automatically called on the first function call that requires these initializations (mainly GEMM).

const char ***rocbblas_status_to_string**(*rocbblas_status* status)

BLAS Auxiliary API

rocbblas_status_to_string

Returns string representing rocbblas_status value

Parameters

- **status** – [in] [rocbblas_status] rocBLAS status to convert to string

5.3.2 Device Memory Allocation Functions

rocbblas_status **rocbblas_start_device_memory_size_query**(*rocbblas_handle* handle)

Indicates that subsequent rocBLAS kernel calls should collect the optimal device memory size in bytes for their given kernel arguments and keep track of the maximum. Each kernel call can reuse temporary device memory on the same stream so the maximum is collected. Returns rocbblas_status_size_query_mismatch if another size query is already in progress; returns rocbblas_status_success otherwise

Parameters

- **handle** – [in] rocbblas handle

rocbblas_status **rocbblas_stop_device_memory_size_query**(*rocbblas_handle* handle, size_t *size)

Stops collecting optimal device memory size information. Returns rocbblas_status_size_query_mismatch if a collection is not underway; rocbblas_status_invalid_handle if handle is nullptr; rocbblas_status_invalid_pointer if size is nullptr; rocbblas_status_success otherwise

Parameters

- **handle** – [in] rocbblas handle
- **size** – [out] maximum of the optimal sizes collected

rocblas_status **rocblas_get_device_memory_size**(*rocblas_handle* handle, size_t *size)

Gets the current device memory size for the handle. Returns `rocblas_status_invalid_handle` if handle is nullptr; `rocblas_status_invalid_pointer` if size is nullptr; `rocblas_status_success` otherwise

Parameters

- **handle** – [in] rocblas handle
- **size** – [out] current device memory size for the handle

rocblas_status **rocblas_set_device_memory_size**(*rocblas_handle* handle, size_t size)

Changes the size of allocated device memory at runtime.

Any previously allocated device memory managed by the handle is freed.

If size > 0 sets the device memory size to the specified size (in bytes). If size == 0, frees the memory allocated so far, and lets rocBLAS manage device memory in the future, expanding it when necessary. Returns `rocblas_status_invalid_handle` if handle is nullptr; `rocblas_status_invalid_pointer` if size is nullptr; `rocblas_status_success` otherwise

Parameters

- **handle** – [in] rocblas handle
- **size** – [in] size of allocated device memory

rocblas_status **rocblas_set_workspace**(*rocblas_handle* handle, void *addr, size_t size)

Sets the device workspace for the handle to use.

Any previously allocated device memory managed by the handle is freed.

Returns `rocblas_status_invalid_handle` if handle is nullptr; `rocblas_status_success` otherwise

Parameters

- **handle** – [in] rocblas handle
- **addr** – [in] address of workspace memory
- **size** – [in] size of workspace memory

bool **rocblas_is_managing_device_memory**(*rocblas_handle* handle)

Returns true when device memory in handle is managed by rocBLAS

Parameters

handle – [in] rocblas handle

bool **rocblas_is_user_managing_device_memory**(*rocblas_handle* handle)

Returns true when device memory in handle is managed by the user

Parameters

handle – [in] rocblas handle

For more detailed information, refer to sections *rocBLAS Beta Features* and *Device Memory Allocation*.

5.3.3 Build Information Functions

rocblas_status **rocblas_get_version_string_size**(size_t *len)

Queries the minimum buffer size for a successful call to *rocblas_get_version_string*.

Parameters

len – [out] pointer to size_t for storing the length

rocblas_status **rocblas_get_version_string**(char *buf, size_t len)

Loads char* buf with the rocblas library version. size_t len is the maximum length of char* buf.

Parameters

- **buf** – [inout] pointer to buffer for version string
- **len** – [in] length of buf

5.4 rocBLAS Level-1 functions

5.4.1 rocblas_iXamax + batched, strided_batched

rocblas_status **rocblas_isamax**(*rocblas_handle* handle, *rocblas_int* n, const float *x, *rocblas_int* incx, *rocblas_int* *result)

rocblas_status **rocblas_idamax**(*rocblas_handle* handle, *rocblas_int* n, const double *x, *rocblas_int* incx, *rocblas_int* *result)

rocblas_status **rocblas_icamax**(*rocblas_handle* handle, *rocblas_int* n, const *rocblas_float_complex* *x, *rocblas_int* incx, *rocblas_int* *result)

rocblas_status **rocblas_izamax**(*rocblas_handle* handle, *rocblas_int* n, const *rocblas_double_complex* *x, *rocblas_int* incx, *rocblas_int* *result)

BLAS Level 1 API

amax finds the first index of the element of maximum magnitude of a vector x.

Parameters

- **handle** – [in] [*rocblas_handle*] handle to the rocblas library context queue.
- **n** – [in] [*rocblas_int*] the number of elements in x.
- **x** – [in] device pointer storing vector x.
- **incx** – [in] [*rocblas_int*] specifies the increment for the elements of y.
- **result** – [inout] device pointer or host pointer to store the amax index. return is 0.0 if n, incx<=0.

rocblas_status **rocblas_isamax_batched**(*rocblas_handle* handle, *rocblas_int* n, const float *const x[], *rocblas_int* incx, *rocblas_int* batch_count, *rocblas_int* *result)

rocblas_status **rocblas_idamax_batched**(*rocblas_handle* handle, *rocblas_int* n, const double *const x[], *rocblas_int* incx, *rocblas_int* batch_count, *rocblas_int* *result)

rocblas_status **rocblas_icamax_batched**(*rocblas_handle* handle, *rocblas_int* n, const *rocblas_float_complex* *const x[], *rocblas_int* incx, *rocblas_int* batch_count, *rocblas_int* *result)

```
rocblas_status rocblas_izamax_batched(rocblas_handle handle, rocblas_int n, const rocblas_double_complex
                                     *const x[], rocblas_int incx, rocblas_int batch_count, rocblas_int
                                     *result)
```

BLAS Level 1 API

amax_batched finds the first index of the element of maximum magnitude of each vector x_i in a batch, for $i = 1, \dots, \text{batch_count}$.

Parameters

- **handle** – [in] [*rocblas_handle*] handle to the rocblas library context queue.
- **n** – [in] [*rocblas_int*] number of elements in each vector x_i .
- **x** – [in] device array of device pointers storing each vector x_i .
- **incx** – [in] [*rocblas_int*] specifies the increment for the elements of each x_i . incx must be > 0 .
- **batch_count** – [in] [*rocblas_int*] number of instances in the batch. Must be > 0 .
- **result** – [out] device or host array of pointers of batch_count size for results. return is 0 if $n, \text{incx} \leq 0$.

```
rocblas_status rocblas_isamax_strided_batched(rocblas_handle handle, rocblas_int n, const float *x,
                                              rocblas_int incx, rocblas_stride stridex, rocblas_int
                                              batch_count, rocblas_int *result)
```

```
rocblas_status rocblas_idamax_strided_batched(rocblas_handle handle, rocblas_int n, const double *x,
                                              rocblas_int incx, rocblas_stride stridex, rocblas_int
                                              batch_count, rocblas_int *result)
```

```
rocblas_status rocblas_icamax_strided_batched(rocblas_handle handle, rocblas_int n, const
                                              rocblas_float_complex *x, rocblas_int incx, rocblas_stride
                                              stridex, rocblas_int batch_count, rocblas_int *result)
```

```
rocblas_status rocblas_izamax_strided_batched(rocblas_handle handle, rocblas_int n, const
                                              rocblas_double_complex *x, rocblas_int incx, rocblas_stride
                                              stridex, rocblas_int batch_count, rocblas_int *result)
```

BLAS Level 1 API

amax_strided_batched finds the first index of the element of maximum magnitude of each vector x_i in a batch, for $i = 1, \dots, \text{batch_count}$.

Parameters

- **handle** – [in] [*rocblas_handle*] handle to the rocblas library context queue.
- **n** – [in] [*rocblas_int*] number of elements in each vector x_i .
- **x** – [in] device pointer to the first vector x_1 .
- **incx** – [in] [*rocblas_int*] specifies the increment for the elements of each x_i . incx must be > 0 .
- **stridex** – [in] [*rocblas_stride*] specifies the pointer increment between one x_i and the next $x_{(i+1)}$.
- **batch_count** – [in] [*rocblas_int*] number of instances in the batch.
- **result** – [out] device or host pointer for storing contiguous batch_count results. return is 0 if $n \leq 0, \text{incx} \leq 0$.

5.4.2 rocblas_iXamin + batched, strided_batched

rocblas_status **rocblas_isamin**(*rocblas_handle* handle, *rocblas_int* n, const float *x, *rocblas_int* incx, *rocblas_int* *result)

rocblas_status **rocblas_idamin**(*rocblas_handle* handle, *rocblas_int* n, const double *x, *rocblas_int* incx, *rocblas_int* *result)

rocblas_status **rocblas_icamin**(*rocblas_handle* handle, *rocblas_int* n, const *rocblas_float_complex* *x, *rocblas_int* incx, *rocblas_int* *result)

rocblas_status **rocblas_izamin**(*rocblas_handle* handle, *rocblas_int* n, const *rocblas_double_complex* *x, *rocblas_int* incx, *rocblas_int* *result)

BLAS Level 1 API

amin finds the first index of the element of minimum magnitude of a vector x.

Parameters

- **handle** – [in] [*rocblas_handle*] handle to the rocblas library context queue.
- **n** – [in] [*rocblas_int*] the number of elements in x.
- **x** – [in] device pointer storing vector x.
- **incx** – [in] [*rocblas_int*] specifies the increment for the elements of y.
- **result** – [inout] device pointer or host pointer to store the amin index. return is 0.0 if n, incx <= 0.

rocblas_status **rocblas_isamin_batched**(*rocblas_handle* handle, *rocblas_int* n, const float *const x[], *rocblas_int* incx, *rocblas_int* batch_count, *rocblas_int* *result)

rocblas_status **rocblas_idamin_batched**(*rocblas_handle* handle, *rocblas_int* n, const double *const x[], *rocblas_int* incx, *rocblas_int* batch_count, *rocblas_int* *result)

rocblas_status **rocblas_icamin_batched**(*rocblas_handle* handle, *rocblas_int* n, const *rocblas_float_complex* *const x[], *rocblas_int* incx, *rocblas_int* batch_count, *rocblas_int* *result)

rocblas_status **rocblas_izamin_batched**(*rocblas_handle* handle, *rocblas_int* n, const *rocblas_double_complex* *const x[], *rocblas_int* incx, *rocblas_int* batch_count, *rocblas_int* *result)

BLAS Level 1 API

amin_batched finds the first index of the element of minimum magnitude of each vector x_i in a batch, for i = 1, ..., batch_count.

Parameters

- **handle** – [in] [*rocblas_handle*] handle to the rocblas library context queue.
- **n** – [in] [*rocblas_int*] number of elements in each vector x_i.
- **x** – [in] device array of device pointers storing each vector x_i.
- **incx** – [in] [*rocblas_int*] specifies the increment for the elements of each x_i. incx must be > 0.
- **batch_count** – [in] [*rocblas_int*] number of instances in the batch. Must be > 0.

- **result** – [out] device or host pointers to array of batch_count size for results. return is 0 if n, incx<=0.

rocblas_status **rocblas_isamin_strided_batched**(*rocblas_handle* handle, *rocblas_int* n, const float *x, *rocblas_int* incx, *rocblas_stride* stridex, *rocblas_int* batch_count, *rocblas_int* *result)

rocblas_status **rocblas_idamin_strided_batched**(*rocblas_handle* handle, *rocblas_int* n, const double *x, *rocblas_int* incx, *rocblas_stride* stridex, *rocblas_int* batch_count, *rocblas_int* *result)

rocblas_status **rocblas_icamin_strided_batched**(*rocblas_handle* handle, *rocblas_int* n, const *rocblas_float_complex* *x, *rocblas_int* incx, *rocblas_stride* stridex, *rocblas_int* batch_count, *rocblas_int* *result)

rocblas_status **rocblas_izamin_strided_batched**(*rocblas_handle* handle, *rocblas_int* n, const *rocblas_double_complex* *x, *rocblas_int* incx, *rocblas_stride* stridex, *rocblas_int* batch_count, *rocblas_int* *result)

BLAS Level 1 API

amin_strided_batched finds the first index of the element of minimum magnitude of each vector x_i in a batch, for i = 1, ..., batch_count.

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **n** – [in] [rocblas_int] number of elements in each vector x_i.
- **x** – [in] device pointer to the first vector x_1.
- **incx** – [in] [rocblas_int] specifies the increment for the elements of each x_i. incx must be > 0.
- **stridex** – [in] [rocblas_stride] specifies the pointer increment between one x_i and the next x_(i + 1).
- **batch_count** – [in] [rocblas_int] number of instances in the batch.
- **result** – [out] device or host pointer to array for storing contiguous batch_count results. return is 0 if n <= 0, incx<=0.

5.4.3 rocblas_Xasum + batched, strided_batched

rocblas_status **rocblas_sasum**(*rocblas_handle* handle, *rocblas_int* n, const float *x, *rocblas_int* incx, float *result)

rocblas_status **rocblas_dasum**(*rocblas_handle* handle, *rocblas_int* n, const double *x, *rocblas_int* incx, double *result)

rocblas_status **rocblas_scasum**(*rocblas_handle* handle, *rocblas_int* n, const *rocblas_float_complex* *x, *rocblas_int* incx, float *result)

rocblas_status **rocblas_dzasum**(*rocblas_handle* handle, *rocblas_int* n, const *rocblas_double_complex* *x, *rocblas_int* incx, double *result)

BLAS Level 1 API

asum computes the sum of the magnitudes of elements of a real vector x, or the sum of magnitudes of the real and imaginary parts of elements if x is a complex vector.

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **n** – [in] [rocblas_int] the number of elements in x and y.
- **x** – [in] device pointer storing vector x.
- **incx** – [in] [rocblas_int] specifies the increment for the elements of x. incx must be > 0.
- **result** – [inout] device pointer or host pointer to store the asum product. return is 0.0 if n <= 0.

rocblas_status **rocblas_sasum_batched**(*rocblas_handle* handle, *rocblas_int* n, const float *const x[], *rocblas_int* incx, *rocblas_int* batch_count, float *results)

rocblas_status **rocblas_dasum_batched**(*rocblas_handle* handle, *rocblas_int* n, const double *const x[], *rocblas_int* incx, *rocblas_int* batch_count, double *results)

rocblas_status **rocblas_scasum_batched**(*rocblas_handle* handle, *rocblas_int* n, const *rocblas_float_complex* *const x[], *rocblas_int* incx, *rocblas_int* batch_count, float *results)

rocblas_status **rocblas_dzasum_batched**(*rocblas_handle* handle, *rocblas_int* n, const *rocblas_double_complex* *const x[], *rocblas_int* incx, *rocblas_int* batch_count, double *results)

BLAS Level 1 API

asum_batched computes the sum of the magnitudes of the elements in a batch of real vectors x_i, or the sum of magnitudes of the real and imaginary parts of elements if x_i is a complex vector, for i = 1, ..., batch_count.

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **n** – [in] [rocblas_int] number of elements in each vector x_i.
- **x** – [in] device array of device pointers storing each vector x_i.
- **incx** – [in] [rocblas_int] specifies the increment for the elements of each x_i. incx must be > 0.
- **results** – [out] device array or host array of batch_count size for results. return is 0.0 if n, incx <= 0.
- **batch_count** – [in] [rocblas_int] number of instances in the batch.

rocblas_status **rocblas_sasum_strided_batched**(*rocblas_handle* handle, *rocblas_int* n, const float *x, *rocblas_int* incx, *rocblas_stride* stridex, *rocblas_int* batch_count, float *results)

rocblas_status **rocblas_dasum_strided_batched**(*rocblas_handle* handle, *rocblas_int* n, const double *x, *rocblas_int* incx, *rocblas_stride* stridex, *rocblas_int* batch_count, double *results)

rocblas_status **rocblas_scasum_strided_batched**(*rocblas_handle* handle, *rocblas_int* n, const *rocblas_float_complex* *x, *rocblas_int* incx, *rocblas_stride* stridex, *rocblas_int* batch_count, float *results)

rocblas_status **rocblas_dzasum_strided_batched**(*rocblas_handle* handle, *rocblas_int* n, const *rocblas_double_complex* *x, *rocblas_int* incx, *rocblas_stride* stridex, *rocblas_int* batch_count, double *results)

BLAS Level 1 API

asum_strided_batched computes the sum of the magnitudes of elements of a real vectors x_i, or the sum of magnitudes of the real and imaginary parts of elements if x_i is a complex vector, for i = 1, ..., batch_count.

Parameters

- **handle** – [in] [rocbas_handle] handle to the rocbas library context queue.
- **n** – [in] [rocbas_int] number of elements in each vector x_i .
- **x** – [in] device pointer to the first vector x_1 .
- **incx** – [in] [rocbas_int] specifies the increment for the elements of each x_i . incx must be > 0 .
- **stridex** – [in] [rocbas_stride] stride from the start of one vector (x_i) and the next one (x_{i+1}). There are no restrictions placed on stride_x. However, ensure that stride_x is of appropriate size. For a typical case this means $\text{stride_x} \geq n * \text{incx}$.
- **results** – [out] device pointer or host pointer to array for storing contiguous batch_count results. return is 0.0 if $n, \text{incx} \leq 0$.
- **batch_count** – [in] [rocbas_int] number of instances in the batch.

5.4.4 rocbas_Xaxpy + batched, strided_batched

rocbas_status **rocbas_saxpy**(*rocbas_handle* handle, *rocbas_int* n, const float *alpha, const float *x, *rocbas_int* incx, float *y, *rocbas_int* incy)

rocbas_status **rocbas_daxpy**(*rocbas_handle* handle, *rocbas_int* n, const double *alpha, const double *x, *rocbas_int* incx, double *y, *rocbas_int* incy)

rocbas_status **rocbas_haxpy**(*rocbas_handle* handle, *rocbas_int* n, const *rocbas_half* *alpha, const *rocbas_half* *x, *rocbas_int* incx, *rocbas_half* *y, *rocbas_int* incy)

rocbas_status **rocbas_caxpy**(*rocbas_handle* handle, *rocbas_int* n, const *rocbas_float_complex* *alpha, const *rocbas_float_complex* *x, *rocbas_int* incx, *rocbas_float_complex* *y, *rocbas_int* incy)

rocbas_status **rocbas_zaxpy**(*rocbas_handle* handle, *rocbas_int* n, const *rocbas_double_complex* *alpha, const *rocbas_double_complex* *x, *rocbas_int* incx, *rocbas_double_complex* *y, *rocbas_int* incy)

BLAS Level 1 API

axpy computes constant alpha multiplied by vector x, plus vector y:

$$y := \text{alpha} * x + y$$
Parameters

- **handle** – [in] [rocbas_handle] handle to the rocbas library context queue.
- **n** – [in] [rocbas_int] the number of elements in x and y.
- **alpha** – [in] device pointer or host pointer to specify the scalar alpha.
- **x** – [in] device pointer storing vector x.
- **incx** – [in] [rocbas_int] specifies the increment for the elements of x.
- **y** – [out] device pointer storing vector y.
- **incy** – [inout] [rocbas_int] specifies the increment for the elements of y.

```
rocblas_status rocblas_saxpy_batched(rocblas_handle handle, rocblas_int n, const float *alpha, const float
                                     *const x[], rocblas_int incx, float *const y[], rocblas_int incy,
                                     rocblas_int batch_count)
```

```
rocblas_status rocblas_daxpy_batched(rocblas_handle handle, rocblas_int n, const double *alpha, const double
                                     *const x[], rocblas_int incx, double *const y[], rocblas_int incy,
                                     rocblas_int batch_count)
```

```
rocblas_status rocblas_haxpy_batched(rocblas_handle handle, rocblas_int n, const rocblas_half *alpha, const
                                     rocblas_half *const x[], rocblas_int incx, rocblas_half *const y[],
                                     rocblas_int incy, rocblas_int batch_count)
```

```
rocblas_status rocblas_caxpy_batched(rocblas_handle handle, rocblas_int n, const rocblas_float_complex
                                     *alpha, const rocblas_float_complex *const x[], rocblas_int incx,
                                     rocblas_float_complex *const y[], rocblas_int incy, rocblas_int
                                     batch_count)
```

```
rocblas_status rocblas_zaxpy_batched(rocblas_handle handle, rocblas_int n, const rocblas_double_complex
                                     *alpha, const rocblas_double_complex *const x[], rocblas_int incx,
                                     rocblas_double_complex *const y[], rocblas_int incy, rocblas_int
                                     batch_count)
```

BLAS Level 1 API

axpy_batched compute $y := \alpha * x + y$ over a set of batched vectors.

Parameters

- **handle** – [in] rocblas_handle handle to the rocblas library context queue.
- **n** – [in] rocblas_int
- **alpha** – [in] specifies the scalar alpha.
- **x** – [in] pointer storing vector x on the GPU.
- **incx** – [in] rocblas_int specifies the increment for the elements of x.
- **y** – [out] pointer storing vector y on the GPU.
- **incy** – [inout] rocblas_int specifies the increment for the elements of y.
- **batch_count** – [in] rocblas_int number of instances in the batch.

```
rocblas_status rocblas_saxpy_strided_batched(rocblas_handle handle, rocblas_int n, const float *alpha, const
                                              float *x, rocblas_int incx, rocblas_stride stridex, float *y,
                                              rocblas_int incy, rocblas_stride stridey, rocblas_int
                                              batch_count)
```

```
rocblas_status rocblas_daxpy_strided_batched(rocblas_handle handle, rocblas_int n, const double *alpha,
                                              const double *x, rocblas_int incx, rocblas_stride stridex,
                                              double *y, rocblas_int incy, rocblas_stride stridey, rocblas_int
                                              batch_count)
```

```
rocblas_status rocblas_haxpy_strided_batched(rocblas_handle handle, rocblas_int n, const rocblas_half
                                              *alpha, const rocblas_half *x, rocblas_int incx,
                                              rocblas_stride stridex, rocblas_half *y, rocblas_int incy,
                                              rocblas_stride stridey, rocblas_int batch_count)
```

```
rocblas_status rocblas_caxpy_strided_batched(rocblas_handle handle, rocblas_int n, const
rocblas_float_complex *alpha, const rocblas_float_complex
*x, rocblas_int incx, rocblas_stride stridex,
rocblas_float_complex *y, rocblas_int incy, rocblas_stride
stridey, rocblas_int batch_count)
```

```
rocblas_status rocblas_zaxpy_strided_batched(rocblas_handle handle, rocblas_int n, const
rocblas_double_complex *alpha, const
rocblas_double_complex *x, rocblas_int incx, rocblas_stride
stridex, rocblas_double_complex *y, rocblas_int incy,
rocblas_stride stridey, rocblas_int batch_count)
```

BLAS Level 1 API

`axpy_strided_batched` compute $y := \alpha * x + y$ over a set of strided batched vectors.

Parameters

- **handle** – [in] `rocblas_handle` handle to the rocblas library context queue.
- **n** – [in] `rocblas_int`.
- **alpha** – [in] specifies the scalar alpha.
- **x** – [in] pointer storing vector x on the GPU.
- **incx** – [in] `rocblas_int` specifies the increment for the elements of x.
- **stridex** – [in] `rocblas_stride` specifies the increment between vectors of x.
- **y** – [out] pointer storing vector y on the GPU.
- **incy** – [inout] `rocblas_int` specifies the increment for the elements of y.
- **stridey** – [in] `rocblas_stride` specifies the increment between vectors of y.
- **batch_count** – [in] `rocblas_int` number of instances in the batch.

5.4.5 rocblas_Xcopy + batched, strided_batched

```
rocblas_status rocblas_scopy(rocblas_handle handle, rocblas_int n, const float *x, rocblas_int incx, float *y,
rocblas_int incy)
```

```
rocblas_status rocblas_dcopy(rocblas_handle handle, rocblas_int n, const double *x, rocblas_int incx, double *y,
rocblas_int incy)
```

```
rocblas_status rocblas_ccopy(rocblas_handle handle, rocblas_int n, const rocblas_float_complex *x, rocblas_int
incx, rocblas_float_complex *y, rocblas_int incy)
```

```
rocblas_status rocblas_zcopy(rocblas_handle handle, rocblas_int n, const rocblas_double_complex *x,
rocblas_int incx, rocblas_double_complex *y, rocblas_int incy)
```

BLAS Level 1 API

`copy` copies each element $x[i]$ into $y[i]$, for $i = 1, \dots, n$:

```
y := x
```

Parameters

- **handle** – [in] [`rocblas_handle`] handle to the rocblas library context queue.

- **n** – [in] [rocblas_int] the number of elements in x to be copied to y.
- **x** – [in] device pointer storing vector x.
- **incx** – [in] [rocblas_int] specifies the increment for the elements of x.
- **y** – [out] device pointer storing vector y.
- **incy** – [in] [rocblas_int] specifies the increment for the elements of y.

rocblas_status **rocblas_scopy_batched**(*rocblas_handle* handle, *rocblas_int* n, const float *const x[], *rocblas_int* incx, float *const y[], *rocblas_int* incy, *rocblas_int* batch_count)

rocblas_status **rocblas_dcopy_batched**(*rocblas_handle* handle, *rocblas_int* n, const double *const x[], *rocblas_int* incx, double *const y[], *rocblas_int* incy, *rocblas_int* batch_count)

rocblas_status **rocblas_ccopy_batched**(*rocblas_handle* handle, *rocblas_int* n, const *rocblas_float_complex* *const x[], *rocblas_int* incx, *rocblas_float_complex* *const y[], *rocblas_int* incy, *rocblas_int* batch_count)

rocblas_status **rocblas_zcopy_batched**(*rocblas_handle* handle, *rocblas_int* n, const *rocblas_double_complex* *const x[], *rocblas_int* incx, *rocblas_double_complex* *const y[], *rocblas_int* incy, *rocblas_int* batch_count)

BLAS Level 1 API

copy_batched copies each element $x_i[j]$ into $y_i[j]$, for $j = 1, \dots, n$; $i = 1, \dots, \text{batch_count}$:

$y_i := x_i$,
 where (x_i, y_i) is the i -th instance of the batch.
 x_i and y_i are vectors.

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **n** – [in] [rocblas_int] the number of elements in each x_i to be copied to y_i .
- **x** – [in] device array of device pointers storing each vector x_i .
- **incx** – [in] [rocblas_int] specifies the increment for the elements of each vector x_i .
- **y** – [out] device array of device pointers storing each vector y_i .
- **incy** – [in] [rocblas_int] specifies the increment for the elements of each vector y_i .
- **batch_count** – [in] [rocblas_int] number of instances in the batch.

rocblas_status **rocblas_scopy_strided_batched**(*rocblas_handle* handle, *rocblas_int* n, const float *x, *rocblas_int* incx, *rocblas_stride* stridex, float *y, *rocblas_int* incy, *rocblas_stride* stridey, *rocblas_int* batch_count)

rocblas_status **rocblas_dcopy_strided_batched**(*rocblas_handle* handle, *rocblas_int* n, const double *x, *rocblas_int* incx, *rocblas_stride* stridex, double *y, *rocblas_int* incy, *rocblas_stride* stridey, *rocblas_int* batch_count)

rocblas_status **rocblas_ccopy_strided_batched**(*rocblas_handle* handle, *rocblas_int* n, const *rocblas_float_complex* *x, *rocblas_int* incx, *rocblas_stride* stridex, *rocblas_float_complex* *y, *rocblas_int* incy, *rocblas_stride* stridey, *rocblas_int* batch_count)

```
rocblas_status rocblas_zcopy_strided_batched(rocblas_handle handle, rocblas_int n, const
                                             rocblas_double_complex *x, rocblas_int incx, rocblas_stride
                                             stridex, rocblas_double_complex *y, rocblas_int incy,
                                             rocblas_stride stridey, rocblas_int batch_count)
```

BLAS Level 1 API

copy_strided_batched copies each element $x_i[j]$ into $y_i[j]$, for $j = 1, \dots, n$; $i = 1, \dots, \text{batch_count}$:

```
y_i := x_i,
where (x_i, y_i) is the i-th instance of the batch.
x_i and y_i are vectors.
```

Parameters

- **handle** – [in] [*rocblas_handle*] handle to the rocblas library context queue.
- **n** – [in] [*rocblas_int*] the number of elements in each x_i to be copied to y_i .
- **x** – [in] device pointer to the first vector (x_1) in the batch.
- **incx** – [in] [*rocblas_int*] specifies the increments for the elements of vectors x_i .
- **stridex** – [in] [*rocblas_stride*] stride from the start of one vector (x_i) and the next one (x_{i+1}). There are no restrictions placed on stride_x. However, the user should take care to ensure that stride_x is of appropriate size. For a typical case, this means $\text{stride_x} \geq n * \text{incx}$.
- **y** – [out] device pointer to the first vector (y_1) in the batch.
- **incy** – [in] [*rocblas_int*] specifies the increment for the elements of vectors y_i .
- **stridey** – [in] [*rocblas_stride*] stride from the start of one vector (y_i) and the next one (y_{i+1}). There are no restrictions placed on stride_y. However, ensure that stride_y is of appropriate size, for a typical case this means $\text{stride_y} \geq n * \text{incy}$. stridey should be non zero.
- **batch_count** – [in] [*rocblas_int*] number of instances in the batch.

5.4.6 rocblas_Xdot + batched, strided_batched

```
rocblas_status rocblas_sdot(rocblas_handle handle, rocblas_int n, const float *x, rocblas_int incx, const float *y,
                             rocblas_int incy, float *result)
```

```
rocblas_status rocblas_ddot(rocblas_handle handle, rocblas_int n, const double *x, rocblas_int incx, const
                             double *y, rocblas_int incy, double *result)
```

```
rocblas_status rocblas_hdot(rocblas_handle handle, rocblas_int n, const rocblas_half *x, rocblas_int incx, const
                             rocblas_half *y, rocblas_int incy, rocblas_half *result)
```

```
rocblas_status rocblas_bfdot(rocblas_handle handle, rocblas_int n, const rocblas_bfloat16 *x, rocblas_int incx,
                             const rocblas_bfloat16 *y, rocblas_int incy, rocblas_bfloat16 *result)
```

```
rocblas_status rocblas_cdotu(rocblas_handle handle, rocblas_int n, const rocblas_float_complex *x, rocblas_int
                             incx, const rocblas_float_complex *y, rocblas_int incy, rocblas_float_complex
                             *result)
```

```
rocblas_status rocblas_cdotc(rocblas_handle handle, rocblas_int n, const rocblas_float_complex *x, rocblas_int incx, const rocblas_float_complex *y, rocblas_int incy, rocblas_float_complex *result)
```

```
rocblas_status rocblas_zdotu(rocblas_handle handle, rocblas_int n, const rocblas_double_complex *x, rocblas_int incx, const rocblas_double_complex *y, rocblas_int incy, rocblas_double_complex *result)
```

```
rocblas_status rocblas_zdotc(rocblas_handle handle, rocblas_int n, const rocblas_double_complex *x, rocblas_int incx, const rocblas_double_complex *y, rocblas_int incy, rocblas_double_complex *result)
```

BLAS Level 1 API

dot(u) performs the dot product of vectors x and y:

```
result = x * y;
```

dotc performs the dot product of the conjugate of complex vector x and complex vector y.

```
result = conjugate (x) * y;
```

Parameters

- **handle** – [in] [*rocblas_handle*] handle to the rocblas library context queue.
- **n** – [in] [*rocblas_int*] the number of elements in x and y.
- **x** – [in] device pointer storing vector x.
- **incx** – [in] [*rocblas_int*] specifies the increment for the elements of y.
- **y** – [in] device pointer storing vector y.
- **incy** – [in] [*rocblas_int*] specifies the increment for the elements of y.
- **result** – [inout] device pointer or host pointer to store the dot product. return is 0.0 if n <= 0.

```
rocblas_status rocblas_sdot_batched(rocblas_handle handle, rocblas_int n, const float *const x[], rocblas_int incx, const float *const y[], rocblas_int incy, rocblas_int batch_count, float *result)
```

```
rocblas_status rocblas_ddot_batched(rocblas_handle handle, rocblas_int n, const double *const x[], rocblas_int incx, const double *const y[], rocblas_int incy, rocblas_int batch_count, double *result)
```

```
rocblas_status rocblas_hdot_batched(rocblas_handle handle, rocblas_int n, const rocblas_half *const x[], rocblas_int incx, const rocblas_half *const y[], rocblas_int incy, rocblas_int batch_count, rocblas_half *result)
```

```
rocblas_status rocblas_bfdot_batched(rocblas_handle handle, rocblas_int n, const rocblas_bfloat16 *const x[], rocblas_int incx, const rocblas_bfloat16 *const y[], rocblas_int incy, rocblas_int batch_count, rocblas_bfloat16 *result)
```

```
rocblas_status rocblas_cdotu_batched(rocblas_handle handle, rocblas_int n, const rocblas_float_complex *const x[], rocblas_int incx, const rocblas_float_complex *const y[], rocblas_int incy, rocblas_int batch_count, rocblas_float_complex *result)
```



```
roclblas_status roclblas_cdotc_batched(roclblas_handle handle, roclblas_int n, const roclblas_float_complex
*const x[], roclblas_int incx, const roclblas_float_complex *const y[],
roclblas_int incy, roclblas_int batch_count, roclblas_float_complex
*result)
```

```
roclblas_status roclblas_zdotu_batched(roclblas_handle handle, roclblas_int n, const roclblas_double_complex
*const x[], roclblas_int incx, const roclblas_double_complex *const y[],
roclblas_int incy, roclblas_int batch_count, roclblas_double_complex
*result)
```

```
roclblas_status roclblas_zdotc_batched(roclblas_handle handle, roclblas_int n, const roclblas_double_complex
*const x[], roclblas_int incx, const roclblas_double_complex *const y[],
roclblas_int incy, roclblas_int batch_count, roclblas_double_complex
*result)
```

BLAS Level 1 API

dot_batched(u) performs a batch of dot products of vectors x and y:

```
result_i = x_i * y_i;
```

dotc_batched performs a batch of dot products of the conjugate of complex vector x and complex vector y

```
result_i = conjugate (x_i) * y_i;
where (x_i, y_i) is the i-th instance of the batch.
x_i and y_i are vectors, for i = 1, ..., batch_count.
```

Parameters

- **handle** – [in] [*roclblas_handle*] handle to the roclblas library context queue.
- **n** – [in] [*roclblas_int*] the number of elements in each x_i and y_i.
- **x** – [in] device array of device pointers storing each vector x_i.
- **incx** – [in] [*roclblas_int*] specifies the increment for the elements of each x_i.
- **y** – [in] device array of device pointers storing each vector y_i.
- **incy** – [in] [*roclblas_int*] specifies the increment for the elements of each y_i.
- **batch_count** – [in] [*roclblas_int*] number of instances in the batch.
- **result** – [inout] device array or host array of batch_count size to store the dot products of each batch. return 0.0 for each element if n <= 0.

```
roclblas_status roclblas_sdot_strided_batched(roclblas_handle handle, roclblas_int n, const float *x,
roclblas_int incx, roclblas_stride stridex, const float *y,
roclblas_int incy, roclblas_stride stridey, roclblas_int
batch_count, float *result)
```

```
roclblas_status roclblas_ddot_strided_batched(roclblas_handle handle, roclblas_int n, const double *x,
roclblas_int incx, roclblas_stride stridex, const double *y,
roclblas_int incy, roclblas_stride stridey, roclblas_int
batch_count, double *result)
```

```
roclblas_status roclblas_hdot_strided_batched(roclblas_handle handle, roclblas_int n, const roclblas_half *x,
roclblas_int incx, roclblas_stride stridex, const roclblas_half *y,
roclblas_int incy, roclblas_stride stridey, roclblas_int
batch_count, roclblas_half *result)
```



```

rocblas_status rocblas_bfdot_strided_batched(rocblas_handle handle, rocblas_int n, const rocblas_bfloat16
*x, rocblas_int incx, rocblas_stride stridex, const
rocblas_bfloat16 *y, rocblas_int incy, rocblas_stride stridey,
rocblas_int batch_count, rocblas_bfloat16 *result)

rocblas_status rocblas_cdotu_strided_batched(rocblas_handle handle, rocblas_int n, const
rocblas_float_complex *x, rocblas_int incx, rocblas_stride
stridex, const rocblas_float_complex *y, rocblas_int incy,
rocblas_stride stridey, rocblas_int batch_count,
rocblas_float_complex *result)

rocblas_status rocblas_cdotc_strided_batched(rocblas_handle handle, rocblas_int n, const
rocblas_float_complex *x, rocblas_int incx, rocblas_stride
stridex, const rocblas_float_complex *y, rocblas_int incy,
rocblas_stride stridey, rocblas_int batch_count,
rocblas_float_complex *result)

rocblas_status rocblas_zdotu_strided_batched(rocblas_handle handle, rocblas_int n, const
rocblas_double_complex *x, rocblas_int incx, rocblas_stride
stridex, const rocblas_double_complex *y, rocblas_int incy,
rocblas_stride stridey, rocblas_int batch_count,
rocblas_double_complex *result)

rocblas_status rocblas_zdotc_strided_batched(rocblas_handle handle, rocblas_int n, const
rocblas_double_complex *x, rocblas_int incx, rocblas_stride
stridex, const rocblas_double_complex *y, rocblas_int incy,
rocblas_stride stridey, rocblas_int batch_count,
rocblas_double_complex *result)

```

BLAS Level 1 API

dot_strided_batched(u) performs a batch of dot products of vectors x and y:

```
result_i = x_i * y_i;
```

dotc_strided_batched performs a batch of dot products of the conjugate of complex vector x and complex vector y

```
result_i = conjugate(x_i) * y_i;
where (x_i, y_i) is the i-th instance of the batch.
x_i and y_i are vectors, for i = 1, ..., batch_count.
```

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **n** – [in] [rocblas_int] the number of elements in each x_i and y_i.
- **x** – [in] device pointer to the first vector (x_1) in the batch.
- **incx** – [in] [rocblas_int] specifies the increment for the elements of each x_i.
- **stridex** – [in] [rocblas_stride] stride from the start of one vector (x_i) and the next one (x_i+1).
- **y** – [in] device pointer to the first vector (y_1) in the batch.
- **incy** – [in] [rocblas_int] specifies the increment for the elements of each y_i.

- **stridey** – [in] [rocblas_stride] stride from the start of one vector (y_i) and the next one (y_{i+1}).
- **batch_count** – [in] [rocblas_int] number of instances in the batch.
- **result** – [inout] device array or host array of batch_count size to store the dot products of each batch. return 0.0 for each element if $n \leq 0$.

5.4.7 rocblas_Xnrm2 + batched, strided_batched

rocblas_status **rocblas_snrm2**(*rocblas_handle* handle, *rocblas_int* n, const float *x, *rocblas_int* incx, float *result)

rocblas_status **rocblas_dnrm2**(*rocblas_handle* handle, *rocblas_int* n, const double *x, *rocblas_int* incx, double *result)

rocblas_status **rocblas_scnrm2**(*rocblas_handle* handle, *rocblas_int* n, const *rocblas_float_complex* *x, *rocblas_int* incx, float *result)

rocblas_status **rocblas_dznrm2**(*rocblas_handle* handle, *rocblas_int* n, const *rocblas_double_complex* *x, *rocblas_int* incx, double *result)

BLAS Level 1 API

nrm2 computes the euclidean norm of a real or complex vector:

```
result := sqrt( x'*x ) for real vectors
result := sqrt( x**H*x ) for complex vectors
```

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **n** – [in] [rocblas_int] the number of elements in x.
- **x** – [in] device pointer storing vector x.
- **incx** – [in] [rocblas_int] specifies the increment for the elements of y.
- **result** – [inout] device pointer or host pointer to store the nrm2 product. return is 0.0 if n, incx ≤ 0.

rocblas_status **rocblas_snrm2_batched**(*rocblas_handle* handle, *rocblas_int* n, const float *const x[], *rocblas_int* incx, *rocblas_int* batch_count, float *results)

rocblas_status **rocblas_dnrm2_batched**(*rocblas_handle* handle, *rocblas_int* n, const double *const x[], *rocblas_int* incx, *rocblas_int* batch_count, double *results)

rocblas_status **rocblas_scnrm2_batched**(*rocblas_handle* handle, *rocblas_int* n, const *rocblas_float_complex* *const x[], *rocblas_int* incx, *rocblas_int* batch_count, float *results)

rocblas_status **rocblas_dznrm2_batched**(*rocblas_handle* handle, *rocblas_int* n, const *rocblas_double_complex* *const x[], *rocblas_int* incx, *rocblas_int* batch_count, double *results)

BLAS Level 1 API

nrm2_batched computes the euclidean norm over a batch of real or complex vectors:

```
result := sqrt( x_i'*x_i ) for real vectors x, for i = 1, ..., batch_count
result := sqrt( x_i**H*x_i ) for complex vectors x, for i = 1, ..., batch_count
```

Parameters

- **handle** – [in] [rocbblas_handle] handle to the rocbblas library context queue.
- **n** – [in] [rocbblas_int] number of elements in each x_i .
- **x** – [in] device array of device pointers storing each vector x_i .
- **incx** – [in] [rocbblas_int] specifies the increment for the elements of each x_i . incx must be > 0.
- **batch_count** – [in] [rocbblas_int] number of instances in the batch.
- **results** – [out] device pointer or host pointer to array of batch_count size for nrm2 results. return is 0.0 for each element if $n \leq 0$, $incx \leq 0$.

rocbblas_status **rocbblas_snrm2_strided_batched**(*rocbblas_handle* handle, *rocbblas_int* n, const float *x, *rocbblas_int* incx, *rocbblas_stride* stridex, *rocbblas_int* batch_count, float *results)

rocbblas_status **rocbblas_dnrm2_strided_batched**(*rocbblas_handle* handle, *rocbblas_int* n, const double *x, *rocbblas_int* incx, *rocbblas_stride* stridex, *rocbblas_int* batch_count, double *results)

rocbblas_status **rocbblas_scnrm2_strided_batched**(*rocbblas_handle* handle, *rocbblas_int* n, const *rocbblas_float_complex* *x, *rocbblas_int* incx, *rocbblas_stride* stridex, *rocbblas_int* batch_count, float *results)

rocbblas_status **rocbblas_dznrm2_strided_batched**(*rocbblas_handle* handle, *rocbblas_int* n, const *rocbblas_double_complex* *x, *rocbblas_int* incx, *rocbblas_stride* stridex, *rocbblas_int* batch_count, double *results)

BLAS Level 1 API

nrm2_strided_batched computes the euclidean norm over a batch of real or complex vectors:

```
result := sqrt( x_i'*x_i ) for real vectors x, for i = 1, ..., batch_count
result := sqrt( x_i**H*x_i ) for complex vectors, for i = 1, ..., batch_count
```

Parameters

- **handle** – [in] [rocbblas_handle] handle to the rocbblas library context queue.
- **n** – [in] [rocbblas_int] number of elements in each x_i .
- **x** – [in] device pointer to the first vector x_1 .
- **incx** – [in] [rocbblas_int] specifies the increment for the elements of each x_i . incx must be > 0.
- **stridex** – [in] [rocbblas_stride] stride from the start of one vector (x_i) and the next one (x_{i+1}). There are no restrictions placed on stride_x. However, ensure that stride_x is of appropriate size. For a typical case this means $stride_x \geq n * incx$.
- **batch_count** – [in] [rocbblas_int] number of instances in the batch.
- **results** – [out] device pointer or host pointer to array for storing contiguous batch_count results. return is 0.0 for each element if $n \leq 0$, $incx \leq 0$.

5.4.8 rocblas_Xrot + batched, strided_batched

rocblas_status **rocblas_srot**(*rocblas_handle* handle, *rocblas_int* n, float *x, *rocblas_int* incx, float *y, *rocblas_int* incy, const float *c, const float *s)

rocblas_status **rocblas_drot**(*rocblas_handle* handle, *rocblas_int* n, double *x, *rocblas_int* incx, double *y, *rocblas_int* incy, const double *c, const double *s)

rocblas_status **rocblas_crot**(*rocblas_handle* handle, *rocblas_int* n, *rocblas_float_complex* *x, *rocblas_int* incx, *rocblas_float_complex* *y, *rocblas_int* incy, const float *c, const *rocblas_float_complex* *s)

rocblas_status **rocblas_csrot**(*rocblas_handle* handle, *rocblas_int* n, *rocblas_float_complex* *x, *rocblas_int* incx, *rocblas_float_complex* *y, *rocblas_int* incy, const float *c, const float *s)

rocblas_status **rocblas_zrot**(*rocblas_handle* handle, *rocblas_int* n, *rocblas_double_complex* *x, *rocblas_int* incx, *rocblas_double_complex* *y, *rocblas_int* incy, const double *c, const *rocblas_double_complex* *s)

rocblas_status **rocblas_zdrot**(*rocblas_handle* handle, *rocblas_int* n, *rocblas_double_complex* *x, *rocblas_int* incx, *rocblas_double_complex* *y, *rocblas_int* incy, const double *c, const double *s)

BLAS Level 1 API

rot applies the Givens rotation matrix defined by $c=\cos(\alpha)$ and $s=\sin(\alpha)$ to vectors x and y. Scalars c and s may be stored in either host or device memory. Location is specified by calling rocblas_set_pointer_mode.

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **n** – [in] [rocblas_int] number of elements in the x and y vectors.
- **x** – [inout] device pointer storing vector x.
- **incx** – [in] [rocblas_int] specifies the increment between elements of x.
- **y** – [inout] device pointer storing vector y.
- **incy** – [in] [rocblas_int] specifies the increment between elements of y.
- **c** – [in] device pointer or host pointer storing scalar cosine component of the rotation matrix.
- **s** – [in] device pointer or host pointer storing scalar sine component of the rotation matrix.

rocblas_status **rocblas_srot_batched**(*rocblas_handle* handle, *rocblas_int* n, float *const x[], *rocblas_int* incx, float *const y[], *rocblas_int* incy, const float *c, const float *s, *rocblas_int* batch_count)

rocblas_status **rocblas_drot_batched**(*rocblas_handle* handle, *rocblas_int* n, double *const x[], *rocblas_int* incx, double *const y[], *rocblas_int* incy, const double *c, const double *s, *rocblas_int* batch_count)

rocblas_status **rocblas_crot_batched**(*rocblas_handle* handle, *rocblas_int* n, *rocblas_float_complex* *const x[], *rocblas_int* incx, *rocblas_float_complex* *const y[], *rocblas_int* incy, const float *c, const *rocblas_float_complex* *s, *rocblas_int* batch_count)

rocblas_status **rocblas_csrot_batched**(*rocblas_handle* handle, *rocblas_int* n, *rocblas_float_complex* *const x[], *rocblas_int* incx, *rocblas_float_complex* *const y[], *rocblas_int* incy, const float *c, const float *s, *rocblas_int* batch_count)

```
rocblas_status rocblas_zrot_batched(rocblas_handle handle, rocblas_int n, rocblas_double_complex *const
x[], rocblas_int incx, rocblas_double_complex *const y[], rocblas_int
incy, const double *c, const rocblas_double_complex *s, rocblas_int
batch_count)
```

```
rocblas_status rocblas_zdrot_batched(rocblas_handle handle, rocblas_int n, rocblas_double_complex *const
x[], rocblas_int incx, rocblas_double_complex *const y[], rocblas_int
incy, const double *c, const double *s, rocblas_int batch_count)
```

BLAS Level 1 API

rot_batched applies the Givens rotation matrix defined by $c=\cos(\alpha)$ and $s=\sin(\alpha)$ to batched vectors x_i and y_i , for $i = 1, \dots, \text{batch_count}$. Scalars c and s may be stored in either host or device memory. Location is specified by calling rocblas_set_pointer_mode.

Parameters

- **handle** – [in] [*rocblas_handle*] handle to the rocblas library context queue.
- **n** – [in] [*rocblas_int*] number of elements in each x_i and y_i vectors.
- **x** – [inout] device array of device pointers storing each vector x_i .
- **incx** – [in] [*rocblas_int*] specifies the increment between elements of each x_i .
- **y** – [inout] device array of device pointers storing each vector y_i .
- **incy** – [in] [*rocblas_int*] specifies the increment between elements of each y_i .
- **c** – [in] device pointer or host pointer to scalar cosine component of the rotation matrix.
- **s** – [in] device pointer or host pointer to scalar sine component of the rotation matrix.
- **batch_count** – [in] [*rocblas_int*] the number of x and y arrays, i.e. the number of batches.

```
rocblas_status rocblas_srot_strided_batched(rocblas_handle handle, rocblas_int n, float *x, rocblas_int incx,
rocblas_stride stride_x, float *y, rocblas_int incy,
rocblas_stride stride_y, const float *c, const float *s,
rocblas_int batch_count)
```

```
rocblas_status rocblas_drot_strided_batched(rocblas_handle handle, rocblas_int n, double *x, rocblas_int
incx, rocblas_stride stride_x, double *y, rocblas_int incy,
rocblas_stride stride_y, const double *c, const double *s,
rocblas_int batch_count)
```

```
rocblas_status rocblas_crot_strided_batched(rocblas_handle handle, rocblas_int n, rocblas_float_complex
*x, rocblas_int incx, rocblas_stride stride_x,
rocblas_float_complex *y, rocblas_int incy, rocblas_stride
stride_y, const float *c, const rocblas_float_complex *s,
rocblas_int batch_count)
```

```
rocblas_status rocblas_csrot_strided_batched(rocblas_handle handle, rocblas_int n, rocblas_float_complex
*x, rocblas_int incx, rocblas_stride stride_x,
rocblas_float_complex *y, rocblas_int incy, rocblas_stride
stride_y, const float *c, const float *s, rocblas_int
batch_count)
```

```
rocblas_status rocblas_zrot_strided_batched(rocblas_handle handle, rocblas_int n, rocblas_double_complex
*x, rocblas_int incx, rocblas_stride stride_x,
rocblas_double_complex *y, rocblas_int incy, rocblas_stride
stride_y, const double *c, const rocblas_double_complex *s,
rocblas_int batch_count)
```

```
rocblas_status rocblas_zdrot_strided_batched(rocblas_handle handle, rocblas_int n,
                                             rocblas_double_complex *x, rocblas_int incx, rocblas_stride
                                             stride_x, rocblas_double_complex *y, rocblas_int incy,
                                             rocblas_stride stride_y, const double *c, const double *s,
                                             rocblas_int batch_count)
```

BLAS Level 1 API

rot_strided_batched applies the Givens rotation matrix defined by $c=\cos(\alpha)$ and $s=\sin(\alpha)$ to strided batched vectors x_i and y_i , for $i = 1, \dots, \text{batch_count}$. Scalars c and s may be stored in either host or device memory, location is specified by calling `rocblas_set_pointer_mode`.

Parameters

- **handle** – [in] [`rocblas_handle`] handle to the rocblas library context queue.
- **n** – [in] [`rocblas_int`] number of elements in each x_i and y_i vectors.
- **x** – [inout] device pointer to the first vector x_1 .
- **incx** – [in] [`rocblas_int`] specifies the increment between elements of each x_i .
- **stride_x** – [in] [`rocblas_stride`] specifies the increment from the beginning of x_i to the beginning of $x_{(i+1)}$.
- **y** – [inout] device pointer to the first vector y_1 .
- **incy** – [in] [`rocblas_int`] specifies the increment between elements of each y_i .
- **stride_y** – [in] [`rocblas_stride`] specifies the increment from the beginning of y_i to the beginning of $y_{(i+1)}$.
- **c** – [in] device pointer or host pointer to scalar cosine component of the rotation matrix.
- **s** – [in] device pointer or host pointer to scalar sine component of the rotation matrix.
- **batch_count** – [in] [`rocblas_int`] the number of x and y arrays, i.e. the number of batches.

5.4.9 rocblas_Xrotg + batched, strided_batched

```
rocblas_status rocblas_srotg(rocblas_handle handle, float *a, float *b, float *c, float *s)
```

```
rocblas_status rocblas_drotg(rocblas_handle handle, double *a, double *b, double *c, double *s)
```

```
rocblas_status rocblas_crotg(rocblas_handle handle, rocblas_float_complex *a, rocblas_float_complex *b, float
                             *c, rocblas_float_complex *s)
```

```
rocblas_status rocblas_zrotg(rocblas_handle handle, rocblas_double_complex *a, rocblas_double_complex *b,
                             double *c, rocblas_double_complex *s)
```

BLAS Level 1 API

rotg creates the Givens rotation matrix for the vector (a b). Scalars a , b , c , and s may be stored in either host or device memory, location is specified by calling `rocblas_set_pointer_mode`. The computation uses the formulas

```
sigma = sgn(a)    if |a| > |b|
      = sgn(b)    if |b| >= |a|
r = sigma*sqrt( a**2 + b**2 )
c = 1; s = 0      if r = 0
c = a/r; s = b/r if r != 0
```

The subroutine also computes

```
z = s    if |a| > |b|,
    = 1/c if |b| >= |a| and c != 0
    = 1   if c = 0
```

This allows c and s to be reconstructed from z as follows:

```
If z = 1, set c = 0, s = 1.
If |z| < 1, set c = sqrt(1 - z**2) and s = z.
If |z| > 1, set c = 1/z and s = sqrt(1 - c**2).
```

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **a** – [inout] pointer to a, an element in vector (a,b), overwritten with r.
- **b** – [inout] pointer to b, an element in vector (a,b), overwritten with z.
- **c** – [out] pointer to c, cosine element of Givens rotation.
- **s** – [out] pointer to s, sine element of Givens rotation.

rocblas_status **rocblas_srotg_batched**(*rocblas_handle* handle, float *const a[], float *const b[], float *const c[], float *const s[], *rocblas_int* batch_count)

rocblas_status **rocblas_drotg_batched**(*rocblas_handle* handle, double *const a[], double *const b[], double *const c[], double *const s[], *rocblas_int* batch_count)

rocblas_status **rocblas_crotg_batched**(*rocblas_handle* handle, *rocblas_float_complex* *const a[], *rocblas_float_complex* *const b[], float *const c[], *rocblas_float_complex* *const s[], *rocblas_int* batch_count)

rocblas_status **rocblas_zrotg_batched**(*rocblas_handle* handle, *rocblas_double_complex* *const a[], *rocblas_double_complex* *const b[], double *const c[], *rocblas_double_complex* *const s[], *rocblas_int* batch_count)

BLAS Level 1 API

rotg_batched creates the Givens rotation matrix for the batched vectors (a_i b_i), for i = 1, ..., batch_count. a, b, c, and s are host pointers to an array of device pointers on the device, where each device pointer points to a scalar value of a_i, b_i, c_i, or s_i.

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **a** – [inout] a, overwritten with r.
- **b** – [inout] b overwritten with z.
- **c** – [out] cosine element of Givens rotation for the batch.
- **s** – [out] sine element of Givens rotation for the batch.
- **batch_count** – [in] [rocblas_int] number of batches (length of arrays a, b, c, and s).

rocblas_status **rocblas_srotg_strided_batched**(*rocblas_handle* handle, float *a, *rocblas_stride* stride_a, float *b, *rocblas_stride* stride_b, float *c, *rocblas_stride* stride_c, float *s, *rocblas_stride* stride_s, *rocblas_int* batch_count)

```
rocblas_status rocblas_drotg_strided_batched(rocblas_handle handle, double *a, rocblas_stride stride_a,
double *b, rocblas_stride stride_b, double *c, rocblas_stride
stride_c, double *s, rocblas_stride stride_s, rocblas_int
batch_count)
```

```
rocblas_status rocblas_crotg_strided_batched(rocblas_handle handle, rocblas_float_complex *a,
rocblas_stride stride_a, rocblas_float_complex *b,
rocblas_stride stride_b, float *c, rocblas_stride stride_c,
rocblas_float_complex *s, rocblas_stride stride_s, rocblas_int
batch_count)
```

```
rocblas_status rocblas_zrotg_strided_batched(rocblas_handle handle, rocblas_double_complex *a,
rocblas_stride stride_a, rocblas_double_complex *b,
rocblas_stride stride_b, double *c, rocblas_stride stride_c,
rocblas_double_complex *s, rocblas_stride stride_s,
rocblas_int batch_count)
```

BLAS Level 1 API

rotg_strided_batched creates the Givens rotation matrix for the strided batched vectors (a_i b_i), for $i = 1, \dots, \text{batch_count}$. a , b , c , and s are host pointers to arrays a , b , c , s on the device.

Parameters

- **handle** – [in] [*rocblas_handle*] handle to the rocblas library context queue.
- **a** – [inout] host pointer to first single input vector element a_1 on the device, overwritten with r .
- **stride_a** – [in] [*rocblas_stride*] distance between elements of a in batch (distance between a_i and $a_{(i+1)}$).
- **b** – [inout] host pointer to first single input vector element b_1 on the device, overwritten with z .
- **stride_b** – [in] [*rocblas_stride*] distance between elements of b in batch (distance between b_i and $b_{(i+1)}$).
- **c** – [out] host pointer to first single cosine element of Givens rotations c_1 on the device.
- **stride_c** – [in] [*rocblas_stride*] distance between elements of c in batch (distance between c_i and $c_{(i+1)}$).
- **s** – [out] host pointer to first single sine element of Givens rotations s_1 on the device.
- **stride_s** – [in] [*rocblas_stride*] distance between elements of s in batch (distance between s_i and $s_{(i+1)}$).
- **batch_count** – [in] [*rocblas_int*] number of batches (length of arrays a , b , c , and s).

5.4.10 rocblas_Xrotm + batched, strided_batched

```
rocblas_status rocblas_srotm(rocblas_handle handle, rocblas_int n, float *x, rocblas_int incx, float *y,
rocblas_int incy, const float *param)
```

```
rocblas_status rocblas_drotm(rocblas_handle handle, rocblas_int n, double *x, rocblas_int incx, double *y,
rocblas_int incy, const double *param)
```

BLAS Level 1 API

rotm applies the modified Givens rotation matrix defined by param to vectors x and y .

Parameters

- **handle** – [in] [rocbblas_handle] handle to the rocbblas library context queue.
- **n** – [in] [rocbblas_int] number of elements in the x and y vectors.
- **x** – [inout] device pointer storing vector x.
- **incx** – [in] [rocbblas_int] specifies the increment between elements of x.
- **y** – [inout] device pointer storing vector y.
- **incy** – [in] [rocbblas_int] specifies the increment between elements of y.
- **param** – [in] device vector or host vector of 5 elements defining the rotation.

```
param[0] = flag
param[1] = H11
param[2] = H21
param[3] = H12
param[4] = H22
```

The flag parameter defines the form of H:

```
flag = -1 => H = ( H11 H12 H21 H22 )
flag =  0 => H = ( 1.0 H12 H21 1.0 )
flag =  1 => H = ( H11 1.0 -1.0 H22 )
flag = -2 => H = ( 1.0 0.0 0.0 1.0 )
```

param may be stored in either host or device memory,
location is specified by calling rocbblas_set_pointer_mode.

rocbblas_status **rocbblas_srotm_batched**(*rocbblas_handle* handle, *rocbblas_int* n, float *const x[], *rocbblas_int* incx, float *const y[], *rocbblas_int* incy, const float *const param[], *rocbblas_int* batch_count)

rocbblas_status **rocbblas_drotm_batched**(*rocbblas_handle* handle, *rocbblas_int* n, double *const x[], *rocbblas_int* incx, double *const y[], *rocbblas_int* incy, const double *const param[], *rocbblas_int* batch_count)

BLAS Level 1 API

rotm_batched applies the modified Givens rotation matrix defined by param_i to batched vectors x_i and y_i, for i = 1, ..., batch_count.

Parameters

- **handle** – [in] [rocbblas_handle] handle to the rocbblas library context queue.
- **n** – [in] [rocbblas_int] number of elements in the x and y vectors.
- **x** – [inout] device array of device pointers storing each vector x_i.
- **incx** – [in] [rocbblas_int] specifies the increment between elements of each x_i.
- **y** – [inout] device array of device pointers storing each vector y_i.
- **incy** – [in] [rocbblas_int] specifies the increment between elements of each y_i.
- **param** – [in] device array of device vectors of 5 elements defining the rotation.

```
param[0] = flag
param[1] = H11
param[2] = H21
param[3] = H12
param[4] = H22
```

The flag parameter defines the form of H:

```
flag = -1 => H = ( H11 H12 H21 H22 )
flag =  0 => H = ( 1.0 H12 H21 1.0 )
flag =  1 => H = ( H11 1.0 -1.0 H22 )
flag = -2 => H = ( 1.0 0.0 0.0 1.0 )
```

param may ONLY be stored on the device **for** the batched version of `rotm_strided_batched` this function.

- **batch_count** – [in] [roclblas_int] the number of x and y arrays, i.e. the number of batches.

roclblas_status **roclblas_srotm_strided_batched**(*roclblas_handle* handle, *roclblas_int* n, float *x, *roclblas_int* incx, *roclblas_stride* stride_x, float *y, *roclblas_int* incy, *roclblas_stride* stride_y, const float *param, *roclblas_stride* stride_param, *roclblas_int* batch_count)

roclblas_status **roclblas_drotm_strided_batched**(*roclblas_handle* handle, *roclblas_int* n, double *x, *roclblas_int* incx, *roclblas_stride* stride_x, double *y, *roclblas_int* incy, *roclblas_stride* stride_y, const double *param, *roclblas_stride* stride_param, *roclblas_int* batch_count)

BLAS Level 1 API

`rotm_strided_batched` applies the modified Givens rotation matrix defined by `param_i` to strided batched vectors `x_i` and `y_i`, for `i = 1, ..., batch_count`

Parameters

- **handle** – [in] [roclblas_handle] handle to the roclblas library context queue.
- **n** – [in] [roclblas_int] number of elements in the x and y vectors.
- **x** – [inout] device pointer pointing to first strided batched vector `x_1`.
- **incx** – [in] [roclblas_int] specifies the increment between elements of each `x_i`.
- **stride_x** – [in] [roclblas_stride] specifies the increment between the beginning of `x_i` and `x_(i + 1)`.
- **y** – [inout] device pointer pointing to first strided batched vector `y_1`.
- **incy** – [in] [roclblas_int] specifies the increment between elements of each `y_i`.
- **stride_y** – [in] [roclblas_stride] specifies the increment between the beginning of `y_i` and `y_(i + 1)`.
- **param** – [in] device pointer pointing to first array of 5 elements defining the rotation (`param_1`).

```
param[0] = flag
param[1] = H11
param[2] = H21
param[3] = H12
```

(continues on next page)

(continued from previous page)

```
param[4] = H22
```

The flag parameter defines the form of H:

```
flag = -1 => H = ( H11 H12 H21 H22 )
flag =  0 => H = ( 1.0 H12 H21 1.0 )
flag =  1 => H = ( H11 1.0 -1.0 H22 )
flag = -2 => H = ( 1.0 0.0 0.0 1.0 )
```

param may ONLY be stored on the device **for** the strided_batched version of this function.

- **stride_param** – [in] [rocbas_stride] specifies the increment between the beginning of param_i and param_(i + 1).
- **batch_count** – [in] [rocbas_int] the number of x and y arrays, i.e. the number of batches.

5.4.11 rocbas_Xrotmg + batched, strided_batched

rocbas_status **rocbas_srotmg**(*rocbas_handle* handle, float *d1, float *d2, float *x1, const float *y1, float *param)

rocbas_status **rocbas_drotmg**(*rocbas_handle* handle, double *d1, double *d2, double *x1, const double *y1, double *param)

BLAS Level 1 API

rotmg creates the modified Givens rotation matrix for the vector (d1 * x1, d2 * y1). Parameters may be stored in either host or device memory. Location is specified by calling rocbas_set_pointer_mode:

Parameters

- **handle** – [in] [rocbas_handle] handle to the rocbas library context queue.
- **d1** – [inout] device pointer or host pointer to input scalar that is overwritten.
- **d2** – [inout] device pointer or host pointer to input scalar that is overwritten.
- **x1** – [inout] device pointer or host pointer to input scalar that is overwritten.
- **y1** – [in] device pointer or host pointer to input scalar.
- **param** – [out] device vector or host vector of five elements defining the rotation.

```
param[0] = flag
param[1] = H11
param[2] = H21
param[3] = H12
param[4] = H22
```

The flag parameter defines the form of H:

```
flag = -1 => H = ( H11 H12 H21 H22 )
flag =  0 => H = ( 1.0 H12 H21 1.0 )
flag =  1 => H = ( H11 1.0 -1.0 H22 )
flag = -2 => H = ( 1.0 0.0 0.0 1.0 )
```

(continues on next page)

(continued from previous page)

param may be stored **in** either host **or** device memory.
Location **is** specified by calling `rocblas_set_pointer_mode`.

rocblas_status **rocblas_srotmg_batched**(*rocblas_handle* handle, float *const d1[], float *const d2[], float *const x1[], const float *const y1[], float *const param[], *rocblas_int* batch_count)

rocblas_status **rocblas_drotmg_batched**(*rocblas_handle* handle, double *const d1[], double *const d2[], double *const x1[], const double *const y1[], double *const param[], *rocblas_int* batch_count)

BLAS Level 1 API

`rotmg_batched` creates the modified Givens rotation matrix for the batched vectors ($d1_i * x1_i$, $d2_i * y1_i$), for $i = 1, \dots, \text{batch_count}$. Parameters may be stored in either host or device memory. Location is specified by calling `rocblas_set_pointer_mode`:

- If the pointer mode is set to `rocblas_pointer_mode_host`, then this function blocks the CPU until the GPU has finished and the results are available in host memory.
- If the pointer mode is set to `rocblas_pointer_mode_device`, then this function returns immediately and synchronization is required to read the results.

Parameters

- **handle** – [in] [`rocblas_handle`] handle to the rocblas library context queue.
- **d1** – [inout] device batched array or host batched array of input scalars that is overwritten.
- **d2** – [inout] device batched array or host batched array of input scalars that is overwritten.
- **x1** – [inout] device batched array or host batched array of input scalars that is overwritten.
- **y1** – [in] device batched array or host batched array of input scalars.
- **param** – [out] device batched array or host batched array of vectors of 5 elements defining the rotation.

```
param[0] = flag
param[1] = H11
param[2] = H21
param[3] = H12
param[4] = H22
```

The flag parameter defines the form of H:

```
flag = -1 => H = ( H11 H12 H21 H22 )
flag =  0 => H = ( 1.0 H12 H21 1.0 )
flag =  1 => H = ( H11 1.0 -1.0 H22 )
flag = -2 => H = ( 1.0 0.0 0.0 1.0 )
```

param may be stored **in** either host **or** device memory.
Location **is** specified by calling `rocblas_set_pointer_mode`.

- **batch_count** – [in] [`rocblas_int`] the number of instances in the batch.

```
rocblas_status rocblas_srotmg_strided_batched(rocblas_handle handle, float *d1, rocblas_stride stride_d1,
float *d2, rocblas_stride stride_d2, float *x1, rocblas_stride
stride_x1, const float *y1, rocblas_stride stride_y1, float
*param, rocblas_stride stride_param, rocblas_int
batch_count)
```

```
rocblas_status rocblas_drotmg_strided_batched(rocblas_handle handle, double *d1, rocblas_stride stride_d1,
double *d2, rocblas_stride stride_d2, double *x1,
rocblas_stride stride_x1, const double *y1, rocblas_stride
stride_y1, double *param, rocblas_stride stride_param,
rocblas_int batch_count)
```

BLAS Level 1 API

rotmg_strided_batched creates the modified Givens rotation matrix for the strided batched vectors ($d1_i * x1_i$, $d2_i * y1_i$), for $i = 1, \dots, \text{batch_count}$. Parameters may be stored in either host or device memory. Location is specified by calling rocblas_set_pointer_mode:

- If the pointer mode is set to rocblas_pointer_mode_host, then this function blocks the CPU until the GPU has finished and the results are available in host memory.
- If the pointer mode is set to rocblas_pointer_mode_device, then this function returns immediately and synchronization is required to read the results.

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **d1** – [inout] device strided_batched array or host strided_batched array of input scalars that is overwritten.
- **stride_d1** – [in] [rocblas_stride] specifies the increment between the beginning of $d1_i$ and $d1_{(i+1)}$.
- **d2** – [inout] device strided_batched array or host strided_batched array of input scalars that is overwritten.
- **stride_d2** – [in] [rocblas_stride] specifies the increment between the beginning of $d2_i$ and $d2_{(i+1)}$.
- **x1** – [inout] device strided_batched array or host strided_batched array of input scalars that is overwritten.
- **stride_x1** – [in] [rocblas_stride] specifies the increment between the beginning of $x1_i$ and $x1_{(i+1)}$.
- **y1** – [in] device strided_batched array or host strided_batched array of input scalars.
- **stride_y1** – [in] [rocblas_stride] specifies the increment between the beginning of $y1_i$ and $y1_{(i+1)}$.
- **param** – [out] device strided_batched array or host strided_batched array of vectors of 5 elements defining the rotation.

```
param[0] = flag
param[1] = H11
param[2] = H21
param[3] = H12
param[4] = H22
```

(continues on next page)

(continued from previous page)

The flag parameter defines the form of H:

```
flag = -1 => H = ( H11 H12 H21 H22 )
flag =  0 => H = ( 1.0 H12 H21 1.0 )
flag =  1 => H = ( H11 1.0 -1.0 H22 )
flag = -2 => H = ( 1.0 0.0 0.0 1.0 )
```

param may be stored **in** either host **or** device memory.
Location **is** specified by calling `rocblas_set_pointer_mode`.

- **stride_param** – [in] [rocblas_stride] specifies the increment between the beginning of param_i and param_(i + 1).
- **batch_count** – [in] [rocblas_int] the number of instances in the batch.

5.4.12 rocblas_Xscal + batched, strided_batched

rocblas_status **rocblas_sscal**(*rocblas_handle* handle, *rocblas_int* n, const float *alpha, float *x, *rocblas_int* incx)

rocblas_status **rocblas_dscal**(*rocblas_handle* handle, *rocblas_int* n, const double *alpha, double *x, *rocblas_int* incx)

rocblas_status **rocblas_cscal**(*rocblas_handle* handle, *rocblas_int* n, const *rocblas_float_complex* *alpha, *rocblas_float_complex* *x, *rocblas_int* incx)

rocblas_status **rocblas_zscal**(*rocblas_handle* handle, *rocblas_int* n, const *rocblas_double_complex* *alpha, *rocblas_double_complex* *x, *rocblas_int* incx)

rocblas_status **rocblas_csscal**(*rocblas_handle* handle, *rocblas_int* n, const float *alpha, *rocblas_float_complex* *x, *rocblas_int* incx)

rocblas_status **rocblas_zdscal**(*rocblas_handle* handle, *rocblas_int* n, const double *alpha, *rocblas_double_complex* *x, *rocblas_int* incx)

BLAS Level 1 API

scal scales each element of vector x with scalar alpha:

```
x := alpha * x
```

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **n** – [in] [rocblas_int] the number of elements in x.
- **alpha** – [in] device pointer or host pointer for the scalar alpha.
- **x** – [inout] device pointer storing vector x.
- **incx** – [in] [rocblas_int] specifies the increment for the elements of x.

rocblas_status **rocblas_sscal_batched**(*rocblas_handle* handle, *rocblas_int* n, const float *alpha, float *const x[], *rocblas_int* incx, *rocblas_int* batch_count)

```

rocblas_status rocblas_dscal_batched(rocblas_handle handle, rocblas_int n, const double *alpha, double
                                     *const x[], rocblas_int incx, rocblas_int batch_count)

rocblas_status rocblas_cscal_batched(rocblas_handle handle, rocblas_int n, const rocblas_float_complex
                                     *alpha, rocblas_float_complex *const x[], rocblas_int incx, rocblas_int
                                     batch_count)

rocblas_status rocblas_zscal_batched(rocblas_handle handle, rocblas_int n, const rocblas_double_complex
                                     *alpha, rocblas_double_complex *const x[], rocblas_int incx,
                                     rocblas_int batch_count)

rocblas_status rocblas_csscal_batched(rocblas_handle handle, rocblas_int n, const float *alpha,
                                     rocblas_float_complex *const x[], rocblas_int incx, rocblas_int
                                     batch_count)

rocblas_status rocblas_zdscal_batched(rocblas_handle handle, rocblas_int n, const double *alpha,
                                     rocblas_double_complex *const x[], rocblas_int incx, rocblas_int
                                     batch_count)

```

BLAS Level 1 API

scal_batched scales each element of vector x_i with scalar α , for $i = 1, \dots, \text{batch_count}$:

```

x_i := alpha * x_i,
where (x_i) is the i-th instance of the batch.

```

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **n** – [in] [rocblas_int] the number of elements in each x_i .
- **alpha** – [in] host pointer or device pointer for the scalar α .
- **x** – [inout] device array of device pointers storing each vector x_i .
- **incx** – [in] [rocblas_int] specifies the increment for the elements of each x_i .
- **batch_count** – [in] [rocblas_int] specifies the number of batches in x .

```

rocblas_status rocblas_sscal_strided_batched(rocblas_handle handle, rocblas_int n, const float *alpha, float
                                              *x, rocblas_int incx, rocblas_stride stride_x, rocblas_int
                                              batch_count)

rocblas_status rocblas_dscal_strided_batched(rocblas_handle handle, rocblas_int n, const double *alpha,
                                              double *x, rocblas_int incx, rocblas_stride stride_x,
                                              rocblas_int batch_count)

rocblas_status rocblas_cscal_strided_batched(rocblas_handle handle, rocblas_int n, const
                                              rocblas_float_complex *alpha, rocblas_float_complex *x,
                                              rocblas_int incx, rocblas_stride stride_x, rocblas_int
                                              batch_count)

rocblas_status rocblas_zscal_strided_batched(rocblas_handle handle, rocblas_int n, const
                                              rocblas_double_complex *alpha, rocblas_double_complex *x,
                                              rocblas_int incx, rocblas_stride stride_x, rocblas_int
                                              batch_count)

```

rocblas_status **rocblas_csscal_strided_batched**(*rocblas_handle* handle, *rocblas_int* n, const float *alpha, *rocblas_float_complex* *x, *rocblas_int* incx, *rocblas_stride* stride_x, *rocblas_int* batch_count)

rocblas_status **rocblas_zdscal_strided_batched**(*rocblas_handle* handle, *rocblas_int* n, const double *alpha, *rocblas_double_complex* *x, *rocblas_int* incx, *rocblas_stride* stride_x, *rocblas_int* batch_count)

BLAS Level 1 API

scal_strided_batched scales each element of vector x_i with scalar alpha, for i = 1, ... , batch_count:

```
x_i := alpha * x_i,
where (x_i) is the i-th instance of the batch.
```

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **n** – [in] [rocblas_int] the number of elements in each x_i.
- **alpha** – [in] host pointer or device pointer for the scalar alpha.
- **x** – [inout] device pointer to the first vector (x_1) in the batch.
- **incx** – [in] [rocblas_int] specifies the increment for the elements of x.
- **stride_x** – [in] [rocblas_stride] stride from the start of one vector (x_i) and the next one (x_{i+1}). There are no restrictions placed on stride_x. However, ensure that stride_x is of appropriate size, for a typical case this means stride_x >= n * incx.
- **batch_count** – [in] [rocblas_int] specifies the number of batches in x.

5.4.13 rocblas_Xswap + batched, strided_batched

rocblas_status **rocblas_sswap**(*rocblas_handle* handle, *rocblas_int* n, float *x, *rocblas_int* incx, float *y, *rocblas_int* incy)

rocblas_status **rocblas_dswap**(*rocblas_handle* handle, *rocblas_int* n, double *x, *rocblas_int* incx, double *y, *rocblas_int* incy)

rocblas_status **rocblas_cswap**(*rocblas_handle* handle, *rocblas_int* n, *rocblas_float_complex* *x, *rocblas_int* incx, *rocblas_float_complex* *y, *rocblas_int* incy)

rocblas_status **rocblas_zswap**(*rocblas_handle* handle, *rocblas_int* n, *rocblas_double_complex* *x, *rocblas_int* incx, *rocblas_double_complex* *y, *rocblas_int* incy)

BLAS Level 1 API

swap interchanges vectors x and y:

```
y := x;
x := y
```

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **n** – [in] [rocblas_int] the number of elements in x and y.

- **x** – [inout] device pointer storing vector x.
- **incx** – [in] [rocblas_int] specifies the increment for the elements of x.
- **y** – [inout] device pointer storing vector y.
- **incy** – [in] [rocblas_int] specifies the increment for the elements of y.

```
rocblas_status rocblas_sswap_batched(rocblas_handle handle, rocblas_int n, float *const x[], rocblas_int incx,
                                     float *const y[], rocblas_int incy, rocblas_int batch_count)
```

```
rocblas_status rocblas_dswap_batched(rocblas_handle handle, rocblas_int n, double *const x[], rocblas_int
                                     incx, double *const y[], rocblas_int incy, rocblas_int batch_count)
```

```
rocblas_status rocblas_cswap_batched(rocblas_handle handle, rocblas_int n, rocblas_float_complex *const x[],
                                     rocblas_int incx, rocblas_float_complex *const y[], rocblas_int incy,
                                     rocblas_int batch_count)
```

```
rocblas_status rocblas_zswap_batched(rocblas_handle handle, rocblas_int n, rocblas_double_complex *const
                                     x[], rocblas_int incx, rocblas_double_complex *const y[], rocblas_int
                                     incy, rocblas_int batch_count)
```

BLAS Level 1 API

swap_batched interchanges vectors x_i and y_i , for $i = 1, \dots, \text{batch_count}$:

```
y_i := x_i;
x_i := y_i
```

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **n** – [in] [rocblas_int] the number of elements in each x_i and y_i .
- **x** – [inout] device array of device pointers storing each vector x_i .
- **incx** – [in] [rocblas_int] specifies the increment for the elements of each x_i .
- **y** – [inout] device array of device pointers storing each vector y_i .
- **incy** – [in] [rocblas_int] specifies the increment for the elements of each y_i .
- **batch_count** – [in] [rocblas_int] number of instances in the batch.

```
rocblas_status rocblas_sswap_strided_batched(rocblas_handle handle, rocblas_int n, float *x, rocblas_int
                                              incx, rocblas_stride stridex, float *y, rocblas_int incy,
                                              rocblas_stride stridey, rocblas_int batch_count)
```

```
rocblas_status rocblas_dswap_strided_batched(rocblas_handle handle, rocblas_int n, double *x, rocblas_int
                                              incx, rocblas_stride stridex, double *y, rocblas_int incy,
                                              rocblas_stride stridey, rocblas_int batch_count)
```

```
rocblas_status rocblas_cswap_strided_batched(rocblas_handle handle, rocblas_int n, rocblas_float_complex
                                              *x, rocblas_int incx, rocblas_stride stridex,
                                              rocblas_float_complex *y, rocblas_int incy, rocblas_stride
                                              stridey, rocblas_int batch_count)
```

```
rocblas_status rocblas_zswap_strided_batched(rocblas_handle handle, rocblas_int n,
                                             rocblas_double_complex *x, rocblas_int incx, rocblas_stride
                                             stridex, rocblas_double_complex *y, rocblas_int incy,
                                             rocblas_stride stridey, rocblas_int batch_count)
```

BLAS Level 1 API

swap_strided_batched interchanges vectors x_i and y_i , for $i = 1, \dots, \text{batch_count}$:

```
y_i := x_i;
x_i := y_i
```

Parameters

- **handle** – [in] [*rocblas_handle*] handle to the rocblas library context queue.
- **n** – [in] [*rocblas_int*] the number of elements in each x_i and y_i .
- **x** – [inout] device pointer to the first vector x_1 .
- **incx** – [in] [*rocblas_int*] specifies the increment for the elements of x .
- **stridex** – [in] [*rocblas_stride*] stride from the start of one vector (x_i) and the next one (x_{i+1}). There are no restrictions placed on stride_x. However, ensure that stride_x is of appropriate size. For a typical case this means $\text{stride_x} \geq n * \text{incx}$.
- **y** – [inout] device pointer to the first vector y_1 .
- **incy** – [in] [*rocblas_int*] specifies the increment for the elements of y .
- **stridey** – [in] [*rocblas_stride*] stride from the start of one vector (y_i) and the next one (y_{i+1}). There are no restrictions placed on stride_y. However, ensure that stride_y is of appropriate size. For a typical case this means $\text{stride_y} \geq n * \text{incy}$. stridey should be non zero.
- **batch_count** – [in] [*rocblas_int*] number of instances in the batch.

5.5 rocBLAS Level-2 functions

5.5.1 rocblas_Xgbmv + batched, strided_batched

```
rocblas_status rocblas_sgbmv(rocblas_handle handle, rocblas_operation trans, rocblas_int m, rocblas_int n,
                              rocblas_int kl, rocblas_int ku, const float *alpha, const float *A, rocblas_int lda,
                              const float *x, rocblas_int incx, const float *beta, float *y, rocblas_int incy)
```

```
rocblas_status rocblas_dgbmv(rocblas_handle handle, rocblas_operation trans, rocblas_int m, rocblas_int n,
                              rocblas_int kl, rocblas_int ku, const double *alpha, const double *A, rocblas_int
                              lda, const double *x, rocblas_int incx, const double *beta, double *y, rocblas_int
                              incy)
```

```
rocblas_status rocblas_cgbmv(rocblas_handle handle, rocblas_operation trans, rocblas_int m, rocblas_int n,
                              rocblas_int kl, rocblas_int ku, const rocblas_float_complex *alpha, const
                              rocblas_float_complex *A, rocblas_int lda, const rocblas_float_complex *x,
                              rocblas_int incx, const rocblas_float_complex *beta, rocblas_float_complex *y,
                              rocblas_int incy)
```

```
rocblas_status rocblas_zgbmv(rocblas_handle handle, rocblas_operation trans, rocblas_int m, rocblas_int n,
                             rocblas_int kl, rocblas_int ku, const rocblas_double_complex *alpha, const
                             rocblas_double_complex *A, rocblas_int lda, const rocblas_double_complex *x,
                             rocblas_int incx, const rocblas_double_complex *beta, rocblas_double_complex
                             *y, rocblas_int incy)
```

BLAS Level 2 API

gbmv 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 banded matrix with kl sub-diagonals and ku super-diagonals.

Note that the empty elements which do not correspond to data will not be referenced.

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **trans** – [in] [rocblas_operation] indicates whether matrix A is transposed (conjugated) or not.
- **m** – [in] [rocblas_int] number of rows of matrix A.
- **n** – [in] [rocblas_int] number of columns of matrix A.
- **kl** – [in] [rocblas_int] number of sub-diagonals of A.
- **ku** – [in] [rocblas_int] number of super-diagonals of A.
- **alpha** – [in] device pointer or host pointer to scalar alpha.
- **A** – [in] device pointer storing banded matrix A. Leading (kl + ku + 1) by n part of the matrix contains the coefficients of the banded matrix. The leading diagonal resides in row (ku + 1) with the first super-diagonal above on the RHS of row ku. The first sub-diagonal resides below on the LHS of row ku + 2. This propagates up and down across sub/super-diagonals.

```
Ex: (m = n = 7; ku = 2, kl = 2)
1 2 3 0 0 0 0      0 0 3 3 3 3 3
4 1 2 3 0 0 0      0 2 2 2 2 2 2
5 4 1 2 3 0 0      ----> 1 1 1 1 1 1 1
0 5 4 1 2 3 0      4 4 4 4 4 4 0
0 0 5 4 1 2 0      5 5 5 5 5 0 0
0 0 0 5 4 1 2      0 0 0 0 0 0 0
0 0 0 0 5 4 1      0 0 0 0 0 0 0
```

- **lda** – [in] [rocblas_int] specifies the leading dimension of A. Must be $\geq (kl + ku + 1)$.
- **x** – [in] device pointer storing vector x.
- **incx** – [in] [rocblas_int] specifies the increment for the elements of x.
- **beta** – [in] device pointer or host pointer to scalar beta.
- **y** – [inout] device pointer storing vector y.
- **incy** – [in] [rocblas_int] specifies the increment for the elements of y.

```
rocblas_status rocblas_sgbmv_batched(rocblas_handle handle, rocblas_operation trans, rocblas_int m,
                                     rocblas_int n, rocblas_int kl, rocblas_int ku, const float *alpha, const
                                     float *const A[], rocblas_int lda, const float *const x[], rocblas_int incx,
                                     const float *beta, float *const y[], rocblas_int incy, rocblas_int
                                     batch_count)
```

```
rocblas_status rocblas_dgbmv_batched(rocblas_handle handle, rocblas_operation trans, rocblas_int m,
                                     rocblas_int n, rocblas_int kl, rocblas_int ku, const double *alpha, const
                                     double *const A[], rocblas_int lda, const double *const x[], rocblas_int
                                     incx, const double *beta, double *const y[], rocblas_int incy, rocblas_int
                                     batch_count)
```

```
rocblas_status rocblas_cgbmv_batched(rocblas_handle handle, rocblas_operation trans, rocblas_int m,
                                     rocblas_int n, rocblas_int kl, rocblas_int ku, const
                                     rocblas_float_complex *alpha, const rocblas_float_complex *const A[],
                                     rocblas_int lda, const rocblas_float_complex *const x[], rocblas_int
                                     incx, const rocblas_float_complex *beta, rocblas_float_complex *const
                                     y[], rocblas_int incy, rocblas_int batch_count)
```

```
rocblas_status rocblas_zgbmv_batched(rocblas_handle handle, rocblas_operation trans, rocblas_int m,
                                     rocblas_int n, rocblas_int kl, rocblas_int ku, const
                                     rocblas_double_complex *alpha, const rocblas_double_complex *const
                                     A[], rocblas_int lda, const rocblas_double_complex *const x[],
                                     rocblas_int incx, const rocblas_double_complex *beta,
                                     rocblas_double_complex *const y[], rocblas_int incy, rocblas_int
                                     batch_count)
```

BLAS Level 2 API

gbmv_batched performs one of the matrix-vector operations:

```
y_i := alpha*A_i*x_i + beta*y_i,   or
y_i := alpha*A_i**T*x_i + beta*y_i, or
y_i := alpha*A_i**H*x_i + beta*y_i,
where (A_i, x_i, y_i) is the i-th instance of the batch.
alpha and beta are scalars, x_i and y_i are vectors and A_i is an
m by n banded matrix with kl sub-diagonals and ku super-diagonals,
for i = 1, ..., batch_count.
```

Note that the empty elements which do not correspond to data will not be referenced.

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **trans** – [in] [rocblas_operation] indicates whether matrix A is transposed (conjugated) or not.
- **m** – [in] [rocblas_int] number of rows of each matrix A_i.
- **n** – [in] [rocblas_int] number of columns of each matrix A_i.
- **kl** – [in] [rocblas_int] number of sub-diagonals of each A_i.
- **ku** – [in] [rocblas_int] number of super-diagonals of each A_i.
- **alpha** – [in] device pointer or host pointer to scalar alpha.

- **A** – [in] device array of device pointers storing each banded matrix A_i . Leading $(kl + ku + 1)$ by n part of the matrix contains the coefficients of the banded matrix. The leading diagonal resides in row $(ku + 1)$ with the first super-diagonal above on the RHS of row ku . The first sub-diagonal resides below on the LHS of row $ku + 2$. This propagates up and down across sub/super-diagonals.

Ex: (m = n = 7; ku = 2, kl = 2)																
1	2	3	0	0	0	0				0	0	3	3	3	3	3
4	1	2	3	0	0	0				0	2	2	2	2	2	2
5	4	1	2	3	0	0		----	>	1	1	1	1	1	1	1
0	5	4	1	2	3	0				4	4	4	4	4	4	0
0	0	5	4	1	2	0				5	5	5	5	5	0	0
0	0	0	5	4	1	2				0	0	0	0	0	0	0
0	0	0	0	5	4	1				0	0	0	0	0	0	0

- **lda** – [in] [rocblas_int] specifies the leading dimension of each A_i . Must be $\geq (kl + ku + 1)$
- **x** – [in] device array of device pointers storing each vector x_i .
- **incx** – [in] [rocblas_int] specifies the increment for the elements of each x_i .
- **beta** – [in] device pointer or host pointer to scalar beta.
- **y** – [inout] device array of device pointers storing each vector y_i .
- **incy** – [in] [rocblas_int] specifies the increment for the elements of each y_i .
- **batch_count** – [in] [rocblas_int] specifies the number of instances in the batch.

rocblas_status **rocblas_sgbmv_strided_batched**(*rocblas_handle* handle, *rocblas_operation* trans, *rocblas_int* m, *rocblas_int* n, *rocblas_int* kl, *rocblas_int* ku, const float *alpha, const float *A, *rocblas_int* lda, *rocblas_stride* stride_A, const float *x, *rocblas_int* incx, *rocblas_stride* stride_x, const float *beta, float *y, *rocblas_int* incy, *rocblas_stride* stride_y, *rocblas_int* batch_count)

rocblas_status **rocblas_dgbmv_strided_batched**(*rocblas_handle* handle, *rocblas_operation* trans, *rocblas_int* m, *rocblas_int* n, *rocblas_int* kl, *rocblas_int* ku, const double *alpha, const double *A, *rocblas_int* lda, *rocblas_stride* stride_A, const double *x, *rocblas_int* incx, *rocblas_stride* stride_x, const double *beta, double *y, *rocblas_int* incy, *rocblas_stride* stride_y, *rocblas_int* batch_count)

rocblas_status **rocblas_cgbmv_strided_batched**(*rocblas_handle* handle, *rocblas_operation* trans, *rocblas_int* m, *rocblas_int* n, *rocblas_int* kl, *rocblas_int* ku, const *rocblas_float_complex* *alpha, const *rocblas_float_complex* *A, *rocblas_int* lda, *rocblas_stride* stride_A, const *rocblas_float_complex* *x, *rocblas_int* incx, *rocblas_stride* stride_x, const *rocblas_float_complex* *beta, *rocblas_float_complex* *y, *rocblas_int* incy, *rocblas_stride* stride_y, *rocblas_int* batch_count)

```

rocblas_status rocblas_zgbmv_strided_batched(rocblas_handle handle, rocblas_operation trans, rocblas_int
m, rocblas_int n, rocblas_int kl, rocblas_int ku, const
rocblas_double_complex *alpha, const
rocblas_double_complex *A, rocblas_int lda, rocblas_stride
stride_A, const rocblas_double_complex *x, rocblas_int incx,
rocblas_stride stride_x, const rocblas_double_complex *beta,
rocblas_double_complex *y, rocblas_int incy, rocblas_stride
stride_y, rocblas_int batch_count)

```

BLAS Level 2 API

gbmv_strided_batched performs one of the matrix-vector operations:

```

y_i := alpha*A_i*x_i + beta*y_i,   or
y_i := alpha*A_i**T*x_i + beta*y_i, or
y_i := alpha*A_i**H*x_i + beta*y_i,
where (A_i, x_i, y_i) is the i-th instance of the batch.
alpha and beta are scalars, x_i and y_i are vectors and A_i is an
m by n banded matrix with kl sub-diagonals and ku super-diagonals,
for i = 1, ..., batch_count.

```

Note that the empty elements which do not correspond to data will not be referenced.

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **trans** – [in] [rocblas_operation] indicates whether matrix A is transposed (conjugated) or not.
- **m** – [in] [rocblas_int] number of rows of matrix A.
- **n** – [in] [rocblas_int] number of columns of matrix A.
- **kl** – [in] [rocblas_int] number of sub-diagonals of A.
- **ku** – [in] [rocblas_int] number of super-diagonals of A.
- **alpha** – [in] device pointer or host pointer to scalar alpha.
- **A** – [in] device pointer to first banded matrix (A_1). Leading (kl + ku + 1) by n part of the matrix contains the coefficients of the banded matrix. The leading diagonal resides in row (ku + 1) with the first super-diagonal above on the RHS of row ku. The first sub-diagonal resides below on the LHS of row ku + 2. This propagates up and down across sub/super-diagonals.

```

Ex: (m = n = 7; ku = 2, kl = 2)
1 2 3 0 0 0 0      0 0 3 3 3 3 3
4 1 2 3 0 0 0      0 2 2 2 2 2 2
5 4 1 2 3 0 0      ----> 1 1 1 1 1 1 1
0 5 4 1 2 3 0      4 4 4 4 4 4 0
0 0 5 4 1 2 0      5 5 5 5 5 0 0
0 0 0 5 4 1 2      0 0 0 0 0 0 0
0 0 0 0 5 4 1      0 0 0 0 0 0 0

```

- **lda** – [in] [rocblas_int] specifies the leading dimension of A. Must be $\geq (kl + ku + 1)$.
- **stride_A** – [in] [rocblas_stride] stride from the start of one matrix (A_i) and the next one (A_{i+1}).
- **x** – [in] device pointer to first vector (x_1).

- **incx** – [in] [roclblas_int] specifies the increment for the elements of x.
- **stride_x** – [in] [roclblas_stride] stride from the start of one vector (x_i) and the next one (x_i+1).
- **beta** – [in] device pointer or host pointer to scalar beta.
- **y** – [inout] device pointer to first vector (y_1).
- **incy** – [in] [roclblas_int] specifies the increment for the elements of y.
- **stride_y** – [in] [roclblas_stride] stride from the start of one vector (y_i) and the next one (y_i+1).
- **batch_count** – [in] [roclblas_int] specifies the number of instances in the batch.

5.5.2 roclblas_Xgemv + batched, strided_batched

roclblas_status **roclblas_sgemv**(*roclblas_handle* handle, *roclblas_operation* trans, *roclblas_int* m, *roclblas_int* n, const float *alpha, const float *A, *roclblas_int* lda, const float *x, *roclblas_int* incx, const float *beta, float *y, *roclblas_int* incy)

roclblas_status **roclblas_dgemv**(*roclblas_handle* handle, *roclblas_operation* trans, *roclblas_int* m, *roclblas_int* n, const double *alpha, const double *A, *roclblas_int* lda, const double *x, *roclblas_int* incx, const double *beta, double *y, *roclblas_int* incy)

roclblas_status **roclblas_cgemv**(*roclblas_handle* handle, *roclblas_operation* trans, *roclblas_int* m, *roclblas_int* n, const *roclblas_float_complex* *alpha, const *roclblas_float_complex* *A, *roclblas_int* lda, const *roclblas_float_complex* *x, *roclblas_int* incx, const *roclblas_float_complex* *beta, *roclblas_float_complex* *y, *roclblas_int* incy)

roclblas_status **roclblas_zgemv**(*roclblas_handle* handle, *roclblas_operation* trans, *roclblas_int* m, *roclblas_int* n, const *roclblas_double_complex* *alpha, const *roclblas_double_complex* *A, *roclblas_int* lda, const *roclblas_double_complex* *x, *roclblas_int* incx, const *roclblas_double_complex* *beta, *roclblas_double_complex* *y, *roclblas_int* incy)

BLAS Level 2 API

gemv 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.

```

Parameters

- **handle** – [in] [roclblas_handle] handle to the roclblas library context queue.
- **trans** – [in] [roclblas_operation] indicates whether matrix A is transposed (conjugated) or not.
- **m** – [in] [roclblas_int] number of rows of matrix A.
- **n** – [in] [roclblas_int] number of columns of matrix A.
- **alpha** – [in] device pointer or host pointer to scalar alpha.
- **A** – [in] device pointer storing matrix A.

- **lda** – [in] [rocblas_int] specifies the leading dimension of A.
- **x** – [in] device pointer storing vector x.
- **incx** – [in] [rocblas_int] specifies the increment for the elements of x.
- **beta** – [in] device pointer or host pointer to scalar beta.
- **y** – [inout] device pointer storing vector y.
- **incy** – [in] [rocblas_int] specifies the increment for the elements of y.

```
rocblas_status rocblas_sgemv_batched(rocblas_handle handle, rocblas_operation trans, rocblas_int m,
                                     rocblas_int n, const float *alpha, const float *const A[], rocblas_int lda,
                                     const float *const x[], rocblas_int incx, const float *beta, float *const y[],
                                     rocblas_int incy, rocblas_int batch_count)
```

```
rocblas_status rocblas_dgemv_batched(rocblas_handle handle, rocblas_operation trans, rocblas_int m,
                                     rocblas_int n, const double *alpha, const double *const A[], rocblas_int
                                     lda, const double *const x[], rocblas_int incx, const double *beta, double
                                     *const y[], rocblas_int incy, rocblas_int batch_count)
```

```
rocblas_status rocblas_cgemv_batched(rocblas_handle handle, rocblas_operation trans, rocblas_int m,
                                     rocblas_int n, const rocblas_float_complex *alpha, const
                                     rocblas_float_complex *const A[], rocblas_int lda, const
                                     rocblas_float_complex *const x[], rocblas_int incx, const
                                     rocblas_float_complex *beta, rocblas_float_complex *const y[],
                                     rocblas_int incy, rocblas_int batch_count)
```

```
rocblas_status rocblas_zgemv_batched(rocblas_handle handle, rocblas_operation trans, rocblas_int m,
                                     rocblas_int n, const rocblas_double_complex *alpha, const
                                     rocblas_double_complex *const A[], rocblas_int lda, const
                                     rocblas_double_complex *const x[], rocblas_int incx, const
                                     rocblas_double_complex *beta, rocblas_double_complex *const y[],
                                     rocblas_int incy, rocblas_int batch_count)
```

BLAS Level 2 API

gemv_batched performs a batch of matrix-vector operations:

```
y_i := alpha*A_i*x_i + beta*y_i,   or
y_i := alpha*A_i**T*x_i + beta*y_i, or
y_i := alpha*A_i**H*x_i + beta*y_i,
where (A_i, x_i, y_i) is the i-th instance of the batch.
alpha and beta are scalars, x_i and y_i are vectors and A_i is an
m by n matrix, for i = 1, ..., batch_count.
```

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **trans** – [in] [rocblas_operation] indicates whether matrices A_i are transposed (conjugated) or not.
- **m** – [in] [rocblas_int] number of rows of each matrix A_i.
- **n** – [in] [rocblas_int] number of columns of each matrix A_i.
- **alpha** – [in] device pointer or host pointer to scalar alpha.
- **A** – [in] device array of device pointers storing each matrix A_i.

- **lda** – [in] [roclblas_int] specifies the leading dimension of each matrix A_i .
- **x** – [in] device array of device pointers storing each vector x_i .
- **incx** – [in] [roclblas_int] specifies the increment for the elements of each vector x_i .
- **beta** – [in] device pointer or host pointer to scalar beta.
- **y** – [inout] device array of device pointers storing each vector y_i .
- **incy** – [in] [roclblas_int] specifies the increment for the elements of each vector y_i .
- **batch_count** – [in] [roclblas_int] number of instances in the batch.

```
roclblas_status roclblas_sgemv_strided_batched(roclblas_handle handle, roclblas_operation transA, roclblas_int m, roclblas_int n, const float *alpha, const float *A, roclblas_int lda, roclblas_stride strideA, const float *x, roclblas_int incx, roclblas_stride stridex, const float *beta, float *y, roclblas_int incy, roclblas_stride stridey, roclblas_int batch_count)
```

```
roclblas_status roclblas_dgemv_strided_batched(roclblas_handle handle, roclblas_operation transA, roclblas_int m, roclblas_int n, const double *alpha, const double *A, roclblas_int lda, roclblas_stride strideA, const double *x, roclblas_int incx, roclblas_stride stridex, const double *beta, double *y, roclblas_int incy, roclblas_stride stridey, roclblas_int batch_count)
```

```
roclblas_status roclblas_cgemv_strided_batched(roclblas_handle handle, roclblas_operation transA, roclblas_int m, roclblas_int n, const roclblas_float_complex *alpha, const roclblas_float_complex *A, roclblas_int lda, roclblas_stride strideA, const roclblas_float_complex *x, roclblas_int incx, roclblas_stride stridex, const roclblas_float_complex *beta, roclblas_float_complex *y, roclblas_int incy, roclblas_stride stridey, roclblas_int batch_count)
```

```
roclblas_status roclblas_zgemv_strided_batched(roclblas_handle handle, roclblas_operation transA, roclblas_int m, roclblas_int n, const roclblas_double_complex *alpha, const roclblas_double_complex *A, roclblas_int lda, roclblas_stride strideA, const roclblas_double_complex *x, roclblas_int incx, roclblas_stride stridex, const roclblas_double_complex *beta, roclblas_double_complex *y, roclblas_int incy, roclblas_stride stridey, roclblas_int batch_count)
```

BLAS Level 2 API

gemv_strided_batched performs a batch of matrix-vector operations:

```
y_i := alpha*A_i*x_i + beta*y_i,   or
y_i := alpha*A_i**T*x_i + beta*y_i, or
y_i := alpha*A_i**H*x_i + beta*y_i,
where (A_i, x_i, y_i) is the i-th instance of the batch.
alpha and beta are scalars, x_i and y_i are vectors and A_i is an
m by n matrix, for i = 1, ..., batch_count.
```

Parameters

- **handle** – [in] [roclblas_handle] handle to the roclblas library context queue.
- **transA** – [in] [roclblas_operation] indicates whether matrices A_i are transposed (conjugated) or not.

- **m** – [in] [rocblas_int] number of rows of matrices A_i.
- **n** – [in] [rocblas_int] number of columns of matrices A_i.
- **alpha** – [in] device pointer or host pointer to scalar alpha.
- **A** – [in] device pointer to the first matrix (A_1) in the batch.
- **lda** – [in] [rocblas_int] specifies the leading dimension of matrices A_i.
- **strideA** – [in] [rocblas_stride] stride from the start of one matrix (A_i) and the next one (A_i+1).
- **x** – [in] device pointer to the first vector (x_1) in the batch.
- **incx** – [in] [rocblas_int] specifies the increment for the elements of vectors x_i.
- **stridex** – [in] [rocblas_stride] stride from the start of one vector (x_i) and the next one (x_i+1). There are no restrictions placed on stride_x. However, ensure that stride_x is of appropriate size. When trans equals rocblas_operation_none this typically means stride_x $\geq n * incx$, otherwise stride_x $\geq m * incx$.
- **beta** – [in] device pointer or host pointer to scalar beta.
- **y** – [inout] device pointer to the first vector (y_1) in the batch.
- **incy** – [in] [rocblas_int] specifies the increment for the elements of vectors y_i.
- **stridey** – [in] [rocblas_stride] stride from the start of one vector (y_i) and the next one (y_i+1). There are no restrictions placed on stride_y. However, ensure that stride_y is of appropriate size. When trans equals rocblas_operation_none this typically means stride_y $\geq m * incy$, otherwise stride_y $\geq n * incy$. stridey should be non zero.
- **batch_count** – [in] [rocblas_int] number of instances in the batch.

5.5.3 rocblas_Xger + batched, strided_batched

rocblas_status **rocblas_sger**(*rocblas_handle* handle, *rocblas_int* m, *rocblas_int* n, const float *alpha, const float *x, *rocblas_int* incx, const float *y, *rocblas_int* incy, float *A, *rocblas_int* lda)

rocblas_status **rocblas_dger**(*rocblas_handle* handle, *rocblas_int* m, *rocblas_int* n, const double *alpha, const double *x, *rocblas_int* incx, const double *y, *rocblas_int* incy, double *A, *rocblas_int* lda)

rocblas_status **rocblas_cgeru**(*rocblas_handle* handle, *rocblas_int* m, *rocblas_int* n, const *rocblas_float_complex* *alpha, const *rocblas_float_complex* *x, *rocblas_int* incx, const *rocblas_float_complex* *y, *rocblas_int* incy, *rocblas_float_complex* *A, *rocblas_int* lda)

rocblas_status **rocblas_zgeru**(*rocblas_handle* handle, *rocblas_int* m, *rocblas_int* n, const *rocblas_double_complex* *alpha, const *rocblas_double_complex* *x, *rocblas_int* incx, const *rocblas_double_complex* *y, *rocblas_int* incy, *rocblas_double_complex* *A, *rocblas_int* lda)

rocblas_status **rocblas_cgerc**(*rocblas_handle* handle, *rocblas_int* m, *rocblas_int* n, const *rocblas_float_complex* *alpha, const *rocblas_float_complex* *x, *rocblas_int* incx, const *rocblas_float_complex* *y, *rocblas_int* incy, *rocblas_float_complex* *A, *rocblas_int* lda)

```
rocblas_status rocblas_zgerc(rocblas_handle handle, rocblas_int m, rocblas_int n, const
    rocblas_double_complex *alpha, const rocblas_double_complex *x, rocblas_int
    incx, const rocblas_double_complex *y, rocblas_int incy, rocblas_double_complex
    *A, rocblas_int lda)
```

BLAS Level 2 API

ger,geru,gerc performs the matrix-vector operations:

```
A := A + alpha*x*y**T , OR
A := A + alpha*x*y**H for gerc
where alpha is a scalar, x and y are vectors, and A is an
m by n matrix.
```

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **m** – [in] [rocblas_int] the number of rows of the matrix A.
- **n** – [in] [rocblas_int] the number of columns of the matrix A.
- **alpha** – [in] device pointer or host pointer to scalar alpha.
- **x** – [in] device pointer storing vector x.
- **incx** – [in] [rocblas_int] specifies the increment for the elements of x.
- **y** – [in] device pointer storing vector y.
- **incy** – [in] [rocblas_int] specifies the increment for the elements of y.
- **A** – [inout] device pointer storing matrix A.
- **lda** – [in] [rocblas_int] specifies the leading dimension of A.

```
rocblas_status rocblas_sger_batched(rocblas_handle handle, rocblas_int m, rocblas_int n, const float *alpha,
    const float *const x[], rocblas_int incx, const float *const y[], rocblas_int
    incy, float *const A[], rocblas_int lda, rocblas_int batch_count)
```

```
rocblas_status rocblas_dger_batched(rocblas_handle handle, rocblas_int m, rocblas_int n, const double *alpha,
    const double *const x[], rocblas_int incx, const double *const y[],
    rocblas_int incy, double *const A[], rocblas_int lda, rocblas_int
    batch_count)
```

```
rocblas_status rocblas_cgeru_batched(rocblas_handle handle, rocblas_int m, rocblas_int n, const
    rocblas_float_complex *alpha, const rocblas_float_complex *const x[],
    rocblas_int incx, const rocblas_float_complex *const y[], rocblas_int
    incy, rocblas_float_complex *const A[], rocblas_int lda, rocblas_int
    batch_count)
```

```
rocblas_status rocblas_zgeru_batched(rocblas_handle handle, rocblas_int m, rocblas_int n, const
    rocblas_double_complex *alpha, const rocblas_double_complex *const
    x[], rocblas_int incx, const rocblas_double_complex *const y[],
    rocblas_int incy, rocblas_double_complex *const A[], rocblas_int lda,
    rocblas_int batch_count)
```

```
rocblas_status rocblas_cgerc_batched(rocblas_handle handle, rocblas_int m, rocblas_int n, const
    rocblas_float_complex *alpha, const rocblas_float_complex *const x[],
    rocblas_int incx, const rocblas_float_complex *const y[], rocblas_int
    incy, rocblas_float_complex *const A[], rocblas_int lda, rocblas_int
    batch_count)
```

```
rocblas_status rocblas_zgerc_batched(rocblas_handle handle, rocblas_int m, rocblas_int n, const
    rocblas_double_complex *alpha, const rocblas_double_complex *const
    x[], rocblas_int incx, const rocblas_double_complex *const y[],
    rocblas_int incy, rocblas_double_complex *const A[], rocblas_int lda,
    rocblas_int batch_count)
```

BLAS Level 2 API

ger_batched, geru_batched, gerc_batched perform a batch of the matrix-vector operations:

```
A := A + alpha*x*y**T , OR
A := A + alpha*x*y**H for gerc
where (A_i, x_i, y_i) is the i-th instance of the batch.
alpha is a scalar, x_i and y_i are vectors and A_i is an
m by n matrix, for i = 1, ..., batch_count.
```

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **m** – [in] [rocblas_int] the number of rows of each matrix A_i.
- **n** – [in] [rocblas_int] the number of columns of each matrix A_i.
- **alpha** – [in] device pointer or host pointer to scalar alpha.
- **x** – [in] device array of device pointers storing each vector x_i.
- **incx** – [in] [rocblas_int] specifies the increment for the elements of each vector x_i.
- **y** – [in] device array of device pointers storing each vector y_i.
- **incy** – [in] [rocblas_int] specifies the increment for the elements of each vector y_i.
- **A** – [inout] device array of device pointers storing each matrix A_i.
- **lda** – [in] [rocblas_int] specifies the leading dimension of each A_i.
- **batch_count** – [in] [rocblas_int] number of instances in the batch.

```
rocblas_status rocblas_sger_strided_batched(rocblas_handle handle, rocblas_int m, rocblas_int n, const float
    *alpha, const float *x, rocblas_int incx, rocblas_stride stridex,
    const float *y, rocblas_int incy, rocblas_stride stridey, float *A,
    rocblas_int lda, rocblas_stride strideA, rocblas_int
    batch_count)
```

```
rocblas_status rocblas_dger_strided_batched(rocblas_handle handle, rocblas_int m, rocblas_int n, const
    double *alpha, const double *x, rocblas_int incx, rocblas_stride
    stridex, const double *y, rocblas_int incy, rocblas_stride
    stridey, double *A, rocblas_int lda, rocblas_stride strideA,
    rocblas_int batch_count)
```

```

rocblas_status rocblas_cgeru_strided_batched(rocblas_handle handle, rocblas_int m, rocblas_int n, const
rocblas_float_complex *alpha, const rocblas_float_complex
*x, rocblas_int incx, rocblas_stride stridex, const
rocblas_float_complex *y, rocblas_int incy, rocblas_stride
stridey, rocblas_float_complex *A, rocblas_int lda,
rocblas_stride strideA, rocblas_int batch_count)

rocblas_status rocblas_zgeru_strided_batched(rocblas_handle handle, rocblas_int m, rocblas_int n, const
rocblas_double_complex *alpha, const
rocblas_double_complex *x, rocblas_int incx, rocblas_stride
stridex, const rocblas_double_complex *y, rocblas_int incy,
rocblas_stride stridey, rocblas_double_complex *A,
rocblas_int lda, rocblas_stride strideA, rocblas_int
batch_count)

rocblas_status rocblas_cgerc_strided_batched(rocblas_handle handle, rocblas_int m, rocblas_int n, const
rocblas_float_complex *alpha, const rocblas_float_complex
*x, rocblas_int incx, rocblas_stride stridex, const
rocblas_float_complex *y, rocblas_int incy, rocblas_stride
stridey, rocblas_float_complex *A, rocblas_int lda,
rocblas_stride strideA, rocblas_int batch_count)

rocblas_status rocblas_zgerc_strided_batched(rocblas_handle handle, rocblas_int m, rocblas_int n, const
rocblas_double_complex *alpha, const
rocblas_double_complex *x, rocblas_int incx, rocblas_stride
stridex, const rocblas_double_complex *y, rocblas_int incy,
rocblas_stride stridey, rocblas_double_complex *A,
rocblas_int lda, rocblas_stride strideA, rocblas_int
batch_count)

```

BLAS Level 2 API

ger_strided_batched, geru_strided_batched, gerc_strided_batched performs the matrix-vector operations:

```

A_i := A_i + alpha*x_i*y_i**T, OR
A_i := A_i + alpha*x_i*y_i**H  for gerc
where (A_i, x_i, y_i) is the i-th instance of the batch.
alpha is a scalar, x_i and y_i are vectors and A_i is an
m by n matrix, for i = 1, ..., batch_count.

```

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **m** – [in] [rocblas_int] the number of rows of each matrix A_i.
- **n** – [in] [rocblas_int] the number of columns of each matrix A_i.
- **alpha** – [in] device pointer or host pointer to scalar alpha.
- **x** – [in] device pointer to the first vector (x_1) in the batch.
- **incx** – [in] [rocblas_int] specifies the increments for the elements of each vector x_i.
- **stridex** – [in] [rocblas_stride] stride from the start of one vector (x_i) and the next one (x_i+1). There are no restrictions placed on stride_x. However, ensure that stride_x is of appropriate size. For a typical case this means stride_x >= m * incx.
- **y** – [inout] device pointer to the first vector (y_1) in the batch.

- **incy** – [in] [rocbas_int] specifies the increment for the elements of each vector y_i .
- **stridey** – [in] [rocbas_stride] stride from the start of one vector (y_i) and the next one (y_{i+1}). There are no restrictions placed on $stride_y$. However, ensure that $stride_y$ is of appropriate size. For a typical case this means $stride_y \geq n * incy$.
- **A** – [inout] device pointer to the first matrix (A_1) in the batch.
- **lda** – [in] [rocbas_int] specifies the leading dimension of each A_i .
- **strideA** – [in] [rocbas_stride] stride from the start of one matrix (A_i) and the next one (A_{i+1})
- **batch_count** – [in] [rocbas_int] number of instances in the batch.

5.5.4 rocbas_Xsbmv + batched, strided_batched

rocbas_status **rocbas_ssbmv**(*rocbas_handle* handle, *rocbas_fill* uplo, *rocbas_int* n, *rocbas_int* k, const float *alpha, const float *A, *rocbas_int* lda, const float *x, *rocbas_int* incx, const float *beta, float *y, *rocbas_int* incy)

rocbas_status **rocbas_dsbmv**(*rocbas_handle* handle, *rocbas_fill* uplo, *rocbas_int* n, *rocbas_int* k, const double *alpha, const double *A, *rocbas_int* lda, const double *x, *rocbas_int* incx, const double *beta, double *y, *rocbas_int* incy)

BLAS Level 2 API

sbmv performs the matrix-vector operation:

$y := \alpha A x + \beta y$
 where α and β are scalars, x and y are n element vectors and
 A should contain an upper or lower triangular n by n symmetric banded matrix.

Parameters

- **handle** – [in] [rocbas_handle] handle to the rocbas library context queue.
- **uplo** – [in] *rocbas_fill* specifies whether the upper ‘*rocbas_fill_upper*’ or lower ‘*rocbas_fill_lower*’
 - if *rocbas_fill_upper*, the lower part of A is not referenced
 - if *rocbas_fill_lower*, the upper part of A is not referenced
- **n** – [in] [rocbas_int]
- **k** – [in] [rocbas_int] specifies the number of sub- and super-diagonals.
- **alpha** – [in] specifies the scalar α .
- **A** – [in] pointer storing matrix A on the GPU.
- **lda** – [in] [rocbas_int] specifies the leading dimension of matrix A .
- **x** – [in] pointer storing vector x on the GPU.
- **incx** – [in] [rocbas_int] specifies the increment for the elements of x .
- **beta** – [in] specifies the scalar β .
- **y** – [out] pointer storing vector y on the GPU.
- **incy** – [in] [rocbas_int] specifies the increment for the elements of y .

```
rocblas_status rocblas_ssbmv_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_int n, rocblas_int k,
    const float *alpha, const float *const A[], rocblas_int lda, const float
    *const x[], rocblas_int incx, const float *beta, float *const y[],
    rocblas_int incy, rocblas_int batch_count)
```

```
rocblas_status rocblas_dsbmv_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_int n, rocblas_int k,
    const double *alpha, const double *const A[], rocblas_int lda, const
    double *const x[], rocblas_int incx, const double *beta, double *const
    y[], rocblas_int incy, rocblas_int batch_count)
```

BLAS Level 2 API

sbmv_batched performs the matrix-vector operation:

```
y_i := alpha*A_i*x_i + beta*y_i
where (A_i, x_i, y_i) is the i-th instance of the batch.
alpha and beta are scalars, x_i and y_i are vectors and A_i is an
n by n symmetric banded matrix, for i = 1, ..., batch_count.
A should contain an upper or lower triangular n by n symmetric banded matrix.
```

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill] specifies whether the upper ‘rocblas_fill_upper’ or lower ‘rocblas_fill_lower’
 - if rocblas_fill_upper, the lower part of A is not referenced
 - if rocblas_fill_lower, the upper part of A is not referenced
- **n** – [in] [rocblas_int] number of rows and columns of each matrix A_i.
- **k** – [in] [rocblas_int] specifies the number of sub- and super-diagonals.
- **alpha** – [in] device pointer or host pointer to scalar alpha.
- **A** – [in] device array of device pointers storing each matrix A_i.
- **lda** – [in] [rocblas_int] specifies the leading dimension of each matrix A_i.
- **x** – [in] device array of device pointers storing each vector x_i.
- **incx** – [in] [rocblas_int] specifies the increment for the elements of each vector x_i.
- **beta** – [in] device pointer or host pointer to scalar beta.
- **y** – [out] device array of device pointers storing each vector y_i.
- **incy** – [in] [rocblas_int] specifies the increment for the elements of each vector y_i.
- **batch_count** – [in] [rocblas_int] number of instances in the batch.

```
rocblas_status rocblas_ssbmv_strided_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_int n,
    rocblas_int k, const float *alpha, const float *A, rocblas_int
    lda, rocblas_stride strideA, const float *x, rocblas_int incx,
    rocblas_stride stridex, const float *beta, float *y, rocblas_int
    incy, rocblas_stride stridey, rocblas_int batch_count)
```

```
rocblas_status rocblas_dsbmv_strided_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_int n,
                                             rocblas_int k, const double *alpha, const double *A,
                                             rocblas_int lda, rocblas_stride strideA, const double *x,
                                             rocblas_int incx, rocblas_stride stridex, const double *beta,
                                             double *y, rocblas_int incy, rocblas_stride stridey, rocblas_int
                                             batch_count)
```

BLAS Level 2 API

sbmv_strided_batched performs the matrix-vector operation:

```
y_i := alpha*A_i*x_i + beta*y_i
where (A_i, x_i, y_i) is the i-th instance of the batch.
alpha and beta are scalars, x_i and y_i are vectors and A_i is an
n by n symmetric banded matrix, for i = 1, ..., batch_count.
A should contain an upper or lower triangular n by n symmetric banded matrix.
```

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill] specifies whether the upper ‘rocblas_fill_upper’ or lower ‘rocblas_fill_lower’
 - if rocblas_fill_upper, the lower part of A is not referenced
 - if rocblas_fill_lower, the upper part of A is not referenced
- **n** – [in] [rocblas_int] number of rows and columns of each matrix A_i.
- **k** – [in] [rocblas_int] specifies the number of sub- and super-diagonals.
- **alpha** – [in] device pointer or host pointer to scalar alpha.
- **A** – [in] Device pointer to the first matrix A_1 on the GPU.
- **lda** – [in] [rocblas_int] specifies the leading dimension of each matrix A_i.
- **strideA** – [in] [rocblas_stride] stride from the start of one matrix (A_i) and the next one (A_{i+1}).
- **x** – [in] Device pointer to the first vector x_1 on the GPU.
- **incx** – [in] [rocblas_int] specifies the increment for the elements of each vector x_i.
- **stridex** – [in] [rocblas_stride] stride from the start of one vector (x_i) and the next one (x_{i+1}). There are no restrictions placed on stridex. However, ensure that stridex is of appropriate size. This typically means stridex >= n * incx. stridex should be non zero.
- **beta** – [in] device pointer or host pointer to scalar beta.
- **y** – [out] Device pointer to the first vector y_1 on the GPU.
- **incy** – [in] [rocblas_int] specifies the increment for the elements of each vector y_i.
- **stridey** – [in] [rocblas_stride] stride from the start of one vector (y_i) and the next one (y_{i+1}). There are no restrictions placed on stridey. However, ensure that stridey is of appropriate size. This typically means stridey >= n * incy. stridey should be non zero.
- **batch_count** – [in] [rocblas_int] number of instances in the batch.

5.5.5 rocblas_Xspmv + batched, strided_batched

rocblas_status **rocblas_sspmv**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_int* n, const float *alpha, const float *A, const float *x, *rocblas_int* incx, const float *beta, float *y, *rocblas_int* incy)

rocblas_status **rocblas_dspmv**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_int* n, const double *alpha, const double *A, const double *x, *rocblas_int* incx, const double *beta, double *y, *rocblas_int* incy)

BLAS Level 2 API

spmv performs the matrix-vector operation:

$y := \alpha A x + \beta y$
 where α and β are scalars, x and y are n element vectors and
 A should contain an upper or lower triangular n by n packed symmetric matrix.

Parameters

- **handle** – [in] [*rocblas_handle*] handle to the rocblas library context queue.
- **uplo** – [in] *rocblas_fill* specifies whether the upper ‘*rocblas_fill_upper*’ or lower ‘*rocblas_fill_lower*’
 - if *rocblas_fill_upper*, the lower part of A is not referenced
 - if *rocblas_fill_lower*, the upper part of A is not referenced
- **n** – [in] [*rocblas_int*]
- **alpha** – [in] specifies the scalar α .
- **A** – [in] pointer storing matrix A on the GPU.
- **x** – [in] pointer storing vector x on the GPU.
- **incx** – [in] [*rocblas_int*] specifies the increment for the elements of x .
- **beta** – [in] specifies the scalar β .
- **y** – [out] pointer storing vector y on the GPU.
- **incy** – [in] [*rocblas_int*] specifies the increment for the elements of y .

rocblas_status **rocblas_sspmv_batched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_int* n, const float *alpha, const float *const A[], const float *const x[], *rocblas_int* incx, const float *beta, float *const y[], *rocblas_int* incy, *rocblas_int* batch_count)

rocblas_status **rocblas_dspmv_batched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_int* n, const double *alpha, const double *const A[], const double *const x[], *rocblas_int* incx, const double *beta, double *const y[], *rocblas_int* incy, *rocblas_int* batch_count)

BLAS Level 2 API

spmv_batched performs the matrix-vector operation:

```

y_i := alpha*A_i*x_i + beta*y_i
where (A_i, x_i, y_i) is the i-th instance of the batch.
alpha and beta are scalars, x_i and y_i are vectors and A_i is an
n by n symmetric matrix, for i = 1, ..., batch_count.
A should contain an upper or lower triangular n by n packed symmetric matrix.

```

Parameters

- **handle** – [in] [rocbas_handle] handle to the rocbas library context queue.
- **uplo** – [in] [rocbas_fill] specifies whether the upper ‘rocbas_fill_upper’ or lower ‘rocbas_fill_lower’
 - if rocbas_fill_upper, the lower part of A is not referenced
 - if rocbas_fill_lower, the upper part of A is not referenced
- **n** – [in] [rocbas_int] number of rows and columns of each matrix A_i.
- **alpha** – [in] device pointer or host pointer to scalar alpha.
- **A** – [in] device array of device pointers storing each matrix A_i.
- **x** – [in] device array of device pointers storing each vector x_i.
- **incx** – [in] [rocbas_int] specifies the increment for the elements of each vector x_i.
- **beta** – [in] device pointer or host pointer to scalar beta.
- **y** – [out] device array of device pointers storing each vector y_i.
- **incy** – [in] [rocbas_int] specifies the increment for the elements of each vector y_i.
- **batch_count** – [in] [rocbas_int] number of instances in the batch.

rocbas_status **rocbas_sspmv_strided_batched**(*rocbas_handle* handle, *rocbas_fill* uplo, *rocbas_int* n, const float *alpha, const float *A, *rocbas_stride* strideA, const float *x, *rocbas_int* incx, *rocbas_stride* stridex, const float *beta, float *y, *rocbas_int* incy, *rocbas_stride* stridey, *rocbas_int* batch_count)

rocbas_status **rocbas_dspmv_strided_batched**(*rocbas_handle* handle, *rocbas_fill* uplo, *rocbas_int* n, const double *alpha, const double *A, *rocbas_stride* strideA, const double *x, *rocbas_int* incx, *rocbas_stride* stridex, const double *beta, double *y, *rocbas_int* incy, *rocbas_stride* stridey, *rocbas_int* batch_count)

BLAS Level 2 API

spmv_strided_batched performs the matrix-vector operation:

```

y_i := alpha*A_i*x_i + beta*y_i
where (A_i, x_i, y_i) is the i-th instance of the batch.
alpha and beta are scalars, x_i and y_i are vectors and A_i is an
n by n symmetric matrix, for i = 1, ..., batch_count.
A should contain an upper or lower triangular n by n packed symmetric matrix.

```

Parameters

- **handle** – [in] [rocbas_handle] handle to the rocbas library context queue.

- **uplo** – [in] [rocblas_fill] specifies whether the upper ‘rocblas_fill_upper’ or lower ‘rocblas_fill_lower’
 - if rocblas_fill_upper, the lower part of A is not referenced
 - if rocblas_fill_lower, the upper part of A is not referenced
- **n** – [in] [rocblas_int] number of rows and columns of each matrix A_i.
- **alpha** – [in] device pointer or host pointer to scalar alpha.
- **A** – [in] Device pointer to the first matrix A₁ on the GPU.
- **strideA** – [in] [rocblas_stride] stride from the start of one matrix (A_i) and the next one (A_{i+1}).
- **x** – [in] Device pointer to the first vector x₁ on the GPU.
- **incx** – [in] [rocblas_int] specifies the increment for the elements of each vector x_i.
- **stridex** – [in] [rocblas_stride] stride from the start of one vector (x_i) and the next one (x_{i+1}). There are no restrictions placed on stridex. However, ensure that stridex is of appropriate size. This typically means stridex >= n * incx. stridex should be non zero.
- **beta** – [in] device pointer or host pointer to scalar beta.
- **y** – [out] Device pointer to the first vector y₁ on the GPU.
- **incy** – [in] [rocblas_int] specifies the increment for the elements of each vector y_i.
- **stridey** – [in] [rocblas_stride] stride from the start of one vector (y_i) and the next one (y_{i+1}). There are no restrictions placed on stridey. However, ensure that stridey is of appropriate size. This typically means stridey >= n * incy. stridey should be non zero.
- **batch_count** – [in] [rocblas_int] number of instances in the batch.

5.5.6 rocblas_Xspr + batched, strided_batched

rocblas_status **rocblas_sspr**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_int* n, const float *alpha, const float *x, *rocblas_int* incx, float *AP)

rocblas_status **rocblas_dspr**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_int* n, const double *alpha, const double *x, *rocblas_int* incx, double *AP)

rocblas_status **rocblas_cspr**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_int* n, const *rocblas_float_complex* *alpha, const *rocblas_float_complex* *x, *rocblas_int* incx, *rocblas_float_complex* *AP)

rocblas_status **rocblas_zspr**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_int* n, const *rocblas_double_complex* *alpha, const *rocblas_double_complex* *x, *rocblas_int* incx, *rocblas_double_complex* *AP)

BLAS Level 2 API

spr performs the matrix-vector operations:

$A := A + \alpha * x * x^T$
 where alpha is a scalar, x is a vector, and A is an
 n by n symmetric matrix, supplied in packed form.

Parameters

- **handle** – [in] [rocbas_handle] handle to the rocbas library context queue.
- **uplo** – [in] [rocbas_fill] specifies whether the upper ‘rocbas_fill_upper’ or lower ‘rocbas_fill_lower’
 - rocbas_fill_upper: The upper triangular part of A is supplied in AP.
 - rocbas_fill_lower: The lower triangular part of A is supplied in AP.
- **n** – [in] [rocbas_int] the number of rows and columns of matrix A. Must be at least 0.
- **alpha** – [in] device pointer or host pointer to scalar alpha.
- **x** – [in] device pointer storing vector x.
- **incx** – [in] [rocbas_int] specifies the increment for the elements of x.
- **AP** – [inout] device pointer storing the packed version of the specified triangular portion of the symmetric matrix A. Of at least size $((n * (n + 1)) / 2)$.

```

if uplo == rocbas_fill_upper:
    The upper triangular portion of the symmetric matrix A is
    supplied.
    The matrix is compacted so that AP contains the triangular
    portion
    column-by-column
    so that:
    AP(0) = A(0,0)
    AP(1) = A(0,1)
    AP(2) = A(1,1), etc.
    Ex: (rocbas_fill_upper; n = 4)
        1 2 4 7
        2 3 5 8  ----> [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
        4 5 6 9
        7 8 9 0

if uplo == rocbas_fill_lower:
    The lower triangular portion of the symmetric matrix A is
    supplied.
    The matrix is compacted so that AP contains the triangular
    portion
    column-by-column
    so that:
    AP(0) = A(0,0)
    AP(1) = A(1,0)
    AP(2) = A(2,1), etc.
    Ex: (rocbas_fill_lower; n = 4)
        1 2 3 4
        2 5 6 7  ----> [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
        3 6 8 9
        4 7 9 0

```

rocbas_status **rocbas_sspr_batched**(*rocbas_handle* handle, *rocbas_fill* uplo, *rocbas_int* n, const float *alpha, const float *const x[], *rocbas_int* incx, float *const AP[], *rocbas_int* batch_count)

rocbas_status **rocbas_dspr_batched**(*rocbas_handle* handle, *rocbas_fill* uplo, *rocbas_int* n, const double *alpha, const double *const x[], *rocbas_int* incx, double *const AP[], *rocbas_int* batch_count)

```
rocblas_status rocblas_cspr_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_int n, const
    rocblas_float_complex *alpha, const rocblas_float_complex *const x[],
    rocblas_int incx, rocblas_float_complex *const AP[], rocblas_int
    batch_count)
```

```
rocblas_status rocblas_zspr_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_int n, const
    rocblas_double_complex *alpha, const rocblas_double_complex *const
    x[], rocblas_int incx, rocblas_double_complex *const AP[], rocblas_int
    batch_count)
```

BLAS Level 2 API

spr_batched performs the matrix-vector operations:

$A_i := A_i + \alpha x_i x_i^T$
 where α **is** a scalar, x_i **is** a vector, **and** A_i **is** an
 n by n symmetric matrix, supplied **in** packed form, **for** $i = 1, \dots, \text{batch_count}$.

Parameters

- **handle** – [**in**] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [**in**] [rocblas_fill] specifies whether the upper ‘rocblas_fill_upper’ or lower ‘rocblas_fill_lower’
 - rocblas_fill_upper: The upper triangular part of each A_i is supplied in AP.
 - rocblas_fill_lower: The lower triangular part of each A_i is supplied in AP.
- **n** – [**in**] [rocblas_int] the number of rows and columns of each matrix A_i . Must be at least 0.
- **alpha** – [**in**] device pointer or host pointer to scalar α .
- **x** – [**in**] device array of device pointers storing each vector x_i .
- **incx** – [**in**] [rocblas_int] specifies the increment for the elements of each x_i .
- **AP** – [**inout**] device array of device pointers storing the packed version of the specified triangular portion of each symmetric matrix A_i of at least size $((n * (n + 1)) / 2)$. Array is of at least size batch_count.

```
if uplo == rocblas_fill_upper:
    The upper triangular portion of each symmetric matrix A_i is
    supplied.
    The matrix is compacted so that AP contains the triangular
    portion
    column-by-column
    so that:
    AP(0) = A(0,0)
    AP(1) = A(0,1)
    AP(2) = A(1,1), etc.
    Ex: (rocblas_fill_upper; n = 4)
        1 2 4 7
        2 3 5 8  ----> [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
        4 5 6 9
        7 8 9 0
```

(continues on next page)

(continued from previous page)

```

if uplo == rocblas_fill_lower:
    The lower triangular portion of each symmetric matrix A_i is
    supplied.
    The matrix is compacted so that AP contains the triangular
    portion
    column-by-column
    so that:
    AP(0) = A(0,0)
    AP(1) = A(1,0)
    AP(2) = A(2,1), etc.
    Ex: (rocblas_fill_lower; n = 4)
        1 2 3 4
        2 5 6 7  -----> [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
        3 6 8 9
        4 7 9 0

```

- **batch_count** – [in] [rocblas_int] number of instances in the batch.

rocblas_status **rocblas_sspr_strided_batched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_int* n, const float *alpha, const float *x, *rocblas_int* incx, *rocblas_stride* stride_x, float *AP, *rocblas_stride* stride_A, *rocblas_int* batch_count)

rocblas_status **rocblas_dspr_strided_batched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_int* n, const double *alpha, const double *x, *rocblas_int* incx, *rocblas_stride* stride_x, double *AP, *rocblas_stride* stride_A, *rocblas_int* batch_count)

rocblas_status **rocblas_cspr_strided_batched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_int* n, const *rocblas_float_complex* *alpha, const *rocblas_float_complex* *x, *rocblas_int* incx, *rocblas_stride* stride_x, *rocblas_float_complex* *AP, *rocblas_stride* stride_A, *rocblas_int* batch_count)

rocblas_status **rocblas_zspr_strided_batched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_int* n, const *rocblas_double_complex* *alpha, const *rocblas_double_complex* *x, *rocblas_int* incx, *rocblas_stride* stride_x, *rocblas_double_complex* *AP, *rocblas_stride* stride_A, *rocblas_int* batch_count)

BLAS Level 2 API

spr_strided_batched performs the matrix-vector operations:

$A_i := A_i + \alpha x_i x_i^T$
 where alpha is a scalar, x_i is a vector, and A_i is an
 n by n symmetric matrix, supplied in packed form, for $i = 1, \dots, \text{batch_count}$.

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill] specifies whether the upper ‘rocblas_fill_upper’ or lower ‘rocblas_fill_lower’

- `rocblas_fill_upper`: The upper triangular part of each A_i is supplied in AP.
- `rocblas_fill_lower`: The lower triangular part of each A_i is supplied in AP.
- **n** – [in] [rocblas_int] the number of rows and columns of each matrix A_i . Must be at least 0.
- **alpha** – [in] device pointer or host pointer to scalar alpha.
- **x** – [in] device pointer pointing to the first vector (x_1).
- **incx** – [in] [rocblas_int] specifies the increment for the elements of each x_i .
- **stride_x** – [in] [rocblas_stride] stride from the start of one vector (x_i) and the next one (x_{i+1}).
- **AP** – [inout] device pointer storing the packed version of the specified triangular portion of each symmetric matrix A_i . Points to the first A_1 .

```

if uplo == rocblas_fill_upper:
    The upper triangular portion of each symmetric matrix  $A_i$  is
    supplied.
    The matrix is compacted so that AP contains the triangular
    portion
    column-by-column
    so that:
    AP(0) = A(0,0)
    AP(1) = A(0,1)
    AP(2) = A(1,1), etc.
    Ex: (rocblas_fill_upper; n = 4)
        1 2 4 7
        2 3 5 8  -----> [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
        4 5 6 9
        7 8 9 0

if uplo == rocblas_fill_lower:
    The lower triangular portion of each symmetric matrix  $A_i$  is
    supplied.
    The matrix is compacted so that AP contains the triangular
    portion
    column-by-column
    so that:
    AP(0) = A(0,0)
    AP(1) = A(1,0)
    AP(2) = A(2,1), etc.
    Ex: (rocblas_fill_lower; n = 4)
        1 2 3 4
        2 5 6 7  -----> [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
        3 6 8 9
        4 7 9 0

```

- **stride_A** – [in] [rocblas_stride] stride from the start of one (A_i) and the next (A_{i+1}).
- **batch_count** – [in] [rocblas_int] number of instances in the batch.

5.5.7 rocblas_Xspr2 + batched, strided_batched

rocblas_status **rocblas_sspr2**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_int* n, const float *alpha, const float *x, *rocblas_int* incx, const float *y, *rocblas_int* incy, float *AP)

rocblas_status **rocblas_dspr2**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_int* n, const double *alpha, const double *x, *rocblas_int* incx, const double *y, *rocblas_int* incy, double *AP)

BLAS Level 2 API

spr2 performs the matrix-vector operation:

$A := A + \alpha x y^T + \alpha y x^T$
 where α is a scalar, x and y are vectors, and A is an n by n symmetric matrix, supplied in packed form.

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill] specifies whether the upper ‘rocblas_fill_upper’ or lower ‘rocblas_fill_lower’
 - rocblas_fill_upper: The upper triangular part of A is supplied in AP.
 - rocblas_fill_lower: The lower triangular part of A is supplied in AP.
- **n** – [in] [rocblas_int] the number of rows and columns of matrix A. Must be at least 0.
- **alpha** – [in] device pointer or host pointer to scalar alpha.
- **x** – [in] device pointer storing vector x.
- **incx** – [in] [rocblas_int] specifies the increment for the elements of x.
- **y** – [in] device pointer storing vector y.
- **incy** – [in] [rocblas_int] specifies the increment for the elements of y.
- **AP** – [inout] device pointer storing the packed version of the specified triangular portion of the symmetric matrix A. Of at least size $((n * (n + 1)) / 2)$.

```
if uplo == rocblas_fill_upper:
    The upper triangular portion of the symmetric matrix A is
    supplied.
    The matrix is compacted so that AP contains the triangular
    portion
    column-by-column
    so that:
    AP(0) = A(0,0)
    AP(1) = A(0,1)
    AP(2) = A(1,1), etc.
    Ex: (rocblas_fill_upper; n = 4)
        1 2 4 7
        2 3 5 8  ----> [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
        4 5 6 9
        7 8 9 0

if uplo == rocblas_fill_lower:
    The lower triangular portion of the symmetric matrix A is
```

(continues on next page)

(continued from previous page)

```

↪supplied.
    The matrix is compacted so that AP contains the triangular_
↪portion
    column-by-column
    so that:
    AP(0) = A(0,0)
    AP(1) = A(1,0)
    AP(n) = A(2,1), etc.
    Ex: (rocblas_fill_lower; n = 4)
        1 2 3 4
        2 5 6 7  -----> [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
        3 6 8 9
        4 7 9 0

```

```
rocblas_status rocblas_sspr2_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_int n, const float
                                     *alpha, const float *const x[], rocblas_int incx, const float *const y[],
                                     rocblas_int incy, float *const AP[], rocblas_int batch_count)
```

```
rocblas_status rocblas_dspr2_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_int n, const double
                                     *alpha, const double *const x[], rocblas_int incx, const double *const
                                     y[], rocblas_int incy, double *const AP[], rocblas_int batch_count)
```

BLAS Level 2 API

spr2_batched performs the matrix-vector operation:

```

A_i := A_i + alpha*x_i*y_i**T + alpha*y_i*x_i**T
where alpha is a scalar, x_i and y_i are vectors, and A_i is an
n by n symmetric matrix, supplied in packed form, for i = 1, ..., batch_count.

```

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill] specifies whether the upper ‘rocblas_fill_upper’ or lower ‘rocblas_fill_lower’
 - rocblas_fill_upper: The upper triangular part of each A_i is supplied in AP.
 - rocblas_fill_lower: The lower triangular part of each A_i is supplied in AP.
- **n** – [in] [rocblas_int] the number of rows and columns of each matrix A_i. Must be at least 0.
- **alpha** – [in] device pointer or host pointer to scalar alpha.
- **x** – [in] device array of device pointers storing each vector x_i.
- **incx** – [in] [rocblas_int] specifies the increment for the elements of each x_i.
- **y** – [in] device array of device pointers storing each vector y_i.
- **incy** – [in] [rocblas_int] specifies the increment for the elements of each y_i.
- **AP** – [inout] device array of device pointers storing the packed version of the specified triangular portion of each symmetric matrix A_i of at least size $((n * (n + 1)) / 2)$. Array is of at least size batch_count.

```

if uplo == rocblas_fill_upper:
    The upper triangular portion of each symmetric matrix A_i is
    supplied.
    The matrix is compacted so that AP contains the triangular
    portion
    column-by-column
    so that:
    AP(0) = A(0,0)
    AP(1) = A(0,1)
    AP(2) = A(1,1), etc.
    Ex: (rocblas_fill_upper; n = 4)
        1 2 4 7
        2 3 5 8  ----> [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
        4 5 6 9
        7 8 9 0

if uplo == rocblas_fill_lower:
    The lower triangular portion of each symmetric matrix A_i is
    supplied.
    The matrix is compacted so that AP contains the triangular
    portion
    column-by-column
    so that:
    AP(0) = A(0,0)
    AP(1) = A(1,0)
    AP(n) = A(2,1), etc.
    Ex: (rocblas_fill_lower; n = 4)
        1 2 3 4
        2 5 6 7  ----> [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
        3 6 8 9
        4 7 9 0

```

- **batch_count** – [in] [rocblas_int] number of instances in the batch.

rocblas_status **rocblas_sspr2_strided_batched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_int* n, const float *alpha, const float *x, *rocblas_int* incx, *rocblas_stride* stride_x, const float *y, *rocblas_int* incy, *rocblas_stride* stride_y, float *AP, *rocblas_stride* stride_A, *rocblas_int* batch_count)

rocblas_status **rocblas_dspr2_strided_batched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_int* n, const double *alpha, const double *x, *rocblas_int* incx, *rocblas_stride* stride_x, const double *y, *rocblas_int* incy, *rocblas_stride* stride_y, double *AP, *rocblas_stride* stride_A, *rocblas_int* batch_count)

BLAS Level 2 API

spr2_strided_batched performs the matrix-vector operation:

$A_i := A_i + \alpha x_i y_i^{**T} + \alpha y_i x_i^{**T}$
 where alpha is a scalar, x_i and y_i are vectors, and A_i is an
 n by n symmetric matrix, supplied in packed form, for i = 1, ..., batch_count.

Parameters

- **handle** – [in] [roclblas_handle] handle to the roclblas library context queue.
- **uplo** – [in] [roclblas_fill] specifies whether the upper ‘roclblas_fill_upper’ or lower ‘roclblas_fill_lower’
 - roclblas_fill_upper: The upper triangular part of each A_i is supplied in AP.
 - roclblas_fill_lower: The lower triangular part of each A_i is supplied in AP.
- **n** – [in] [roclblas_int] the number of rows and columns of each matrix A_i. Must be at least 0.
- **alpha** – [in] device pointer or host pointer to scalar alpha.
- **x** – [in] device pointer pointing to the first vector (x_1).
- **incx** – [in] [roclblas_int] specifies the increment for the elements of each x_i.
- **stride_x** – [in] [roclblas_stride] stride from the start of one vector (x_i) and the next one (x_i+1).
- **y** – [in] device pointer pointing to the first vector (y_1).
- **incy** – [in] [roclblas_int] specifies the increment for the elements of each y_i.
- **stride_y** – [in] [roclblas_stride] stride from the start of one vector (y_i) and the next one (y_i+1).
- **AP** – [inout] device pointer storing the packed version of the specified triangular portion of each symmetric matrix A_i. Points to the first A_1.

```

if uplo == roclblas_fill_upper:
    The upper triangular portion of each symmetric matrix A_i is
    supplied.
    The matrix is compacted so that AP contains the triangular
    portion
    column-by-column
    so that:
    AP(0) = A(0,0)
    AP(1) = A(0,1)
    AP(2) = A(1,1), etc.
    Ex: (roclblas_fill_upper; n = 4)
        1 2 4 7
        2 3 5 8  -----> [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
        4 5 6 9
        7 8 9 0

if uplo == roclblas_fill_lower:
    The lower triangular portion of each symmetric matrix A_i is
    supplied.
    The matrix is compacted so that AP contains the triangular
    portion
    column-by-column
    so that:
    AP(0) = A(0,0)
    AP(1) = A(1,0)
    AP(n) = A(2,1), etc.
    Ex: (roclblas_fill_lower; n = 4)
        1 2 3 4

```

(continues on next page)

(continued from previous page)

```

2 5 6 7  -----> [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
3 6 8 9
4 7 9 0

```

- **stride_A** – [in] [rocblas_stride] stride from the start of one (A_i) and the next (A_i+1).
- **batch_count** – [in] [rocblas_int] number of instances in the batch.

5.5.8 rocblas_Xsymv + batched, strided_batched

rocblas_status **rocblas_ssymv**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_int* n, const float *alpha, const float *A, *rocblas_int* lda, const float *x, *rocblas_int* incx, const float *beta, float *y, *rocblas_int* incy)

rocblas_status **rocblas_dsymv**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_int* n, const double *alpha, const double *A, *rocblas_int* lda, const double *x, *rocblas_int* incx, const double *beta, double *y, *rocblas_int* incy)

rocblas_status **rocblas_csymv**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_int* n, const *rocblas_float_complex* *alpha, const *rocblas_float_complex* *A, *rocblas_int* lda, const *rocblas_float_complex* *x, *rocblas_int* incx, const *rocblas_float_complex* *beta, *rocblas_float_complex* *y, *rocblas_int* incy)

rocblas_status **rocblas_zsymv**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_int* n, const *rocblas_double_complex* *alpha, const *rocblas_double_complex* *A, *rocblas_int* lda, const *rocblas_double_complex* *x, *rocblas_int* incx, const *rocblas_double_complex* *beta, *rocblas_double_complex* *y, *rocblas_int* incy)

BLAS Level 2 API

symv 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 should contain an upper or lower triangular n by n symmetric matrix.

```

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill] specifies whether the upper ‘rocblas_fill_upper’ or lower ‘rocblas_fill_lower’
 - if rocblas_fill_upper, the lower part of A is not referenced.
 - if rocblas_fill_lower, the upper part of A is not referenced.
- **n** – [in] [rocblas_int]
- **alpha** – [in] specifies the scalar alpha.
- **A** – [in] pointer storing matrix A on the GPU
- **lda** – [in] [rocblas_int] specifies the leading dimension of A.
- **x** – [in] pointer storing vector x on the GPU.
- **incx** – [in] [rocblas_int] specifies the increment for the elements of x.

- **beta** – [in] specifies the scalar beta
- **y** – [out] pointer storing vector y on the GPU.
- **incy** – [in] [rocblas_int] specifies the increment for the elements of y.

```
rocblas_status rocblas_ssymv_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_int n, const float
    *alpha, const float *const A[], rocblas_int lda, const float *const x[],
    rocblas_int incx, const float *beta, float *const y[], rocblas_int incy,
    rocblas_int batch_count)
```

```
rocblas_status rocblas_dsymv_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_int n, const double
    *alpha, const double *const A[], rocblas_int lda, const double *const x[],
    rocblas_int incx, const double *beta, double *const y[], rocblas_int incy,
    rocblas_int batch_count)
```

```
rocblas_status rocblas_csymv_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_int n, const
    rocblas_float_complex *alpha, const rocblas_float_complex *const A[],
    rocblas_int lda, const rocblas_float_complex *const x[], rocblas_int
    incx, const rocblas_float_complex *beta, rocblas_float_complex *const
    y[], rocblas_int incy, rocblas_int batch_count)
```

```
rocblas_status rocblas_zsymv_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_int n, const
    rocblas_double_complex *alpha, const rocblas_double_complex *const
    A[], rocblas_int lda, const rocblas_double_complex *const x[],
    rocblas_int incx, const rocblas_double_complex *beta,
    rocblas_double_complex *const y[], rocblas_int incy, rocblas_int
    batch_count)
```

BLAS Level 2 API

symv_batched performs the matrix-vector operation:

```
y_i := alpha*A_i*x_i + beta*y_i
where (A_i, x_i, y_i) is the i-th instance of the batch.
alpha and beta are scalars, x_i and y_i are vectors and A_i is an
n by n symmetric matrix, for i = 1, ..., batch_count.
A_i should contain an upper or lower triangular symmetric matrix
and the opposing triangular part of A is not referenced.
```

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue
- **uplo** – [in] [rocblas_fill] specifies whether the upper ‘rocblas_fill_upper’ or lower ‘rocblas_fill_lower’
 - if rocblas_fill_upper, the lower part of A is not referenced.
 - if rocblas_fill_lower, the upper part of A is not referenced.
- **n** – [in] [rocblas_int] number of rows and columns of each matrix A_i.
- **alpha** – [in] device pointer or host pointer to scalar alpha.
- **A** – [in] device array of device pointers storing each matrix A_i.
- **lda** – [in] [rocblas_int] specifies the leading dimension of each matrix A_i.
- **x** – [in] device array of device pointers storing each vector x_i.

- **incx** – [in] [rocbas_int] specifies the increment for the elements of each vector x_i .
- **beta** – [in] device pointer or host pointer to scalar beta.
- **y** – [out] device array of device pointers storing each vector y_i .
- **incy** – [in] [rocbas_int] specifies the increment for the elements of each vector y_i .
- **batch_count** – [in] [rocbas_int] number of instances in the batch.

rocbas_status **rocbas_ssymv_strided_batched**(*rocbas_handle* handle, *rocbas_fill* uplo, *rocbas_int* n, const float *alpha, const float *A, *rocbas_int* lda, *rocbas_stride* strideA, const float *x, *rocbas_int* incx, *rocbas_stride* stridex, const float *beta, float *y, *rocbas_int* incy, *rocbas_stride* stridey, *rocbas_int* batch_count)

rocbas_status **rocbas_dsymv_strided_batched**(*rocbas_handle* handle, *rocbas_fill* uplo, *rocbas_int* n, const double *alpha, const double *A, *rocbas_int* lda, *rocbas_stride* strideA, const double *x, *rocbas_int* incx, *rocbas_stride* stridex, const double *beta, double *y, *rocbas_int* incy, *rocbas_stride* stridey, *rocbas_int* batch_count)

rocbas_status **rocbas_csymv_strided_batched**(*rocbas_handle* handle, *rocbas_fill* uplo, *rocbas_int* n, const *rocbas_float_complex* *alpha, const *rocbas_float_complex* *A, *rocbas_int* lda, *rocbas_stride* strideA, const *rocbas_float_complex* *x, *rocbas_int* incx, *rocbas_stride* stridex, const *rocbas_float_complex* *beta, *rocbas_float_complex* *y, *rocbas_int* incy, *rocbas_stride* stridey, *rocbas_int* batch_count)

rocbas_status **rocbas_zsymv_strided_batched**(*rocbas_handle* handle, *rocbas_fill* uplo, *rocbas_int* n, const *rocbas_double_complex* *alpha, const *rocbas_double_complex* *A, *rocbas_int* lda, *rocbas_stride* strideA, const *rocbas_double_complex* *x, *rocbas_int* incx, *rocbas_stride* stridex, const *rocbas_double_complex* *beta, *rocbas_double_complex* *y, *rocbas_int* incy, *rocbas_stride* stridey, *rocbas_int* batch_count)

BLAS Level 2 API

symv_strided_batched performs the matrix-vector operation:

$y_i := \alpha A_i x_i + \beta y_i$
 where (A_i, x_i, y_i) **is** the i -th instance of the batch.
 α **and** β are scalars, x_i **and** y_i are vectors **and** A_i **is** an n by n symmetric matrix, **for** $i = 1, \dots, \text{batch_count}$.
 A should contain an upper **or** lower triangular symmetric matrix **and** the opposing triangular part of A **is not** referenced.

Parameters

- **handle** – [in] [rocbas_handle] handle to the rocbas library context queue
- **uplo** – [in] [rocbas_fill] specifies whether the upper ‘rocbas_fill_upper’ or lower ‘rocbas_fill_lower’
 - if rocbas_fill_upper, the lower part of A is not referenced

- if `rocblas_fill_lower`, the upper part of `A` is not referenced
- **n** – [in] [rocblas_int] number of rows and columns of each matrix `A_i`.
- **alpha** – [in] device pointer or host pointer to scalar alpha.
- **A** – [in] Device pointer to the first matrix `A_1` on the GPU.
- **lda** – [in] [rocblas_int] specifies the leading dimension of each matrix `A_i`.
- **strideA** – [in] [rocblas_stride] stride from the start of one matrix (`A_i`) and the next one (`A_{i+1}`).
- **x** – [in] Device pointer to the first vector `x_1` on the GPU.
- **incx** – [in] [rocblas_int] specifies the increment for the elements of each vector `x_i`.
- **stridx** – [in] [rocblas_stride] stride from the start of one vector (`x_i`) and the next one (`x_{i+1}`). There are no restrictions placed on `stride_x`. However, ensure that `stridx` is of appropriate size. This typically means `stridx >= n * incx`. `stridx` should be non zero.
- **beta** – [in] device pointer or host pointer to scalar beta.
- **y** – [out] Device pointer to the first vector `y_1` on the GPU.
- **incy** – [in] [rocblas_int] specifies the increment for the elements of each vector `y_i`.
- **stridey** – [in] [rocblas_stride] stride from the start of one vector (`y_i`) and the next one (`y_{i+1}`). There are no restrictions placed on `stride_y`. However, ensure that `stridey` is of appropriate size. This typically means `stridey >= n * incy`. `stridey` should be non zero.
- **batch_count** – [in] [rocblas_int] number of instances in the batch.

5.5.9 rocblas_Xsyr + batched, strided_batched

rocblas_status **rocblas_ssyr**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_int* n, const float *alpha, const float *x, *rocblas_int* incx, float *A, *rocblas_int* lda)

rocblas_status **rocblas_dsyr**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_int* n, const double *alpha, const double *x, *rocblas_int* incx, double *A, *rocblas_int* lda)

rocblas_status **rocblas_csyr**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_int* n, const *rocblas_float_complex* *alpha, const *rocblas_float_complex* *x, *rocblas_int* incx, *rocblas_float_complex* *A, *rocblas_int* lda)

rocblas_status **rocblas_zsyr**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_int* n, const *rocblas_double_complex* *alpha, const *rocblas_double_complex* *x, *rocblas_int* incx, *rocblas_double_complex* *A, *rocblas_int* lda)

BLAS Level 2 API

`syr` performs the matrix-vector operations:

`A := A + alpha*x*x**T`
 where `alpha` is a scalar, `x` is a vector, and `A` is an
`n` by `n` symmetric matrix.

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.

- **uplo** – [in] [rocblas_fill] specifies whether the upper ‘rocblas_fill_upper’ or lower ‘rocblas_fill_lower’
 - if rocblas_fill_upper, the lower part of A is not referenced
 - if rocblas_fill_lower, the upper part of A is not referenced
- **n** – [in] [rocblas_int] the number of rows and columns of matrix A.
- **alpha** – [in] device pointer or host pointer to scalar alpha.
- **x** – [in] device pointer storing vector x.
- **incx** – [in] [rocblas_int] specifies the increment for the elements of x.
- **A** – [inout] device pointer storing matrix A.
- **lda** – [in] [rocblas_int] specifies the leading dimension of A.

rocblas_status **rocblas_ssyrbatched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_int* n, const float *alpha, const float *const x[], *rocblas_int* incx, float *const A[], *rocblas_int* lda, *rocblas_int* batch_count)

rocblas_status **rocblas_dsyrbatched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_int* n, const double *alpha, const double *const x[], *rocblas_int* incx, double *const A[], *rocblas_int* lda, *rocblas_int* batch_count)

rocblas_status **rocblas_csyrbatched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_int* n, const *rocblas_float_complex* *alpha, const *rocblas_float_complex* *const x[], *rocblas_int* incx, *rocblas_float_complex* *const A[], *rocblas_int* lda, *rocblas_int* batch_count)

rocblas_status **rocblas_zsyrbatched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_int* n, const *rocblas_double_complex* *alpha, const *rocblas_double_complex* *const x[], *rocblas_int* incx, *rocblas_double_complex* *const A[], *rocblas_int* lda, *rocblas_int* batch_count)

BLAS Level 2 API

syrbatched performs a batch of matrix-vector operations:

```
A[i] := A[i] + alpha*x[i]*x[i]**T
where alpha is a scalar, x is an array of vectors, and A is an array of
n by n symmetric matrices, for i = 1 , ... , batch_count.
```

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill] specifies whether the upper ‘rocblas_fill_upper’ or lower ‘rocblas_fill_lower’
 - if rocblas_fill_upper, the lower part of A is not referenced
 - if rocblas_fill_lower, the upper part of A is not referenced
- **n** – [in] [rocblas_int] the number of rows and columns of matrix A.
- **alpha** – [in] device pointer or host pointer to scalar alpha.
- **x** – [in] device array of device pointers storing each vector x_i.
- **incx** – [in] [rocblas_int] specifies the increment for the elements of each x_i.

- **A** – [inout] device array of device pointers storing each matrix A_i .
- **lda** – [in] [rocblas_int] specifies the leading dimension of each A_i .
- **batch_count** – [in] [rocblas_int] number of instances in the batch.

rocblas_status **rocblas_ssytr_strided_batched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_int* n, const float *alpha, const float *x, *rocblas_int* incx, *rocblas_stride* stridex, float *A, *rocblas_int* lda, *rocblas_stride* strideA, *rocblas_int* batch_count)

rocblas_status **rocblas_dsyr_strided_batched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_int* n, const double *alpha, const double *x, *rocblas_int* incx, *rocblas_stride* stridex, double *A, *rocblas_int* lda, *rocblas_stride* strideA, *rocblas_int* batch_count)

rocblas_status **rocblas_csyr_strided_batched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_int* n, const *rocblas_float_complex* *alpha, const *rocblas_float_complex* *x, *rocblas_int* incx, *rocblas_stride* stridex, *rocblas_float_complex* *A, *rocblas_int* lda, *rocblas_stride* strideA, *rocblas_int* batch_count)

rocblas_status **rocblas_zsyr_strided_batched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_int* n, const *rocblas_double_complex* *alpha, const *rocblas_double_complex* *x, *rocblas_int* incx, *rocblas_stride* stridex, *rocblas_double_complex* *A, *rocblas_int* lda, *rocblas_stride* strideA, *rocblas_int* batch_count)

BLAS Level 2 API

sytr_strided_batched performs the matrix-vector operations:

$A[i] := A[i] + \alpha * x[i] * x[i]^T$
 where α is a scalar, vectors, and A is an array of
 n by n symmetric matrices, for $i = 1, \dots, \text{batch_count}$.

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill] specifies whether the upper ‘rocblas_fill_upper’ or lower ‘rocblas_fill_lower’
 - if rocblas_fill_upper, the lower part of A is not referenced
 - if rocblas_fill_lower, the upper part of A is not referenced
- **n** – [in] [rocblas_int] the number of rows and columns of each matrix A .
- **alpha** – [in] device pointer or host pointer to scalar α .
- **x** – [in] device pointer to the first vector x_1 .
- **incx** – [in] [rocblas_int] specifies the increment for the elements of each x_i .
- **stridex** – [in] [rocblas_stride] specifies the pointer increment between vectors (x_i) and (x_{i+1}).
- **A** – [inout] device pointer to the first matrix A_1 .
- **lda** – [in] [rocblas_int] specifies the leading dimension of each A_i .

- **strideA** – [in] [rocbas_stride] stride from the start of one matrix (A_i) and the next one (A_{i+1}).
- **batch_count** – [in] [rocbas_int] number of instances in the batch.

5.5.10 rocbas_Xsyr2 + batched, strided_batched

rocbas_status **rocbas_ssyr2**(*rocbas_handle* handle, *rocbas_fill* uplo, *rocbas_int* n, const float *alpha, const float *x, *rocbas_int* incx, const float *y, *rocbas_int* incy, float *A, *rocbas_int* lda)

rocbas_status **rocbas_dsyr2**(*rocbas_handle* handle, *rocbas_fill* uplo, *rocbas_int* n, const double *alpha, const double *x, *rocbas_int* incx, const double *y, *rocbas_int* incy, double *A, *rocbas_int* lda)

rocbas_status **rocbas_csyr2**(*rocbas_handle* handle, *rocbas_fill* uplo, *rocbas_int* n, const *rocbas_float_complex* *alpha, const *rocbas_float_complex* *x, *rocbas_int* incx, const *rocbas_float_complex* *y, *rocbas_int* incy, *rocbas_float_complex* *A, *rocbas_int* lda)

rocbas_status **rocbas_zsyr2**(*rocbas_handle* handle, *rocbas_fill* uplo, *rocbas_int* n, const *rocbas_double_complex* *alpha, const *rocbas_double_complex* *x, *rocbas_int* incx, const *rocbas_double_complex* *y, *rocbas_int* incy, *rocbas_double_complex* *A, *rocbas_int* lda)

BLAS Level 2 API

syr2 performs the matrix-vector operations:

$A := A + \alpha x y^T + \alpha y x^T$
 where α is a scalar, x and y are vectors, and A is an n by n symmetric matrix.

Parameters

- **handle** – [in] [rocbas_handle] handle to the rocbas library context queue.
- **uplo** – [in] [rocbas_fill] specifies whether the upper ‘rocbas_fill_upper’ or lower ‘rocbas_fill_lower’
 - if rocbas_fill_upper, the lower part of A is not referenced
 - if rocbas_fill_lower, the upper part of A is not referenced
- **n** – [in] [rocbas_int] the number of rows and columns of matrix A.
- **alpha** – [in] device pointer or host pointer to scalar alpha.
- **x** – [in] device pointer storing vector x.
- **incx** – [in] [rocbas_int] specifies the increment for the elements of x.
- **y** – [in] device pointer storing vector y.
- **incy** – [in] [rocbas_int] specifies the increment for the elements of y.
- **A** – [inout] device pointer storing matrix A.
- **lda** – [in] [rocbas_int] specifies the leading dimension of A.

```
rocblas_status rocblas_ssyrr2_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_int n, const float
    *alpha, const float *const x[], rocblas_int incx, const float *const y[],
    rocblas_int incy, float *const A[], rocblas_int lda, rocblas_int
    batch_count)
```

```
rocblas_status rocblas_dsyr2_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_int n, const double
    *alpha, const double *const x[], rocblas_int incx, const double *const
    y[], rocblas_int incy, double *const A[], rocblas_int lda, rocblas_int
    batch_count)
```

```
rocblas_status rocblas_csyr2_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_int n, const
    rocblas_float_complex *alpha, const rocblas_float_complex *const x[],
    rocblas_int incx, const rocblas_float_complex *const y[], rocblas_int
    incy, rocblas_float_complex *const A[], rocblas_int lda, rocblas_int
    batch_count)
```

```
rocblas_status rocblas_zsyr2_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_int n, const
    rocblas_double_complex *alpha, const rocblas_double_complex *const
    x[], rocblas_int incx, const rocblas_double_complex *const y[],
    rocblas_int incy, rocblas_double_complex *const A[], rocblas_int lda,
    rocblas_int batch_count)
```

BLAS Level 2 API

syr2_batched performs a batch of matrix-vector operations:

$A[i] := A[i] + \alpha x[i] y[i]^T + \alpha y[i] x[i]^T$
 where α is a scalar, $x[i]$ and $y[i]$ are vectors, and $A[i]$ is a
 n by n symmetric matrix, for $i = 1, \dots, \text{batch_count}$.

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill] specifies whether the upper ‘rocblas_fill_upper’ or lower ‘rocblas_fill_lower’
 - if rocblas_fill_upper, the lower part of A is not referenced
 - if rocblas_fill_lower, the upper part of A is not referenced
- **n** – [in] [rocblas_int] the number of rows and columns of matrix A.
- **alpha** – [in] device pointer or host pointer to scalar alpha.
- **x** – [in] device array of device pointers storing each vector x_i .
- **incx** – [in] [rocblas_int] specifies the increment for the elements of each x_i .
- **y** – [in] device array of device pointers storing each vector y_i .
- **incy** – [in] [rocblas_int] specifies the increment for the elements of each y_i .
- **A** – [inout] device array of device pointers storing each matrix A_i .
- **lda** – [in] [rocblas_int] specifies the leading dimension of each A_i .
- **batch_count** – [in] [rocblas_int] number of instances in the batch.

```
rocblas_status rocblas_ssy2_strided_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_int n, const
float *alpha, const float *x, rocblas_int incx, rocblas_stride
stridex, const float *y, rocblas_int incy, rocblas_stride stridey,
float *A, rocblas_int lda, rocblas_stride strideA, rocblas_int
batch_count)
```

```
rocblas_status rocblas_dsyr2_strided_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_int n, const
double *alpha, const double *x, rocblas_int incx,
rocblas_stride stridex, const double *y, rocblas_int incy,
rocblas_stride stridey, double *A, rocblas_int lda,
rocblas_stride strideA, rocblas_int batch_count)
```

```
rocblas_status rocblas_csy2_strided_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_int n, const
rocblas_float_complex *alpha, const rocblas_float_complex
*x, rocblas_int incx, rocblas_stride stridex, const
rocblas_float_complex *y, rocblas_int incy, rocblas_stride
stridey, rocblas_float_complex *A, rocblas_int lda,
rocblas_stride strideA, rocblas_int batch_count)
```

```
rocblas_status rocblas_zsy2_strided_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_int n, const
rocblas_double_complex *alpha, const
rocblas_double_complex *x, rocblas_int incx, rocblas_stride
stridex, const rocblas_double_complex *y, rocblas_int incy,
rocblas_stride stridey, rocblas_double_complex *A,
rocblas_int lda, rocblas_stride strideA, rocblas_int
batch_count)
```

BLAS Level 2 API

syr2_strided_batched the matrix-vector operations:

```
A[i] := A[i] + alpha*x[i]*y[i]**T + alpha*y[i]*x[i]**T
where alpha is a scalar, x[i] and y[i] are vectors, and A[i] is a
n by n symmetric matrices, for i = 1 , ... , batch_count
```

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill] specifies whether the upper ‘rocblas_fill_upper’ or lower ‘rocblas_fill_lower’
 - if rocblas_fill_upper, the lower part of A is not referenced
 - if rocblas_fill_lower, the upper part of A is not referenced
- **n** – [in] [rocblas_int] the number of rows and columns of each matrix A.
- **alpha** – [in] device pointer or host pointer to scalar alpha.
- **x** – [in] device pointer to the first vector x₁.
- **incx** – [in] [rocblas_int] specifies the increment for the elements of each x_i.
- **stridex** – [in] [rocblas_stride] specifies the pointer increment between vectors (x_i) and (x_{i+1}).
- **y** – [in] device pointer to the first vector y₁.
- **incy** – [in] [rocblas_int] specifies the increment for the elements of each y_i.

- **stridey** – [in] [rocblas_stride] specifies the pointer increment between vectors (y_i) and (y_{i+1}).
- **A** – [inout] device pointer to the first matrix A_1 .
- **lda** – [in] [rocblas_int] specifies the leading dimension of each A_i .
- **strideA** – [in] [rocblas_stride] stride from the start of one matrix (A_i) and the next one (A_{i+1}).
- **batch_count** – [in] [rocblas_int] number of instances in the batch.

5.5.11 rocblas_Xtbmv + batched, strided_batched

```
rocblas_status rocblas_stbmv(rocblas_handle handle, rocblas_fill uplo, rocblas_operation trans,
                             rocblas_diagonal diag, rocblas_int m, rocblas_int k, const float *A, rocblas_int
                             lda, float *x, rocblas_int incx)
```

```
rocblas_status rocblas_dtbmv(rocblas_handle handle, rocblas_fill uplo, rocblas_operation trans,
                             rocblas_diagonal diag, rocblas_int m, rocblas_int k, const double *A, rocblas_int
                             lda, double *x, rocblas_int incx)
```

```
rocblas_status rocblas_ctbmv(rocblas_handle handle, rocblas_fill uplo, rocblas_operation trans,
                             rocblas_diagonal diag, rocblas_int m, rocblas_int k, const rocblas_float_complex
                             *A, rocblas_int lda, rocblas_float_complex *x, rocblas_int incx)
```

```
rocblas_status rocblas_ztbmv(rocblas_handle handle, rocblas_fill uplo, rocblas_operation trans,
                             rocblas_diagonal diag, rocblas_int m, rocblas_int k, const
                             rocblas_double_complex *A, rocblas_int lda, rocblas_double_complex *x,
                             rocblas_int incx)
```

BLAS Level 2 API

tbmv performs one of the matrix-vector operations:

```
x := A*x           or
x := A**T*x        or
x := A**H*x,
x is a vectors and A is a banded m by m matrix (see description below).
```

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill]
 - rocblas_fill_upper: A is an upper banded triangular matrix.
 - rocblas_fill_lower: A is a lower banded triangular matrix.
- **trans** – [in] [rocblas_operation] indicates whether matrix A is transposed (conjugated) or not.
- **diag** – [in] [rocblas_diagonal]
 - rocblas_diagonal_unit: The main diagonal of A is assumed to consist of only 1's and is not referenced.
 - rocblas_diagonal_non_unit: No assumptions are made of A's main diagonal.
- **m** – [in] [rocblas_int] the number of rows and columns of the matrix represented by A.

- **k** – [in] [rocblas_int]

```

    if uplo == rocblas_fill_upper, k specifies the number of super-
    ↪diagonals
      of the matrix A.

    if uplo == rocblas_fill_lower, k specifies the number of sub-
    ↪diagonals
      of the matrix A.
      k must satisfy  $k > 0 \ \&\& \ k < \text{lda}$ .

```

- **A** – [in] device pointer storing banded triangular matrix A.

```

    if uplo == rocblas_fill_upper:
        The matrix represented is an upper banded triangular matrix
        with the main diagonal and k super-diagonals, everything
        else can be assumed to be 0.
        The matrix is compacted so that the main diagonal resides on_
    ↪the k'th
        row, the first super diagonal resides on the RHS of the k-1
    ↪'th row, etc,
        with the k'th diagonal on the RHS of the 0'th row.
        Ex: (rocblas_fill_upper; m = 5; k = 2)
            1 6 9 0 0      0 0 9 8 7
            0 2 7 8 0      0 6 7 8 9
            0 0 3 8 7      1 2 3 4 5
            0 0 0 4 9      0 0 0 0 0
            0 0 0 0 5      0 0 0 0 0

    if uplo == rocblas_fill_lower:
        The matrix represented is a lower banded triangular matrix
        with the main diagonal and k sub-diagonals, everything else_
    ↪can be
        assumed to be 0.
        The matrix is compacted so that the main diagonal resides on_
    ↪the 0'th row,
        working up to the k'th diagonal residing on the LHS of the k
    ↪'th row.
        Ex: (rocblas_fill_lower; m = 5; k = 2)
            1 0 0 0 0      1 2 3 4 5
            6 2 0 0 0      6 7 8 9 0
            9 7 3 0 0      9 8 7 0 0
            0 8 8 4 0      0 0 0 0 0
            0 0 7 9 5      0 0 0 0 0

```

- **lda** – [in] [rocblas_int] specifies the leading dimension of A. lda must satisfy $\text{lda} > k$.
- **x** – [inout] device pointer storing vector x.
- **incx** – [in] [rocblas_int] specifies the increment for the elements of x.

rocblas_status **rocblas_stbmv_batched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* trans, *rocblas_diagonal* diag, *rocblas_int* m, *rocblas_int* k, const float *const A[], *rocblas_int* lda, float *const x[], *rocblas_int* incx, *rocblas_int* batch_count)

```
rocblas_status rocblas_dtbmv_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_operation trans,
                                     rocblas_diagonal diag, rocblas_int m, rocblas_int k, const double *const
                                     A[], rocblas_int lda, double *const x[], rocblas_int incx, rocblas_int
                                     batch_count)
```

```
rocblas_status rocblas_ctbmv_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_operation trans,
                                     rocblas_diagonal diag, rocblas_int m, rocblas_int k, const
                                     rocblas_float_complex *const A[], rocblas_int lda,
                                     rocblas_float_complex *const x[], rocblas_int incx, rocblas_int
                                     batch_count)
```

```
rocblas_status rocblas_ztbmv_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_operation trans,
                                     rocblas_diagonal diag, rocblas_int m, rocblas_int k, const
                                     rocblas_double_complex *const A[], rocblas_int lda,
                                     rocblas_double_complex *const x[], rocblas_int incx, rocblas_int
                                     batch_count)
```

BLAS Level 2 API

tbmv_batched performs one of the matrix-vector operations:

```
x_i := A_i*x_i      or
x_i := A_i**T*x_i   or
x_i := A_i**H*x_i,
where (A_i, x_i) is the i-th instance of the batch.
x_i is a vector and A_i is an m by m matrix, for i = 1, ..., batch_count.
```

Parameters

- **handle** – [in] [*rocblas_handle*] handle to the rocblas library context queue.
- **uplo** – [in] [*rocblas_fill*]
 - *rocblas_fill_upper*: each A_i is an upper banded triangular matrix.
 - *rocblas_fill_lower*: each A_i is a lower banded triangular matrix.
- **trans** – [in] [*rocblas_operation*] indicates whether each matrix A_i is tranposed (conjugated) or not.
- **diag** – [in] [*rocblas_diagonal*]
 - *rocblas_diagonal_unit*: The main diagonal of each A_i is assumed to consist of only 1's and is not referenced.
 - *rocblas_diagonal_non_unit*: No assumptions are made of each A_i's main diagonal.
- **m** – [in] [*rocblas_int*] the number of rows and columns of the matrix represented by each A_i.
- **k** – [in] [*rocblas_int*]

```
if uplo == rocblas_fill_upper, k specifies the number of super-
↪diagonals
  of each matrix A_i.

if uplo == rocblas_fill_lower, k specifies the number of sub-
↪diagonals
  of each matrix A_i.
  k must satisfy k > 0 && k < lda.
```

- **A** – [in] device array of device pointers storing each banded triangular matrix A_i .

```

    if uplo == rocblas_fill_upper:
        The matrix represented is an upper banded triangular matrix
        with the main diagonal and k super-diagonals, everything
        else can be assumed to be 0.
        The matrix is compacted so that the main diagonal resides on
        ↪ the k'th
        row, the first super diagonal resides on the RHS of the k-1
        ↪ 'th row, etc,
        with the k'th diagonal on the RHS of the 0'th row.
        Ex: (rocblas_fill_upper; m = 5; k = 2)
            1 6 9 0 0          0 0 9 8 7
            0 2 7 8 0          0 6 7 8 9
            0 0 3 8 7  ---->  1 2 3 4 5
            0 0 0 4 9          0 0 0 0 0
            0 0 0 0 5          0 0 0 0 0

    if uplo == rocblas_fill_lower:
        The matrix represented is a lower banded triangular matrix
        with the main diagonal and k sub-diagonals, everything else
        ↪ can be
        assumed to be 0.
        The matrix is compacted so that the main diagonal resides on
        ↪ the 0'th row,
        working up to the k'th diagonal residing on the LHS of the k
        ↪ 'th row.
        Ex: (rocblas_fill_lower; m = 5; k = 2)
            1 0 0 0 0          1 2 3 4 5
            6 2 0 0 0          6 7 8 9 0
            9 7 3 0 0  ---->  9 8 7 0 0
            0 8 8 4 0          0 0 0 0 0
            0 0 7 9 5          0 0 0 0 0

```

- **lda** – [in] [rocblas_int] specifies the leading dimension of each A_i . lda must satisfy $lda > k$.
- **x** – [inout] device array of device pointer storing each vector x_i .
- **incx** – [in] [rocblas_int] specifies the increment for the elements of each x_i .
- **batch_count** – [in] [rocblas_int] number of instances in the batch.

rocblas_status rocblas_stbmv_strided_batched(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* trans, *rocblas_diagonal* diag, *rocblas_int* m, *rocblas_int* k, const float *A, *rocblas_int* lda, *rocblas_stride* stride_A, float *x, *rocblas_int* incx, *rocblas_stride* stride_x, *rocblas_int* batch_count)

rocblas_status rocblas_dtbmv_strided_batched(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* trans, *rocblas_diagonal* diag, *rocblas_int* m, *rocblas_int* k, const double *A, *rocblas_int* lda, *rocblas_stride* stride_A, double *x, *rocblas_int* incx, *rocblas_stride* stride_x, *rocblas_int* batch_count)


```
rocblas_status rocblas_ctbmv_strided_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_operation
trans, rocblas_diagonal diag, rocblas_int m, rocblas_int k,
const rocblas_float_complex *A, rocblas_int lda,
rocblas_stride stride_A, rocblas_float_complex *x,
rocblas_int incx, rocblas_stride stride_x, rocblas_int
batch_count)
```

```
rocblas_status rocblas_ztbmv_strided_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_operation
trans, rocblas_diagonal diag, rocblas_int m, rocblas_int k,
const rocblas_double_complex *A, rocblas_int lda,
rocblas_stride stride_A, rocblas_double_complex *x,
rocblas_int incx, rocblas_stride stride_x, rocblas_int
batch_count)
```

BLAS Level 2 API

tbmv_strided_batched performs one of the matrix-vector operations:

```
x_i := A_i*x_i      or
x_i := A_i**T*x_i   or
x_i := A_i**H*x_i,
where (A_i, x_i) is the i-th instance of the batch.
x_i is a vector and A_i is an m by m matrix, for i = 1, ..., batch_count.
```

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill]
 - rocblas_fill_upper: each A_i is an upper banded triangular matrix.
 - rocblas_fill_lower: each A_i is a lower banded triangular matrix.
- **trans** – [in] [rocblas_operation] indicates whether each matrix A_i is tranposed (conjugated) or not.
- **diag** – [in] [rocblas_diagonal]
 - rocblas_diagonal_unit: The main diagonal of each A_i is assumed to consist of only 1's and is not referenced.
 - rocblas_diagonal_non_unit: No assumptions are made of each A_i's main diagonal.
- **m** – [in] [rocblas_int] the number of rows and columns of the matrix represented by each A_i.
- **k** – [in] [rocblas_int]

```
if uplo == rocblas_fill_upper, k specifies the number of super-
diagonals
of each matrix A_i.

if uplo == rocblas_fill_lower, k specifies the number of sub-
diagonals
of each matrix A_i.
k must satisfy k > 0 && k < lda.
```

- **A** – [in] device array to the first matrix A_i of the batch. Stores each banded triangular matrix A_i.

```

    if uplo == rocblas_fill_upper:
        The matrix represented is an upper banded triangular matrix
        with the main diagonal and k super-diagonals, everything
        else can be assumed to be 0.
        The matrix is compacted so that the main diagonal resides on
        ↪ the k'th
        row, the first super diagonal resides on the RHS of the k-1
        ↪ 'th row, etc,
        with the k'th diagonal on the RHS of the 0'th row.
        Ex: (rocblas_fill_upper; m = 5; k = 2)
            1 6 9 0 0      0 0 9 8 7
            0 2 7 8 0      0 6 7 8 9
            0 0 3 8 7      ----> 1 2 3 4 5
            0 0 0 4 9      0 0 0 0 0
            0 0 0 0 5      0 0 0 0 0

    if uplo == rocblas_fill_lower:
        The matrix represented is a lower banded triangular matrix
        with the main diagonal and k sub-diagonals, everything else
        ↪ can be
        assumed to be 0.
        The matrix is compacted so that the main diagonal resides on
        ↪ the 0'th row,
        working up to the k'th diagonal residing on the LHS of the k
        ↪ 'th row.
        Ex: (rocblas_fill_lower; m = 5; k = 2)
            1 0 0 0 0      1 2 3 4 5
            6 2 0 0 0      6 7 8 9 0
            9 7 3 0 0      ----> 9 8 7 0 0
            0 8 8 4 0      0 0 0 0 0
            0 0 7 9 5      0 0 0 0 0

```

- **lda** – [in] [rocblas_int] specifies the leading dimension of each A_i. lda must satisfy lda > k.
- **stride_A** – [in] [rocblas_stride] stride from the start of one A_i matrix to the next A_i(i + 1).
- **x** – [inout] device array to the first vector x_i of the batch.
- **incx** – [in] [rocblas_int] specifies the increment for the elements of each x_i.
- **stride_x** – [in] [rocblas_stride] stride from the start of one x_i matrix to the next x_i(i + 1).
- **batch_count** – [in] [rocblas_int] number of instances in the batch.

5.5.12 rocblas_Xtbsv + batched, strided_batched

```
rocblas_status rocblas_stbsv(rocblas_handle handle, rocblas_fill uplo, rocblas_operation transA,
                             rocblas_diagonal diag, rocblas_int n, rocblas_int k, const float *A, rocblas_int lda,
                             float *x, rocblas_int incx)
```

```
rocblas_status rocblas_dtbsv(rocblas_handle handle, rocblas_fill uplo, rocblas_operation transA,
                             rocblas_diagonal diag, rocblas_int n, rocblas_int k, const double *A, rocblas_int
                             lda, double *x, rocblas_int incx)
```

```
rocblas_status rocblas_ctbsv(rocblas_handle handle, rocblas_fill uplo, rocblas_operation transA,
                             rocblas_diagonal diag, rocblas_int n, rocblas_int k, const rocblas_float_complex
                             *A, rocblas_int lda, rocblas_float_complex *x, rocblas_int incx)
```

```
rocblas_status rocblas_ztbsv(rocblas_handle handle, rocblas_fill uplo, rocblas_operation transA,
                             rocblas_diagonal diag, rocblas_int n, rocblas_int k, const rocblas_double_complex
                             *A, rocblas_int lda, rocblas_double_complex *x, rocblas_int incx)
```

BLAS Level 2 API

tbsv solves:

```
A*x = b or
A**T*x = b or
A**H*x = b
where x and b are vectors and A is a banded triangular matrix.
```

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill]
 - rocblas_fill_upper: A is an upper triangular matrix.
 - rocblas_fill_lower: A is a lower triangular matrix.
- **transA** – [in] [rocblas_operation]
 - rocblas_operation_none: Solves $A*x = b$
 - rocblas_operation_transpose: Solves $A^{**T}*x = b$
 - rocblas_operation_conjugate_transpose: Solves $A^{**H}*x = b$
- **diag** – [in] [rocblas_diagonal]
 - rocblas_diagonal_unit: A is assumed to be unit triangular (i.e. the diagonal elements of A are not used in computations).
 - rocblas_diagonal_non_unit: A is not assumed to be unit triangular.
- **n** – [in] [rocblas_int] n specifies the number of rows of b. $n \geq 0$.
- **k** – [in] [rocblas_int]

```
if(uplo == rocblas_fill_upper)
    k specifies the number of super-diagonals of A.
if(uplo == rocblas_fill_lower)
    k specifies the number of sub-diagonals of A.
k >= 0.
```

- **A** – [in] device pointer storing the matrix A in banded format.
- **lda** – [in] [rocblas_int] specifies the leading dimension of A. $\text{lda} \geq (k + 1)$.
- **x** – [inout] device pointer storing input vector b. Overwritten by the output vector x.
- **incx** – [in] [rocblas_int] specifies the increment for the elements of x.

rocblas_status **rocblas_stbsv_batched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_diagonal* diag, *rocblas_int* n, *rocblas_int* k, const float *const A[], *rocblas_int* lda, float *const x[], *rocblas_int* incx, *rocblas_int* batch_count)

rocblas_status **rocblas_dtbsv_batched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_diagonal* diag, *rocblas_int* n, *rocblas_int* k, const double *const A[], *rocblas_int* lda, double *const x[], *rocblas_int* incx, *rocblas_int* batch_count)

rocblas_status **rocblas_ctbsv_batched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_diagonal* diag, *rocblas_int* n, *rocblas_int* k, const *rocblas_float_complex* *const A[], *rocblas_int* lda, *rocblas_float_complex* *const x[], *rocblas_int* incx, *rocblas_int* batch_count)

rocblas_status **rocblas_ztbsv_batched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_diagonal* diag, *rocblas_int* n, *rocblas_int* k, const *rocblas_double_complex* *const A[], *rocblas_int* lda, *rocblas_double_complex* *const x[], *rocblas_int* incx, *rocblas_int* batch_count)

BLAS Level 2 API

tbsv_batched solves:

```
A_i*x_i = b_i or
A_i**T*x_i = b_i or
A_i**H*x_i = b_i
where x_i and b_i are vectors and A_i is a banded triangular matrix,
for i = [1, batch_count].
```

The input vectors b_i are overwritten by the output vectors x_i.

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill]
 - rocblas_fill_upper: A_i is an upper triangular matrix.
 - rocblas_fill_lower: A_i is a lower triangular matrix.
- **transA** – [in] [rocblas_operation]
 - rocblas_operation_none: Solves $A_i x_i = b_i$
 - rocblas_operation_transpose: Solves $A_i^T x_i = b_i$
 - rocblas_operation_conjugate_transpose: Solves $A_i^H x_i = b_i$
- **diag** – [in] [rocblas_diagonal]

- `rocblas_diagonal_unit`: each A_i is assumed to be unit triangular (i.e. the diagonal elements of each A_i are not used in computations).
- `rocblas_diagonal_non_unit`: each A_i is not assumed to be unit triangular.
- **n** – [in] [rocblas_int] n specifies the number of rows of each b_i . $n \geq 0$.
- **k** – [in] [rocblas_int]

```

if(uplo == rocblas_fill_upper)
    k specifies the number of super-diagonals of each  $A_i$ .
if(uplo == rocblas_fill_lower)
    k specifies the number of sub-diagonals of each  $A_i$ .
k >= 0.

```

- **A** – [in] device vector of device pointers storing each matrix A_i in banded format.
- **lda** – [in] [rocblas_int] specifies the leading dimension of each A_i . $lda \geq (k + 1)$.
- **x** – [inout] device vector of device pointers storing each input vector b_i . Overwritten by each output vector x_i .
- **incx** – [in] [rocblas_int] specifies the increment for the elements of each x_i .
- **batch_count** – [in] [rocblas_int] number of instances in the batch.

```

rocblas_status rocblas_stbsv_strided_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_operation
transA, rocblas_diagonal diag, rocblas_int n, rocblas_int k,
const float *A, rocblas_int lda, rocblas_stride stride_A, float
*x, rocblas_int incx, rocblas_stride stride_x, rocblas_int
batch_count)

```

```

rocblas_status rocblas_dtbsv_strided_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_operation
transA, rocblas_diagonal diag, rocblas_int n, rocblas_int k,
const double *A, rocblas_int lda, rocblas_stride stride_A,
double *x, rocblas_int incx, rocblas_stride stride_x,
rocblas_int batch_count)

```

```

rocblas_status rocblas_ctbsv_strided_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_operation
transA, rocblas_diagonal diag, rocblas_int n, rocblas_int k,
const rocblas_float_complex *A, rocblas_int lda,
rocblas_stride stride_A, rocblas_float_complex *x,
rocblas_int incx, rocblas_stride stride_x, rocblas_int
batch_count)

```

```

rocblas_status rocblas_ztbsv_strided_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_operation
transA, rocblas_diagonal diag, rocblas_int n, rocblas_int k,
const rocblas_double_complex *A, rocblas_int lda,
rocblas_stride stride_A, rocblas_double_complex *x,
rocblas_int incx, rocblas_stride stride_x, rocblas_int
batch_count)

```

BLAS Level 2 API

`tbsv_strided_batched` solves:

```

 $A_i * x_i = b_i$  or
 $A_i^T * x_i = b_i$  or
 $A_i^H * x_i = b_i$ 

```

(continues on next page)

(continued from previous page)

where x_i and b_i are vectors and A_i is a banded triangular matrix,
 for $i = [1, \text{batch_count}]$.

The input vectors b_i are overwritten by the output vectors x_i .

Parameters

- **handle** – [in] [rocbblas_handle] handle to the rocbblas library context queue.
- **uplo** – [in] [rocbblas_fill]
 - rocbblas_fill_upper: A_i is an upper triangular matrix.
 - rocbblas_fill_lower: A_i is a lower triangular matrix.
- **transA** – [in] [rocbblas_operation]
 - rocbblas_operation_none: Solves $A_i * x_i = b_i$
 - rocbblas_operation_transpose: Solves $A_i^T * x_i = b_i$
 - rocbblas_operation_conjugate_transpose: Solves $A_i^H * x_i = b_i$
- **diag** – [in] [rocbblas_diagonal]
 - rocbblas_diagonal_unit: each A_i is assumed to be unit triangular (i.e. the diagonal elements of each A_i are not used in computations).
 - rocbblas_diagonal_non_unit: each A_i is not assumed to be unit triangular.
- **n** – [in] [rocbblas_int] n specifies the number of rows of each b_i . $n \geq 0$.
- **k** – [in] [rocbblas_int]

```

if(uplo == rocbblas_fill_upper)
    k specifies the number of super-diagonals of each  $A_i$ .
if(uplo == rocbblas_fill_lower)
    k specifies the number of sub-diagonals of each  $A_i$ .
k >= 0.
  
```

- **A** – [in] device pointer pointing to the first banded matrix A_1 .
- **lda** – [in] [rocbblas_int] specifies the leading dimension of each A_i . $lda \geq (k + 1)$.
- **stride_A** – [in] [rocbblas_stride] specifies the distance between the start of one matrix (A_i) and the next (A_{i+1}).
- **x** – [inout] device pointer pointing to the first input vector b_1 . Overwritten by output vectors x .
- **incx** – [in] [rocbblas_int] specifies the increment for the elements of each x_i .
- **stride_x** – [in] [rocbblas_stride] specifies the distance between the start of one vector (x_i) and the next (x_{i+1}).
- **batch_count** – [in] [rocbblas_int] number of instances in the batch.

5.5.13 rocblas_Xtpmv + batched, strided_batched

rocblas_status **rocblas_stpmv**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_diagonal* diag, *rocblas_int* m, const float *A, float *x, *rocblas_int* incx)

rocblas_status **rocblas_dtpmv**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_diagonal* diag, *rocblas_int* m, const double *A, double *x, *rocblas_int* incx)

rocblas_status **rocblas_ctpmv**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_diagonal* diag, *rocblas_int* m, const *rocblas_float_complex* *A, *rocblas_float_complex* *x, *rocblas_int* incx)

rocblas_status **rocblas_ztpmv**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_diagonal* diag, *rocblas_int* m, const *rocblas_double_complex* *A, *rocblas_double_complex* *x, *rocblas_int* incx)

BLAS Level 2 API

tpmv 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 the pack form.
The vector x is overwritten.
```

Parameters

- **handle** – [in] [*rocblas_handle*] handle to the rocblas library context queue.
- **uplo** – [in] [*rocblas_fill*]
 - *rocblas_fill_upper*: A is an upper triangular matrix.
 - *rocblas_fill_lower*: A is a lower triangular matrix.
- **transA** – [in] [*rocblas_operation*]
- **diag** – [in] [*rocblas_diagonal*]
 - *rocblas_diagonal_unit*: A is assumed to be unit triangular.
 - *rocblas_diagonal_non_unit*: A is not assumed to be unit triangular.
- **m** – [in] [*rocblas_int*] m specifies the number of rows of A. $m \geq 0$.
- **A** – [in] device pointer storing matrix A, of dimension at least $(m + 1) / 2$.
 - Before entry with *uplo* = *rocblas_fill_upper*, the array A must contain the upper triangular matrix packed sequentially, column by column, so that A[0] contains $a_{\{0,0\}}$, A[1] and A[2] contain $a_{\{0,1\}}$ and $a_{\{1,1\}}$, respectively, and so on.
 - Before entry with *uplo* = *rocblas_fill_lower*, the array A must contain the lower triangular matrix packed sequentially, column by column, so that A[0] contains $a_{\{0,0\}}$, A[1] and A[2] contain $a_{\{1,0\}}$ and $a_{\{2,0\}}$, respectively, and so on.

Note that when *DIAG* = *rocblas_diagonal_unit*, the diagonal elements of A are not referenced, but are assumed to be unity.
- **x** – [in] device pointer storing vector x.
- **incx** – [in] [*rocblas_int*] specifies the increment for the elements of x. incx must not be zero.

```
rocblas_status rocblas_stpmv_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_operation transA,
                                     rocblas_diagonal diag, rocblas_int m, const float *const *A, float *const
                                     *x, rocblas_int incx, rocblas_int batch_count)
```

```
rocblas_status rocblas_dtpmv_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_operation transA,
                                     rocblas_diagonal diag, rocblas_int m, const double *const *A, double
                                     *const *x, rocblas_int incx, rocblas_int batch_count)
```

```
rocblas_status rocblas_ctpmv_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_operation transA,
                                     rocblas_diagonal diag, rocblas_int m, const rocblas_float_complex
                                     *const *A, rocblas_float_complex *const *x, rocblas_int incx,
                                     rocblas_int batch_count)
```

```
rocblas_status rocblas_ztpmv_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_operation transA,
                                     rocblas_diagonal diag, rocblas_int m, const rocblas_double_complex
                                     *const *A, rocblas_double_complex *const *x, rocblas_int incx,
                                     rocblas_int batch_count)
```

BLAS Level 2 API

tpmv_batched performs one of the matrix-vector operations:

```
x_i = A_i*x_i or
x_i = A**T*x_i, 0 < i < batch_count
where x_i is an n element vector and A_i is an n by n (unit, or non-unit, upper or
↳ lower triangular matrix)
The vectors x_i are overwritten.
```

Parameters

- **handle** – [in] [*rocblas_handle*] handle to the rocblas library context queue.
- **uplo** – [in] [*rocblas_fill*]
 - *rocblas_fill_upper*: A_i is an upper triangular matrix.
 - *rocblas_fill_lower*: A_i is a lower triangular matrix.
- **transA** – [in] [*rocblas_operation*]
- **diag** – [in] [*rocblas_diagonal*]
 - *rocblas_diagonal_unit*: A_i is assumed to be unit triangular.
 - *rocblas_diagonal_non_unit*: A_i is not assumed to be unit triangular.
- **m** – [in] [*rocblas_int*] m specifies the number of rows of matrices A_i. m >= 0.
- **A** – [in] device pointer storing pointer of matrices A_i, of dimension (lda, m).
- **x** – [in] device pointer storing vectors x_i.
- **incx** – [in] [*rocblas_int*] specifies the increment for the elements of vectors x_i.
- **batch_count** – [in] [*rocblas_int*] The number of batched matrices/vectors.

```
rocblas_status rocblas_stpmv_strided_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_operation
transA, rocblas_diagonal diag, rocblas_int m, const float *A,
rocblas_stride stride_A, float *x, rocblas_int incx,
rocblas_stride stride_x, rocblas_int batch_count)
```



```

rocblas_status rocblas_dtpmv_strided_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_operation
transA, rocblas_diagonal diag, rocblas_int m, const double
*A, rocblas_stride stride_A, double *x, rocblas_int incx,
rocblas_stride stride_x, rocblas_int batch_count)

rocblas_status rocblas_ctpmv_strided_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_operation
transA, rocblas_diagonal diag, rocblas_int m, const
rocblas_float_complex *A, rocblas_stride stride_A,
rocblas_float_complex *x, rocblas_int incx, rocblas_stride
stride_x, rocblas_int batch_count)

rocblas_status rocblas_ztpmv_strided_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_operation
transA, rocblas_diagonal diag, rocblas_int m, const
rocblas_double_complex *A, rocblas_stride stride_A,
rocblas_double_complex *x, rocblas_int incx, rocblas_stride
stride_x, rocblas_int batch_count)

```

BLAS Level 2 API

tpmv_strided_batched performs one of the matrix-vector operations:

```

x_i = A_i*x_i or
x_i = A**T*x_i, 0 < i < batch_count
where x_i is an n element vector and A_i is an n by n (unit, or non-unit, upper or
lower triangular matrix)
with strides specifying how to retrieve $x_i$ (resp. $A_i$) from $x_{i-1}$ (resp.
$A_i$).
The vectors x_i are overwritten.

```

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill]
 - rocblas_fill_upper: A_i is an upper triangular matrix.
 - rocblas_fill_lower: A_i is a lower triangular matrix.
- **transA** – [in] [rocblas_operation]
- **diag** – [in] [rocblas_diagonal]
 - rocblas_diagonal_unit: A_i is assumed to be unit triangular.
 - rocblas_diagonal_non_unit: A_i is not assumed to be unit triangular.
- **m** – [in] [rocblas_int] m specifies the number of rows of matrices A_i. m >= 0.
- **A** – [in] device pointer of the matrix A_0, of dimension (lda, m)
- **stride_A** – [in] [rocblas_stride] stride from the start of one A_i matrix to the next A_{i + 1}.
- **x** – [in] device pointer storing the vector x_0.
- **incx** – [in] [rocblas_int] specifies the increment for the elements of one vector x.
- **stride_x** – [in] [rocblas_stride] stride from the start of one x_i vector to the next x_{i + 1}.
- **batch_count** – [in] [rocblas_int] The number of batched matrices/vectors.

5.5.14 rocblas_Xtpsv + batched, strided_batched

rocblas_status **rocblas_stpsv**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_diagonal* diag, *rocblas_int* n, const float *AP, float *x, *rocblas_int* incx)

rocblas_status **rocblas_dtpsv**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_diagonal* diag, *rocblas_int* n, const double *AP, double *x, *rocblas_int* incx)

rocblas_status **rocblas_ctpsv**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_diagonal* diag, *rocblas_int* n, const *rocblas_float_complex* *AP, *rocblas_float_complex* *x, *rocblas_int* incx)

rocblas_status **rocblas_ztpsv**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_diagonal* diag, *rocblas_int* n, const *rocblas_double_complex* *AP, *rocblas_double_complex* *x, *rocblas_int* incx)

BLAS Level 2 API

tpsv solves:

$A * x = b$ **or**
 $A^{**T} * x = b$ **or**
 $A^{**H} * x = b$
 where x **and** b are vectors **and** A **is** a triangular matrix stored **in** the packed **format**.

The input vector b is overwritten by the output vector x .

Parameters

- **handle** – [in] [*rocblas_handle*] handle to the rocblas library context queue.
- **uplo** – [in] [*rocblas_fill*]
 - *rocblas_fill_upper*: A is an upper triangular matrix.
 - *rocblas_fill_lower*: A is a lower triangular matrix.
- **transA** – [in] [*rocblas_operation*]
 - *rocblas_operation_none*: Solves $A * x = b$
 - *rocblas_operation_transpose*: Solves $A^{**T} * x = b$
 - *rocblas_operation_conjugate_transpose*: Solves $A^{**H} * x = b$
- **diag** – [in] [*rocblas_diagonal*]
 - *rocblas_diagonal_unit*: A is assumed to be unit triangular (i.e. the diagonal elements of A are not used in computations).
 - *rocblas_diagonal_non_unit*: A is not assumed to be unit triangular.
- **n** – [in] [*rocblas_int*] n specifies the number of rows of b . $n \geq 0$.
- **AP** – [in] device pointer storing the packed version of matrix A , of dimension $\geq (n * (n + 1) / 2)$.
- **x** – [inout] device pointer storing vector b on input, overwritten by x on output.
- **incx** – [in] [*rocblas_int*] specifies the increment for the elements of x .

```
rocblas_status rocblas_stpsv_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_operation transA,
                                     rocblas_diagonal diag, rocblas_int n, const float *const AP[], float
                                     *const x[], rocblas_int incx, rocblas_int batch_count)
```

```
rocblas_status rocblas_dtpsv_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_operation transA,
                                     rocblas_diagonal diag, rocblas_int n, const double *const AP[], double
                                     *const x[], rocblas_int incx, rocblas_int batch_count)
```

```
rocblas_status rocblas_ctpsv_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_operation transA,
                                     rocblas_diagonal diag, rocblas_int n, const rocblas_float_complex
                                     *const AP[], rocblas_float_complex *const x[], rocblas_int incx,
                                     rocblas_int batch_count)
```

```
rocblas_status rocblas_ztpsv_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_operation transA,
                                     rocblas_diagonal diag, rocblas_int n, const rocblas_double_complex
                                     *const AP[], rocblas_double_complex *const x[], rocblas_int incx,
                                     rocblas_int batch_count)
```

BLAS Level 2 API

tpsv_batched solves:

```
A_i*x_i = b_i or
A_i**T*x_i = b_i or
A_i**H*x_i = b_i
where x_i and b_i are vectors and A_i is a triangular matrix stored in the packed_
↪format,
for i in [1, batch_count].
```

The input vectors b_i are overwritten by the output vectors x_i .

Parameters

- **handle** – [in] [*rocblas_handle*] handle to the rocblas library context queue.
- **uplo** – [in] [*rocblas_fill*]
 - *rocblas_fill_upper*: each A_i is an upper triangular matrix.
 - *rocblas_fill_lower*: each A_i is a lower triangular matrix.
- **transA** – [in] [*rocblas_operation*]
 - *rocblas_operation_none*: Solves $A*x = b$
 - *rocblas_operation_transpose*: Solves $A^{**T}*x = b$
 - *rocblas_operation_conjugate_transpose*: Solves $A^{**H}*x = b$
- **diag** – [in] [*rocblas_diagonal*]
 - *rocblas_diagonal_unit*: Each A_i is assumed to be unit triangular (i.e. the diagonal elements of each A_i are not used in computations).
 - *rocblas_diagonal_non_unit*: each A_i is not assumed to be unit triangular.
- **n** – [in] [*rocblas_int*] n specifies the number of rows of each b_i . $n \geq 0$.
- **AP** – [in] device array of device pointers storing the packed versions of each matrix A_i , of dimension $\geq (n * (n + 1) / 2)$.
- **x** – [inout] device array of device pointers storing each input vector b_i , overwritten by x_i on output.

- **incx** – [in] [rocblas_int] specifies the increment for the elements of each x_i .
- **batch_count** – [in] [rocblas_int] specifies the number of instances in the batch.

rocblas_status **rocblas_stpsv_strided_batched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_diagonal* diag, *rocblas_int* n, const float *AP, *rocblas_stride* stride_A, float *x, *rocblas_int* incx, *rocblas_stride* stride_x, *rocblas_int* batch_count)

rocblas_status **rocblas_dtpsv_strided_batched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_diagonal* diag, *rocblas_int* n, const double *AP, *rocblas_stride* stride_A, double *x, *rocblas_int* incx, *rocblas_stride* stride_x, *rocblas_int* batch_count)

rocblas_status **rocblas_ctpsv_strided_batched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_diagonal* diag, *rocblas_int* n, const *rocblas_float_complex* *AP, *rocblas_stride* stride_A, *rocblas_float_complex* *x, *rocblas_int* incx, *rocblas_stride* stride_x, *rocblas_int* batch_count)

rocblas_status **rocblas_ztpsv_strided_batched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_diagonal* diag, *rocblas_int* n, const *rocblas_double_complex* *AP, *rocblas_stride* stride_A, *rocblas_double_complex* *x, *rocblas_int* incx, *rocblas_stride* stride_x, *rocblas_int* batch_count)

BLAS Level 2 API

tpsv_strided_batched solves:

```
A_i*x_i = b_i or
A_i**T*x_i = b_i or
A_i**H*x_i = b_i
where x_i and b_i are vectors and A_i is a triangular matrix stored in the packed_
format,
for i in [1, batch_count].
```

The input vectors b_i are overwritten by the output vectors x_i .

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill]
 - rocblas_fill_upper: each A_i is an upper triangular matrix.
 - rocblas_fill_lower: each A_i is a lower triangular matrix.
- **transA** – [in] [rocblas_operation]
 - rocblas_operation_none: Solves $A*x = b$
 - rocblas_operation_transpose: Solves $A**T*x = b$
 - rocblas_operation_conjugate_transpose: Solves $A**H*x = b$
- **diag** – [in] [rocblas_diagonal]
 - rocblas_diagonal_unit: each A_i is assumed to be unit triangular (i.e. the diagonal elements of each A_i are not used in computations).
 - rocblas_diagonal_non_unit: each A_i is not assumed to be unit triangular.

- **n** – [in] [rocblas_int] n specifies the number of rows of each b_i . $n \geq 0$.
- **AP** – [in] device pointer pointing to the first packed matrix A_1 , of dimension $\geq (n * (n + 1) / 2)$.
- **stride_A** – [in] [rocblas_stride] stride from the beginning of one packed matrix (AP_i) and the next (AP_{i+1}).
- **x** – [inout] device pointer pointing to the first input vector b_1 . Overwritten by each x_i on output.
- **incx** – [in] [rocblas_int] specifies the increment for the elements of each x_i .
- **stride_x** – [in] [rocblas_stride] stride from the beginning of one vector (x_i) and the next (x_{i+1}).
- **batch_count** – [in] [rocblas_int] specifies the number of instances in the batch.

5.5.15 rocblas_Xtrmv + batched, strided_batched

rocblas_status **rocblas_strmv**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_diagonal* diag, *rocblas_int* m, const float *A, *rocblas_int* lda, float *x, *rocblas_int* incx)

rocblas_status **rocblas_dtrmv**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_diagonal* diag, *rocblas_int* m, const double *A, *rocblas_int* lda, double *x, *rocblas_int* incx)

rocblas_status **rocblas_ctrmv**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_diagonal* diag, *rocblas_int* m, const *rocblas_float_complex* *A, *rocblas_int* lda, *rocblas_float_complex* *x, *rocblas_int* incx)

rocblas_status **rocblas_ztrmv**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_diagonal* diag, *rocblas_int* m, const *rocblas_double_complex* *A, *rocblas_int* lda, *rocblas_double_complex* *x, *rocblas_int* incx)

BLAS Level 2 API

trmv 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.
The vector x is overwritten.
```

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill]
 - rocblas_fill_upper: A is an upper triangular matrix.
 - rocblas_fill_lower: A is a lower triangular matrix.
- **transA** – [in] [rocblas_operation]
- **diag** – [in] [rocblas_diagonal]
 - rocblas_diagonal_unit: A is assumed to be unit triangular.

- `rocblas_diagonal_non_unit`: A is not assumed to be unit triangular.
- **m** – [in] [rocblas_int] m specifies the number of rows of A. $m \geq 0$.
- **A** – [in] device pointer storing matrix A, of dimension (lda, m).
- **lda** – [in] [rocblas_int] specifies the leading dimension of A. $lda = \max(1, m)$.
- **x** – [in] device pointer storing vector x.
- **incx** – [in] [rocblas_int] specifies the increment for the elements of x.

rocblas_status **rocblas_strmv_batched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_diagonal* diag, *rocblas_int* m, const float *const *A, *rocblas_int* lda, float *const *x, *rocblas_int* incx, *rocblas_int* batch_count)

rocblas_status **rocblas_dtrmv_batched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_diagonal* diag, *rocblas_int* m, const double *const *A, *rocblas_int* lda, double *const *x, *rocblas_int* incx, *rocblas_int* batch_count)

rocblas_status **rocblas_ctrmv_batched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_diagonal* diag, *rocblas_int* m, const *rocblas_float_complex* *const *A, *rocblas_int* lda, *rocblas_float_complex* *const *x, *rocblas_int* incx, *rocblas_int* batch_count)

rocblas_status **rocblas_ztrmv_batched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_diagonal* diag, *rocblas_int* m, const *rocblas_double_complex* *const *A, *rocblas_int* lda, *rocblas_double_complex* *const *x, *rocblas_int* incx, *rocblas_int* batch_count)

BLAS Level 2 API

`trmv_batched` performs one of the matrix-vector operations:

```
x_i = A_i * x_i or
x_i = A**T * x_i, 0 < i < batch_count
where x_i is an n element vector and A_i is an n by n (unit, or non-unit, upper or
↳ lower triangular matrix)
The vectors x_i are overwritten.
```

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill]
 - `rocblas_fill_upper`: A_i is an upper triangular matrix.
 - `rocblas_fill_lower`: A_i is a lower triangular matrix.
- **transA** – [in] [rocblas_operation]
- **diag** – [in] [rocblas_diagonal]
 - `rocblas_diagonal_unit`: A_i is assumed to be unit triangular.
 - `rocblas_diagonal_non_unit`: A_i is not assumed to be unit triangular.
- **m** – [in] [rocblas_int] m specifies the number of rows of matrices A_i. $m \geq 0$.
- **A** – [in] device pointer storing pointer of matrices A_i, of dimension (lda, m)

- **lda** – [in] [rocblas_int] specifies the leading dimension of A_i. $\text{lda} \geq \max(1, m)$.
- **x** – [in] device pointer storing vectors x_i.
- **incx** – [in] [rocblas_int] specifies the increment for the elements of vectors x_i.
- **batch_count** – [in] [rocblas_int] The number of batched matrices/vectors.

```
rocblas_status rocblas_strmv_strided_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_operation
transA, rocblas_diagonal diag, rocblas_int m, const float *A,
rocblas_int lda, rocblas_stride stride_A, float *x, rocblas_int
incx, rocblas_stride stride_x, rocblas_int batch_count)
```

```
rocblas_status rocblas_dtrmv_strided_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_operation
transA, rocblas_diagonal diag, rocblas_int m, const double
*A, rocblas_int lda, rocblas_stride stride_A, double *x,
rocblas_int incx, rocblas_stride stride_x, rocblas_int
batch_count)
```

```
rocblas_status rocblas_ctrmv_strided_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_operation
transA, rocblas_diagonal diag, rocblas_int m, const
rocblas_float_complex *A, rocblas_int lda, rocblas_stride
stride_A, rocblas_float_complex *x, rocblas_int incx,
rocblas_stride stride_x, rocblas_int batch_count)
```

```
rocblas_status rocblas_ztrmv_strided_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_operation
transA, rocblas_diagonal diag, rocblas_int m, const
rocblas_double_complex *A, rocblas_int lda, rocblas_stride
stride_A, rocblas_double_complex *x, rocblas_int incx,
rocblas_stride stride_x, rocblas_int batch_count)
```

BLAS Level 2 API

trmv_strided_batched performs one of the matrix-vector operations:

```
x_i = A_i * x_i or
x_i = A**T * x_i, 0 < i < batch_count
where x_i is an n element vector and A_i is an n by n (unit, or non-unit, upper or
↳ lower triangular matrix)
with strides specifying how to retrieve $x_i$ (resp. $A_i$) from $x_{i-1}$ (resp.
↳ $A_i$).
```

The vectors x_i are overwritten.

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill]
 - rocblas_fill_upper: A_i is an upper triangular matrix.
 - rocblas_fill_lower: A_i is a lower triangular matrix.
- **transA** – [in] [rocblas_operation]
- **diag** – [in] [rocblas_diagonal]
 - rocblas_diagonal_unit: A_i is assumed to be unit triangular.
 - rocblas_diagonal_non_unit: A_i is not assumed to be unit triangular.

- **m** – [in] [rocblas_int] m specifies the number of rows of matrices A_i. m ≥ 0.
- **A** – [in] device pointer of the matrix A₀, of dimension (lda, m).
- **lda** – [in] [rocblas_int] specifies the leading dimension of A_i. lda ≥ max(1, m).
- **stride_A** – [in] [rocblas_stride] stride from the start of one A_i matrix to the next A_{i + 1}.
- **x** – [in] device pointer storing the vector x₀.
- **incx** – [in] [rocblas_int] specifies the increment for the elements of one vector x.
- **stride_x** – [in] [rocblas_stride] stride from the start of one x_i vector to the next x_{i + 1}.
- **batch_count** – [in] [rocblas_int] The number of batched matrices/vectors.

5.5.16 rocblas_Xtrsv + batched, strided_batched

rocblas_status **rocblas_strsv**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_diagonal* diag, *rocblas_int* m, const float *A, *rocblas_int* lda, float *x, *rocblas_int* incx)

rocblas_status **rocblas_dtrsv**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_diagonal* diag, *rocblas_int* m, const double *A, *rocblas_int* lda, double *x, *rocblas_int* incx)

rocblas_status **rocblas_ctrsv**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_diagonal* diag, *rocblas_int* m, const *rocblas_float_complex* *A, *rocblas_int* lda, *rocblas_float_complex* *x, *rocblas_int* incx)

rocblas_status **rocblas_ztrsv**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_diagonal* diag, *rocblas_int* m, const *rocblas_double_complex* *A, *rocblas_int* lda, *rocblas_double_complex* *x, *rocblas_int* incx)

BLAS Level 2 API

trsv solves:

$A \cdot x = b$ **or**
 $A^{**T} \cdot x = b$
 where x **and** b are vectors **and** A **is** a triangular matrix.
 The vector x **is** overwritten on b.

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill]
 - rocblas_fill_upper: A is an upper triangular matrix.
 - rocblas_fill_lower: A is a lower triangular matrix.
- **transA** – [in] [rocblas_operation]
- **diag** – [in] [rocblas_diagonal]
 - rocblas_diagonal_unit: A is assumed to be unit triangular.
 - rocblas_diagonal_non_unit: A is not assumed to be unit triangular.

- **m** – [in] [rocblas_int] m specifies the number of rows of b. $m \geq 0$.
- **A** – [in] device pointer storing matrix A, of dimension (lda, m)
- **lda** – [in] [rocblas_int] specifies the leading dimension of A. $lda = \max(1, m)$.
- **x** – [in] device pointer storing vector x.
- **incx** – [in] [rocblas_int] specifies the increment for the elements of x.

rocblas_status **rocblas_strsv_batched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_diagonal* diag, *rocblas_int* m, const float *const A[], *rocblas_int* lda, float *const x[], *rocblas_int* incx, *rocblas_int* batch_count)

rocblas_status **rocblas_dtrsv_batched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_diagonal* diag, *rocblas_int* m, const double *const A[], *rocblas_int* lda, double *const x[], *rocblas_int* incx, *rocblas_int* batch_count)

rocblas_status **rocblas_ctrsv_batched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_diagonal* diag, *rocblas_int* m, const *rocblas_float_complex* *const A[], *rocblas_int* lda, *rocblas_float_complex* *const x[], *rocblas_int* incx, *rocblas_int* batch_count)

rocblas_status **rocblas_ztrsv_batched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_diagonal* diag, *rocblas_int* m, const *rocblas_double_complex* *const A[], *rocblas_int* lda, *rocblas_double_complex* *const x[], *rocblas_int* incx, *rocblas_int* batch_count)

BLAS Level 2 API

trsv_batched solves:

```
A_i*x_i = b_i or
A_i**T*x_i = b_i
where (A_i, x_i, b_i) is the i-th instance of the batch.
x_i and b_i are vectors and A_i is an
m by m triangular matrix.
```

The vector x is overwritten on b.

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill]
 - rocblas_fill_upper: A is an upper triangular matrix.
 - rocblas_fill_lower: A is a lower triangular matrix.
- **transA** – [in] [rocblas_operation]
- **diag** – [in] [rocblas_diagonal]
 - rocblas_diagonal_unit: A is assumed to be unit triangular.
 - rocblas_diagonal_non_unit: A is not assumed to be unit triangular.
- **m** – [in] [rocblas_int] m specifies the number of rows of b. $m \geq 0$.
- **A** – [in] device array of device pointers storing each matrix A_i.
- **lda** – [in] [rocblas_int] specifies the leading dimension of each A_i. $lda = \max(1, m)$

- **x** – [in] device array of device pointers storing each vector x_i .
- **incx** – [in] [rocblas_int] specifies the increment for the elements of x .
- **batch_count** – [in] [rocblas_int] number of instances in the batch.

rocblas_status **rocblas_strsv_strided_batched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_diagonal* diag, *rocblas_int* m, const float *A, *rocblas_int* lda, *rocblas_stride* stride_A, float *x, *rocblas_int* incx, *rocblas_stride* stride_x, *rocblas_int* batch_count)

rocblas_status **rocblas_dtrsv_strided_batched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_diagonal* diag, *rocblas_int* m, const double *A, *rocblas_int* lda, *rocblas_stride* stride_A, double *x, *rocblas_int* incx, *rocblas_stride* stride_x, *rocblas_int* batch_count)

rocblas_status **rocblas_ctrsv_strided_batched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_diagonal* diag, *rocblas_int* m, const *rocblas_float_complex* *A, *rocblas_int* lda, *rocblas_stride* stride_A, *rocblas_float_complex* *x, *rocblas_int* incx, *rocblas_stride* stride_x, *rocblas_int* batch_count)

rocblas_status **rocblas_ztrsv_strided_batched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_diagonal* diag, *rocblas_int* m, const *rocblas_double_complex* *A, *rocblas_int* lda, *rocblas_stride* stride_A, *rocblas_double_complex* *x, *rocblas_int* incx, *rocblas_stride* stride_x, *rocblas_int* batch_count)

BLAS Level 2 API

trsv_strided_batched solves:

```
A_i*x_i = b_i or
A_i**T*x_i = b_i
where (A_i, x_i, b_i) is the i-th instance of the batch.
x_i and b_i are vectors and A_i is an m by m triangular matrix, for i = 1, ...,
↪ batch_count.
```

The vector x is overwritten on b .

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill]
 - *rocblas_fill_upper*: A is an upper triangular matrix.
 - *rocblas_fill_lower*: A is a lower triangular matrix.
- **transA** – [in] [rocblas_operation]
- **diag** – [in] [rocblas_diagonal]
 - *rocblas_diagonal_unit*: A is assumed to be unit triangular.
 - *rocblas_diagonal_non_unit*: A is not assumed to be unit triangular.
- **m** – [in] [rocblas_int] m specifies the number of rows of each b_i . $m \geq 0$.
- **A** – [in] device pointer to the first matrix (A_1) in the batch, of dimension (lda, m).

- **stride_A** – [in] [rocblas_stride] stride from the start of one A_i matrix to the next $A_{(i+1)}$.
- **lda** – [in] [rocblas_int] specifies the leading dimension of each A_i . $lda = \max(1, m)$.
- **x** – [inout] device pointer to the first vector (x_1) in the batch.
- **stride_x** – [in] [rocblas_stride] stride from the start of one x_i vector to the next $x_{(i+1)}$
- **incx** – [in] [rocblas_int] specifies the increment for the elements of each x_i .
- **batch_count** – [in] [rocblas_int] number of instances in the batch.

5.5.17 rocblas_Xhemv + batched, strided_batched

```
rocblas_status rocblas_chemv(rocblas_handle handle, rocblas_fill uplo, rocblas_int n, const
    rocblas_float_complex *alpha, const rocblas_float_complex *A, rocblas_int lda,
    const rocblas_float_complex *x, rocblas_int incx, const rocblas_float_complex
    *beta, rocblas_float_complex *y, rocblas_int incy)
```

```
rocblas_status rocblas_zhemv(rocblas_handle handle, rocblas_fill uplo, rocblas_int n, const
    rocblas_double_complex *alpha, const rocblas_double_complex *A, rocblas_int
    lda, const rocblas_double_complex *x, rocblas_int incx, const
    rocblas_double_complex *beta, rocblas_double_complex *y, rocblas_int incy)
```

BLAS Level 2 API

hemv performs one of the matrix-vector operations:

```
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.
```

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill]
 - rocblas_fill_upper: the upper triangular part of the Hermitian matrix A is supplied.
 - rocblas_fill_lower: the lower triangular part of the Hermitian matrix A is supplied.
- **n** – [in] [rocblas_int] the order of the matrix A.
- **alpha** – [in] device pointer or host pointer to scalar alpha.
- **A** – [in] device pointer storing matrix A. Of dimension (lda, n).

```
if uplo == rocblas_fill_upper:
    The upper triangular part of A must contain
    the upper triangular part of a Hermitian matrix. The lower
    triangular part of A will not be referenced.

if uplo == rocblas_fill_lower:
    The lower triangular part of A must contain
    the lower triangular part of a Hermitian matrix. The upper
    triangular part of A will not be referenced.
    As a Hermitian matrix, the imaginary part of the main diagonal
    of A will not be referenced and is assumed to be == 0.
```

- **lda** – [in] [rocblas_int] specifies the leading dimension of A. must be $\geq \max(1, n)$.
- **x** – [in] device pointer storing vector x.
- **incx** – [in] [rocblas_int] specifies the increment for the elements of x.
- **beta** – [in] device pointer or host pointer to scalar beta.
- **y** – [inout] device pointer storing vector y.
- **incy** – [in] [rocblas_int] specifies the increment for the elements of y.

```
rocblas_status rocblas_chemv_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_int n, const
    rocblas_float_complex *alpha, const rocblas_float_complex *const A[],
    rocblas_int lda, const rocblas_float_complex *const x[], rocblas_int
    incx, const rocblas_float_complex *beta, rocblas_float_complex *const
    y[], rocblas_int incy, rocblas_int batch_count)
```

```
rocblas_status rocblas_zhemv_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_int n, const
    rocblas_double_complex *alpha, const rocblas_double_complex *const
    A[], rocblas_int lda, const rocblas_double_complex *const x[],
    rocblas_int incx, const rocblas_double_complex *beta,
    rocblas_double_complex *const y[], rocblas_int incy, rocblas_int
    batch_count)
```

BLAS Level 2 API

hemv_batched performs one of the matrix-vector operations:

```
y_i := alpha*A_i*x_i + beta*y_i
where alpha and beta are scalars, x_i and y_i are n element vectors and A_i is an
n by n Hermitian matrix, for each batch in i = [1, batch_count].
```

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill]
 - rocblas_fill_upper: the upper triangular part of the Hermitian matrix A is supplied.
 - rocblas_fill_lower: the lower triangular part of the Hermitian matrix A is supplied.
- **n** – [in] [rocblas_int] the order of each matrix A_i.
- **alpha** – [in] device pointer or host pointer to scalar alpha.
- **A** – [in] device array of device pointers storing each matrix A_i of dimension (lda, n).

```
if uplo == rocblas_fill_upper:
    The upper triangular part of each A_i must contain
    the upper triangular part of a Hermitian matrix. The lower
    triangular part of each A_i will not be referenced.

if uplo == rocblas_fill_lower:
    The lower triangular part of each A_i must contain
    the lower triangular part of a Hermitian matrix. The upper
    triangular part of each A_i will not be referenced.
    As a Hermitian matrix, the imaginary part of the main diagonal
    of each A_i will not be referenced and is assumed to be == 0.
```

- **lda** – [in] [rocblas_int] specifies the leading dimension of each A_i. must be $\geq \max(1, n)$.
- **x** – [in] device array of device pointers storing each vector x_i.
- **incx** – [in] [rocblas_int] specifies the increment for the elements of each x_i.
- **beta** – [in] device pointer or host pointer to scalar beta.
- **y** – [inout] device array of device pointers storing each vector y_i.
- **incy** – [in] [rocblas_int] specifies the increment for the elements of y.
- **batch_count** – [in] [rocblas_int] number of instances in the batch.

```
rocblas_status rocblas_chemv_strided_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_int n, const
rocblas_float_complex *alpha, const rocblas_float_complex
*A, rocblas_int lda, rocblas_stride stride_A, const
rocblas_float_complex *x, rocblas_int incx, rocblas_stride
stride_x, const rocblas_float_complex *beta,
rocblas_float_complex *y, rocblas_int incy, rocblas_stride
stride_y, rocblas_int batch_count)
```

```
rocblas_status rocblas_zhemv_strided_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_int n, const
rocblas_double_complex *alpha, const
rocblas_double_complex *A, rocblas_int lda, rocblas_stride
stride_A, const rocblas_double_complex *x, rocblas_int incx,
rocblas_stride stride_x, const rocblas_double_complex *beta,
rocblas_double_complex *y, rocblas_int incy, rocblas_stride
stride_y, rocblas_int batch_count)
```

BLAS Level 2 API

hemv_strided_batched performs one of the matrix-vector operations:

```
y_i := alpha*A_i*x_i + beta*y_i
where alpha and beta are scalars, x_i and y_i are n element vectors and A_i is an
n by n Hermitian matrix, for each batch in i = [1, batch_count].
```

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill]
 - rocblas_fill_upper: the upper triangular part of the Hermitian matrix A is supplied.
 - rocblas_fill_lower: the lower triangular part of the Hermitian matrix A is supplied.
- **n** – [in] [rocblas_int] the order of each matrix A_i.
- **alpha** – [in] device pointer or host pointer to scalar alpha.
- **A** – [in] device array of device pointers storing each matrix A_i of dimension (lda, n).

```
if uplo == rocblas_fill_upper:
    The upper triangular part of each A_i must contain
    the upper triangular part of a Hermitian matrix. The lower
    triangular part of each A_i will not be referenced.

if uplo == rocblas_fill_lower:
    The lower triangular part of each A_i must contain
```

(continues on next page)

(continued from previous page)

the lower triangular part of a Hermitian matrix. The upper triangular part of each A_i will **not** be referenced. As a Hermitian matrix, the imaginary part of the main diagonal of each A_i will **not** be referenced **and is** assumed to be `== 0`.

- **lda** – [in] [rocblas_int] specifies the leading dimension of each A_i . must be $\geq \max(1, n)$.
- **stride_A** – [in] [rocblas_stride] stride from the start of one (A_i) to the next (A_{i+1}).
- **x** – [in] device array of device pointers storing each vector x_i .
- **incx** – [in] [rocblas_int] specifies the increment for the elements of each x_i .
- **stride_x** – [in] [rocblas_stride] stride from the start of one vector (x_i) and the next one (x_{i+1}).
- **beta** – [in] device pointer or host pointer to scalar beta.
- **y** – [inout] device array of device pointers storing each vector y_i .
- **incy** – [in] [rocblas_int] specifies the increment for the elements of y .
- **stride_y** – [in] [rocblas_stride] stride from the start of one vector (y_i) and the next one (y_{i+1}).
- **batch_count** – [in] [rocblas_int] number of instances in the batch.

5.5.18 rocblas_Xhbm + batched, strided_batched

rocblas_status rocblas_chbm(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_int* n, *rocblas_int* k, const *rocblas_float_complex* *alpha, const *rocblas_float_complex* *A, *rocblas_int* lda, const *rocblas_float_complex* *x, *rocblas_int* incx, const *rocblas_float_complex* *beta, *rocblas_float_complex* *y, *rocblas_int* incy)

rocblas_status rocblas_zhbm(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_int* n, *rocblas_int* k, const *rocblas_double_complex* *alpha, const *rocblas_double_complex* *A, *rocblas_int* lda, const *rocblas_double_complex* *x, *rocblas_int* incx, const *rocblas_double_complex* *beta, *rocblas_double_complex* *y, *rocblas_int* incy)

BLAS Level 2 API

hbm performs the matrix-vector operations:

$y := \alpha A x + \beta y$
 where α **and** β are scalars, x **and** y are n element vectors **and** A **is** an n by n Hermitian band matrix, **with** k **super**-diagonals.

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill]
 - `rocblas_fill_upper`: The upper triangular part of A is being supplied.
 - `rocblas_fill_lower`: The lower triangular part of A is being supplied.
- **n** – [in] [rocblas_int] the order of the matrix A .
- **k** – [in] [rocblas_int] the number of super-diagonals of the matrix A . Must be ≥ 0 .

- **alpha** – [in] device pointer or host pointer to scalar alpha.
- **A** – [in] device pointer storing matrix A. Of dimension (lda, n).

```

if uplo == rocblas_fill_upper:
    The leading (k + 1) by n part of A must contain the upper
    triangular band part of the Hermitian matrix, with the
    ↪leading
    diagonal in row (k + 1), the first super-diagonal on the RHS
    of row k, etc.
    The top left k by x triangle of A will not be referenced.
    Ex (upper, lda = n = 4, k = 1):
        A                                Represented matrix
        (0,0) (5,9) (6,8) (7,7)          (1, 0) (5, 9) (0, 0) (0, 0)
        (1,0) (2,0) (3,0) (4,0)          (5,-9) (2, 0) (6, 8) (0, 0)
        (0,0) (0,0) (0,0) (0,0)          (0, 0) (6,-8) (3, 0) (7, 7)
        (0,0) (0,0) (0,0) (0,0)          (0, 0) (0, 0) (7,-7) (4, 0)

if uplo == rocblas_fill_lower:
    The leading (k + 1) by n part of A must contain the lower
    triangular band part of the Hermitian matrix, with the
    ↪leading
    diagonal in row (1), the first sub-diagonal on the LHS of
    row 2, etc.
    The bottom right k by k triangle of A will not be referenced.
    Ex (lower, lda = 2, n = 4, k = 1):
        A                                Represented matrix
        (1,0) (2,0) (3,0) (4,0)          (1, 0) (5,-9) (0, 0) (0, ↪
    ↪0)                                  (5, 9) (2, 0) (6,-8) (0, ↪
        (5,9) (6,8) (7,7) (0,0)          (0, 0) (6, 8) (3, 0) (7,-
    ↪0)                                  (0, 0) (0, 0) (7, 7) (4, ↪
        ↪7)                                  ↪0)
        ↪0)
    As a Hermitian matrix, the imaginary part of the main diagonal
    of A will not be referenced and is assumed to be == 0.

```

- **lda** – [in] [rocblas_int] specifies the leading dimension of A. must be $\geq k + 1$.
- **x** – [in] device pointer storing vector x.
- **incx** – [in] [rocblas_int] specifies the increment for the elements of x.
- **beta** – [in] device pointer or host pointer to scalar beta.
- **y** – [inout] device pointer storing vector y.
- **incy** – [in] [rocblas_int] specifies the increment for the elements of y.

```

rocblas_status rocblas_chbmv_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_int n, rocblas_int k,
    const rocblas_float_complex *alpha, const rocblas_float_complex *const
    A[], rocblas_int lda, const rocblas_float_complex *const x[], rocblas_int
    incx, const rocblas_float_complex *beta, rocblas_float_complex *const
    y[], rocblas_int incy, rocblas_int batch_count)

```

```

rocblas_status rocblas_zhbmvm_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_int n, rocblas_int k,
const rocblas_double_complex *alpha, const rocblas_double_complex
*const A[], rocblas_int lda, const rocblas_double_complex *const x[],
rocblas_int incx, const rocblas_double_complex *beta,
rocblas_double_complex *const y[], rocblas_int incy, rocblas_int
batch_count)

```

BLAS Level 2 API

hbmvm_batched performs one of the matrix-vector operations:

```

y_i := alpha*A_i*x_i + beta*y_i
where alpha and beta are scalars, x_i and y_i are n element vectors and A_i is an
n by n Hermitian band matrix with k super-diagonals, for each batch in i = [1,
batch_count].

```

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill]
 - rocblas_fill_upper: The upper triangular part of each A_i is being supplied.
 - rocblas_fill_lower: The lower triangular part of each A_i is being supplied.
- **n** – [in] [rocblas_int] the order of each matrix A_i.
- **k** – [in] [rocblas_int] the number of super-diagonals of each matrix A_i. Must be >= 0.
- **alpha** – [in] device pointer or host pointer to scalar alpha.
- **A** – [in] device array of device pointers storing each matrix_i A of dimension (lda, n).

```

if uplo == rocblas_fill_upper:
    The leading (k + 1) by n part of each A_i must contain the
upper
    triangular band part of the Hermitian matrix, with the
leading
    diagonal in row (k + 1), the first super-diagonal on the RHS
    of row k, etc.
    The top left k by x triangle of each A_i will not be
referenced.
    Ex (upper, lda = n = 4, k = 1):
    A
    (0,0) (5,9) (6,8) (7,7)
    (1,0) (2,0) (3,0) (4,0)
    (0,0) (0,0) (0,0) (0,0)
    (0,0) (0,0) (0,0) (0,0)
    Represented matrix
    (1, 0) (5, 9) (0, 0) (0, 0)
    (5,-9) (2, 0) (6, 8) (0, 0)
    (0, 0) (6,-8) (3, 0) (7, 7)
    (0, 0) (0, 0) (7,-7) (4, 0)

if uplo == rocblas_fill_lower:
    The leading (k + 1) by n part of each A_i must contain the
lower
    triangular band part of the Hermitian matrix, with the
leading
    diagonal in row (1), the first sub-diagonal on the LHS of
    row 2, etc.
    The bottom right k by k triangle of each A_i will not be

```

(continues on next page)

(continued from previous page)

```

↪referenced.
        Ex (lower, lda = 2, n = 4, k = 1):
        A                                     Represented matrix
        (1,0) (2,0) (3,0) (4,0)              (1, 0) (5,-9) (0, 0) (0,↪
↪0)
        (5,9) (6,8) (7,7) (0,0)              (5, 9) (2, 0) (6,-8) (0,↪
↪0)
                                           (0, 0) (6, 8) (3, 0) (7,-
↪7)
                                           (0, 0) (0, 0) (7, 7) (4,↪
↪0)

```

As a Hermitian matrix, the imaginary part of the main diagonal of each A_i will **not** be referenced **and is** assumed to be == 0.

- **lda** – [in] [rocblas_int] specifies the leading dimension of each A_i. must be >= max(1, n).
- **x** – [in] device array of device pointers storing each vector x_i.
- **incx** – [in] [rocblas_int] specifies the increment for the elements of each x_i.
- **beta** – [in] device pointer or host pointer to scalar beta.
- **y** – [inout] device array of device pointers storing each vector y_i.
- **incy** – [in] [rocblas_int] specifies the increment for the elements of y.
- **batch_count** – [in] [rocblas_int] number of instances in the batch.

```

rocblas_status rocblas_chbmv_strided_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_int n,
                                             rocblas_int k, const rocblas_float_complex *alpha, const
                                             rocblas_float_complex *A, rocblas_int lda, rocblas_stride
                                             stride_A, const rocblas_float_complex *x, rocblas_int incx,
                                             rocblas_stride stride_x, const rocblas_float_complex *beta,
                                             rocblas_float_complex *y, rocblas_int incy, rocblas_stride
                                             stride_y, rocblas_int batch_count)

```

```

rocblas_status rocblas_zhbmv_strided_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_int n,
                                             rocblas_int k, const rocblas_double_complex *alpha, const
                                             rocblas_double_complex *A, rocblas_int lda, rocblas_stride
                                             stride_A, const rocblas_double_complex *x, rocblas_int incx,
                                             rocblas_stride stride_x, const rocblas_double_complex *beta,
                                             rocblas_double_complex *y, rocblas_int incy, rocblas_stride
                                             stride_y, rocblas_int batch_count)

```

BLAS Level 2 API

hbm_v_strided_batched performs one of the matrix-vector operations:

```

y_i := alpha*A_i*x_i + beta*y_i
where alpha and beta are scalars, x_i and y_i are n element vectors and A_i is an
n by n Hermitian band matrix with k super-diagonals, for each batch in i = [1,↪
↪batch_count].

```

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.

- **uplo** – [in] [rocblas_fill]
 - rocblas_fill_upper: The upper triangular part of each A_i is being supplied.
 - rocblas_fill_lower: The lower triangular part of each A_i is being supplied.
- **n** – [in] [rocblas_int] the order of each matrix A_i.
- **k** – [in] [rocblas_int] the number of super-diagonals of each matrix A_i. Must be ≥ 0 .
- **alpha** – [in] device pointer or host pointer to scalar alpha.
- **A** – [in] device array pointing to the first matrix A_1. Each A_i is of dimension (lda, n).

```

if uplo == rocblas_fill_upper:
    The leading (k + 1) by n part of each A_i must contain the
    ↪upper
    triangular band part of the Hermitian matrix, with the
    ↪leading
    diagonal in row (k + 1), the first super-diagonal on the RHS
    of row k, etc.
    The top left k by x triangle of each A_i will not be
    ↪referenced.
    Ex (upper, lda = n = 4, k = 1):
    A                                     Represented matrix
    (0,0) (5,9) (6,8) (7,7)             (1, 0) (5, 9) (0, 0) (0, 0)
    (1,0) (2,0) (3,0) (4,0)             (5,-9) (2, 0) (6, 8) (0, 0)
    (0,0) (0,0) (0,0) (0,0)             (0, 0) (6,-8) (3, 0) (7, 7)
    (0,0) (0,0) (0,0) (0,0)             (0, 0) (0, 0) (7,-7) (4, 0)

if uplo == rocblas_fill_lower:
    The leading (k + 1) by n part of each A_i must contain the
    ↪lower
    triangular band part of the Hermitian matrix, with the
    ↪leading
    diagonal in row (1), the first sub-diagonal on the LHS of
    row 2, etc.
    The bottom right k by k triangle of each A_i will not be
    ↪referenced.
    Ex (lower, lda = 2, n = 4, k = 1):
    A                                     Represented matrix
    (1,0) (2,0) (3,0) (4,0)             (1, 0) (5,-9) (0, 0) (0,
    ↪0)
    (5,9) (6,8) (7,7) (0,0)             (5, 9) (2, 0) (6,-8) (0,
    ↪0)
    (0, 0) (6, 8) (3, 0) (7,-
    ↪7)
    (0, 0) (0, 0) (7, 7) (4,
    ↪0)

    As a Hermitian matrix, the imaginary part of the main diagonal
    of each A_i will not be referenced and is assumed to be == 0.

```

- **lda** – [in] [rocblas_int] specifies the leading dimension of each A_i. must be $\geq \max(1, n)$.
- **stride_A** – [in] [rocblas_stride] stride from the start of one matrix (A_i) and the next one (A_{i+1}).

- **x** – [in] device array pointing to the first vector y_1 .
- **incx** – [in] [rocblas_int] specifies the increment for the elements of each x_i .
- **stride_x** – [in] [rocblas_stride] stride from the start of one vector (x_i) and the next one (x_{i+1}).
- **beta** – [in] device pointer or host pointer to scalar beta.
- **y** – [inout] device array pointing to the first vector y_1 .
- **incy** – [in] [rocblas_int] specifies the increment for the elements of y.
- **stride_y** – [in] [rocblas_stride] stride from the start of one vector (y_i) and the next one (y_{i+1}).
- **batch_count** – [in] [rocblas_int] number of instances in the batch.

5.5.19 rocblas_Xhpmv + batched, strided_batched

```
rocblas_status rocblas_chpmv(rocblas_handle handle, rocblas_fill uplo, rocblas_int n, const
    rocblas_float_complex *alpha, const rocblas_float_complex *AP, const
    rocblas_float_complex *x, rocblas_int incx, const rocblas_float_complex *beta,
    rocblas_float_complex *y, rocblas_int incy)
```

```
rocblas_status rocblas_zhpmv(rocblas_handle handle, rocblas_fill uplo, rocblas_int n, const
    rocblas_double_complex *alpha, const rocblas_double_complex *AP, const
    rocblas_double_complex *x, rocblas_int incx, const rocblas_double_complex
    *beta, rocblas_double_complex *y, rocblas_int incy)
```

BLAS Level 2 API

hpmv 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 (see description below).
```

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill]
 - rocblas_fill_upper: the upper triangular part of the Hermitian matrix A is supplied in AP.
 - rocblas_fill_lower: the lower triangular part of the Hermitian matrix A is supplied in AP.
- **n** – [in] [rocblas_int] the order of the matrix A. Must be ≥ 0 .
- **alpha** – [in] device pointer or host pointer to scalar alpha.
- **AP** – [in] device pointer storing the packed version of the specified triangular portion of the Hermitian matrix A. Of at least size $((n * (n + 1)) / 2)$.

```
if uplo == rocblas_fill_upper:
    The upper triangular portion of the Hermitian matrix A is
    supplied.
    The matrix is compacted so that AP contains the triangular
    portion
```

(continues on next page)

(continued from previous page)

```

column-by-column
so that:
AP(0) = A(0,0)
AP(1) = A(0,1)
AP(2) = A(1,1), etc.
Ex: (rocblas_fill_upper; n = 3)
    (1, 0) (2, 1) (3, 2)
    (2,-1) (4, 0) (5,-1) ---> [(1,0),(2,1),(4,0),(3,2),(5,-1),
    ↪(6,0)]
    (3,-2) (5, 1) (6, 0)

if uplo == rocblas_fill_lower:
    The lower triangular portion of the Hermitian matrix A is
    ↪supplied.
    The matrix is compacted so that AP contains the triangular
    ↪portion
    column-by-column
    so that:
    AP(0) = A(0,0)
    AP(1) = A(1,0)
    AP(2) = A(2,1), etc.
    Ex: (rocblas_fill_lower; n = 3)
    (1, 0) (2, 1) (3, 2)
    (2,-1) (4, 0) (5,-1) ---> [(1,0),(2,-1),(3,-2),(4,0),(5,1),
    ↪(6,0)]
    (3,-2) (5, 1) (6, 0)

```

Note that the imaginary part of the diagonal elements are not accessed and are assumed to be 0.

- **x** – [in] device pointer storing vector x.
- **incx** – [in] [rocblas_int] specifies the increment for the elements of x.
- **beta** – [in] device pointer or host pointer to scalar beta.
- **y** – [inout] device pointer storing vector y.
- **incy** – [in] [rocblas_int] specifies the increment for the elements of y.

```

rocblas_status rocblas_chpmv_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_int n, const
    rocblas_float_complex *alpha, const rocblas_float_complex *const AP[],
    const rocblas_float_complex *const x[], rocblas_int incx, const
    rocblas_float_complex *beta, rocblas_float_complex *const y[],
    rocblas_int incy, rocblas_int batch_count)

```

```

rocblas_status rocblas_zhpmv_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_int n, const
    rocblas_double_complex *alpha, const rocblas_double_complex *const
    AP[], const rocblas_double_complex *const x[], rocblas_int incx, const
    rocblas_double_complex *beta, rocblas_double_complex *const y[],
    rocblas_int incy, rocblas_int batch_count)

```

BLAS Level 2 API

hpmv_batched performs the matrix-vector operation:

```
y_i := alpha*A_i*x_i + beta*y_i
where alpha and beta are scalars, x_i and y_i are n element vectors and A_i is an
n by n Hermitian matrix, supplied in packed form (see description below),
for each batch in i = [1, batch_count].
```

Parameters

- **handle** – [in] [rocbblas_handle] handle to the rocbblas library context queue.
- **uplo** – [in] [rocbblas_fill]
 - rocbblas_fill_upper: the upper triangular part of each Hermitian matrix A_i is supplied in AP.
 - rocbblas_fill_lower: the lower triangular part of each Hermitian matrix A_i is supplied in AP.
- **n** – [in] [rocbblas_int] the order of each matrix A_i.
- **alpha** – [in] device pointer or host pointer to scalar alpha.
- **AP** – [in] device pointer of device pointers storing the packed version of the specified triangular portion of each Hermitian matrix A_i. Each A_i is of at least size $((n * (n + 1)) / 2)$.

```
if uplo == rocbblas_fill_upper:
    The upper triangular portion of each Hermitian matrix A_i is
    supplied.
    The matrix is compacted so that each AP_i contains the
    triangular portion
    column-by-column
    so that:
    AP(0) = A(0,0)
    AP(1) = A(0,1)
    AP(2) = A(1,1), etc.
    Ex: (rocbblas_fill_upper; n = 3)
        (1, 0) (2, 1) (3, 2)
        (2,-1) (4, 0) (5,-1) ---> [(1,0),(2,1),(4,0),(3,2),(5,-
    1),(6,0)]
        (3,-2) (5, 1) (6, 0)

if uplo == rocbblas_fill_lower:
    The lower triangular portion of each Hermitian matrix A_i is
    supplied.
    The matrix is compacted so that each AP_i contains the
    triangular portion
    column-by-column
    so that:
    AP(0) = A(0,0)
    AP(1) = A(1,0)
    AP(2) = A(2,1), etc.
    Ex: (rocbblas_fill_lower; n = 3)
        (1, 0) (2, 1) (3, 2)
        (2,-1) (4, 0) (5,-1) ---> [(1,0),(2,-1),(3,-2),(4,0),(5,
    1),(6,0)]
        (3,-2) (5, 1) (6, 0)
```

(continues on next page)

(continued from previous page)

Note that the imaginary part of the diagonal elements are **not** accessed and are assumed to be 0.

- **x** – [in] device array of device pointers storing each vector x_i .
- **incx** – [in] [rocblas_int] specifies the increment for the elements of each x_i .
- **beta** – [in] device pointer or host pointer to scalar beta.
- **y** – [inout] device array of device pointers storing each vector y_i .
- **incy** – [in] [rocblas_int] specifies the increment for the elements of y .
- **batch_count** – [in] [rocblas_int] number of instances in the batch.

rocblas_status **rocblas_chpmv_strided_batched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_int* n, const *rocblas_float_complex* *alpha, const *rocblas_float_complex* *AP, *rocblas_stride* stride_A, const *rocblas_float_complex* *x, *rocblas_int* incx, *rocblas_stride* stride_x, const *rocblas_float_complex* *beta, *rocblas_float_complex* *y, *rocblas_int* incy, *rocblas_stride* stride_y, *rocblas_int* batch_count)

rocblas_status **rocblas_zhpmv_strided_batched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_int* n, const *rocblas_double_complex* *alpha, const *rocblas_double_complex* *AP, *rocblas_stride* stride_A, const *rocblas_double_complex* *x, *rocblas_int* incx, *rocblas_stride* stride_x, const *rocblas_double_complex* *beta, *rocblas_double_complex* *y, *rocblas_int* incy, *rocblas_stride* stride_y, *rocblas_int* batch_count)

BLAS Level 2 API

hpmv_strided_batched performs the matrix-vector operation:

$y_i := \alpha A_i x_i + \beta y_i$
 where α and β are scalars, x_i and y_i are n element vectors and A_i is an n by n Hermitian matrix, supplied in packed form (see description below),
 for each batch in $i = [1, \text{batch_count}]$.

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill]
 - *rocblas_fill_upper*: the upper triangular part of each Hermitian matrix A_i is supplied in AP.
 - *rocblas_fill_lower*: the lower triangular part of each Hermitian matrix A_i is supplied in AP.
- **n** – [in] [rocblas_int] the order of each matrix A_i .
- **alpha** – [in] device pointer or host pointer to scalar alpha.
- **AP** – [in] device pointer pointing to the beginning of the first matrix (AP_1). Stores the packed version of the specified triangular portion of each Hermitian matrix AP_i of size $((n * (n + 1)) / 2)$.

```

if uplo == rocblas_fill_upper:
    The upper triangular portion of each Hermitian matrix A_i is
    ↪supplied.
    The matrix is compacted so that each AP_i contains the
    ↪triangular portion
    column-by-column
    so that:
    AP(0) = A(0,0)
    AP(1) = A(0,1)
    AP(2) = A(1,1), etc.
    Ex: (rocblas_fill_upper; n = 3)
        (1, 0) (2, 1) (3, 2)
        (2,-1) (4, 0) (5,-1) ---> [(1,0),(2,1),(4,0),(3,2),(5,-1),
    ↪(6,0)]
        (3,-2) (5, 1) (6, 0)

if uplo == rocblas_fill_lower:
    The lower triangular portion of each Hermitian matrix A_i is
    ↪supplied.
    The matrix is compacted so that each AP_i contains the
    ↪triangular portion
    column-by-column
    so that:
    AP(0) = A(0,0)
    AP(1) = A(1,0)
    AP(2) = A(2,1), etc.
    Ex: (rocblas_fill_lower; n = 3)
        (1, 0) (2, 1) (3, 2)
        (2,-1) (4, 0) (5,-1) ---> [(1,0),(2,-1),(3,-2),(4,0),(5,1),
    ↪(6,0)]
        (3,-2) (5, 1) (6, 0)
    Note that the imaginary part of the diagonal elements are not
    ↪accessed
    and are assumed to be 0.

```

- **stride_A** – [in] [rocblas_stride] stride from the start of one matrix (AP_i) and the next one (AP_{i+1}).
- **x** – [in] device array pointing to the beginning of the first vector (x_1).
- **incx** – [in] [rocblas_int] specifies the increment for the elements of each x_i.
- **stride_x** – [in] [rocblas_stride] stride from the start of one vector (x_i) and the next one (x_{i+1}).
- **beta** – [in] device pointer or host pointer to scalar beta.
- **y** – [inout] device array pointing to the beginning of the first vector (y_1).
- **incy** – [in] [rocblas_int] specifies the increment for the elements of y.
- **stride_y** – [in] [rocblas_stride] stride from the start of one vector (y_i) and the next one (y_{i+1}).
- **batch_count** – [in] [rocblas_int] number of instances in the batch.

5.5.20 rocblas_Xher + batched, strided_batched

rocblas_status **rocblas_cher**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_int* n, const float *alpha, const *rocblas_float_complex* *x, *rocblas_int* incx, *rocblas_float_complex* *A, *rocblas_int* lda)

rocblas_status **rocblas_zher**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_int* n, const double *alpha, const *rocblas_double_complex* *x, *rocblas_int* incx, *rocblas_double_complex* *A, *rocblas_int* lda)

BLAS Level 2 API

her performs the matrix-vector operations:

$A := A + \alpha x x^H$
 where α is a real scalar, x is a vector, and A is an
 n by n Hermitian matrix.

Parameters

- **handle** – [in] [*rocblas_handle*] handle to the rocblas library context queue.
- **uplo** – [in] [*rocblas_fill*] specifies whether the upper ‘rocblas_fill_upper’ or lower ‘rocblas_fill_lower’
 - rocblas_fill_upper: The upper triangular part of A is supplied in A.
 - rocblas_fill_lower: The lower triangular part of A is supplied in A.
- **n** – [in] [*rocblas_int*] the number of rows and columns of matrix A. Must be at least 0.
- **alpha** – [in] device pointer or host pointer to scalar alpha.
- **x** – [in] device pointer storing vector x.
- **incx** – [in] [*rocblas_int*] specifies the increment for the elements of x.
- **A** – [inout] device pointer storing the specified triangular portion of the Hermitian matrix A. Of size (lda * n).

```

    if uplo == rocblas_fill_upper:
        The upper triangular portion of the Hermitian matrix A
        is supplied.
        The lower triangular portion will not be touched.

    if uplo == rocblas_fill_lower:
        The lower triangular portion of the Hermitian matrix A
        is supplied.
        The upper triangular portion will not be touched.
        Note that the imaginary part of the diagonal elements are
        not accessed
        and are assumed to be 0.
  
```

- **lda** – [in] [*rocblas_int*] specifies the leading dimension of A. Must be at least max(1, n).

rocblas_status **rocblas_cher_batched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_int* n, const float *alpha, const *rocblas_float_complex* *const x[], *rocblas_int* incx, *rocblas_float_complex* *const A[], *rocblas_int* lda, *rocblas_int* batch_count)


```
rocblas_status rocblas_zher_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_int n, const double
                                     *alpha, const rocblas_double_complex *const x[], rocblas_int incx,
                                     rocblas_double_complex *const A[], rocblas_int lda, rocblas_int
                                     batch_count)
```

BLAS Level 2 API

her_batched performs the matrix-vector operations:

$A_i := A_i + \alpha x_i x_i^* H$
 where α is a real scalar, x_i is a vector, and A_i is an
 n by n symmetric matrix, for $i = 1, \dots, \text{batch_count}$.

Parameters

- **handle** – [in] [*rocblas_handle*] handle to the rocblas library context queue.
- **uplo** – [in] [*rocblas_fill*] specifies whether the upper ‘rocblas_fill_upper’ or lower ‘rocblas_fill_lower’
 - rocblas_fill_upper: The upper triangular part of each A_i is supplied in A .
 - rocblas_fill_lower: The lower triangular part of each A_i is supplied in A .
- **n** – [in] [*rocblas_int*] the number of rows and columns of each matrix A_i . Must be at least 0.
- **alpha** – [in] device pointer or host pointer to scalar α .
- **x** – [in] device array of device pointers storing each vector x_i .
- **incx** – [in] [*rocblas_int*] specifies the increment for the elements of each x_i .
- **A** – [inout] device array of device pointers storing the specified triangular portion of each Hermitian matrix A_i of at least size $((n * (n + 1)) / 2)$. Array is of at least size batch_count.

```
if uplo == rocblas_fill_upper:
    The upper triangular portion of each Hermitian matrix A_
    i is supplied.
    The lower triangular portion of each A_i will not be
    touched.
if uplo == rocblas_fill_lower:
    The lower triangular portion of each Hermitian matrix A_
    i is supplied.
    The upper triangular portion of each A_i will not be
    touched.
    Note that the imaginary part of the diagonal elements are
    not accessed
    and are assumed to be 0.
```

- **lda** – [in] [*rocblas_int*] specifies the leading dimension of each A_i . Must be at least $\max(1, n)$.
- **batch_count** – [in] [*rocblas_int*] number of instances in the batch.

```
rocblas_status rocblas_cher_strided_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_int n, const
                                              float *alpha, const rocblas_float_complex *x, rocblas_int incx,
                                              rocblas_stride stride_x, rocblas_float_complex *A, rocblas_int
                                              lda, rocblas_stride stride_A, rocblas_int batch_count)
```

```
rocblas_status rocblas_zher_strided_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_int n, const
double *alpha, const rocblas_double_complex *x, rocblas_int
incx, rocblas_stride stride_x, rocblas_double_complex *A,
rocblas_int lda, rocblas_stride stride_A, rocblas_int
batch_count)
```

BLAS Level 2 API

her_strided_batched performs the matrix-vector operations:

$A_i := A_i + \alpha x_i x_i^* H$
 where α **is** a real scalar, x_i **is** a vector, **and** A_i **is** an
 n by n Hermitian matrix, **for** $i = 1, \dots, \text{batch_count}$.

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill] specifies whether the upper ‘rocblas_fill_upper’ or lower ‘rocblas_fill_lower’
 - rocblas_fill_upper: The upper triangular part of each A_i is supplied in A.
 - rocblas_fill_lower: The lower triangular part of each A_i is supplied in A.
- **n** – [in] [rocblas_int] the number of rows and columns of each matrix A_i . Must be at least 0.
- **alpha** – [in] device pointer or host pointer to scalar alpha.
- **x** – [in] device pointer pointing to the first vector (x_1).
- **incx** – [in] [rocblas_int] specifies the increment for the elements of each x_i .
- **stride_x** – [in] [rocblas_stride] stride from the start of one vector (x_i) and the next one (x_{i+1}).
- **A** – [inout] device array of device pointers storing the specified triangular portion of each Hermitian matrix A_i . Points to the first matrix (A_1).

```
if uplo == rocblas_fill_upper:
    The upper triangular portion of each Hermitian matrix  $A_i$  is
    ↪supplied.
    The lower triangular portion of each  $A_i$  will not be touched.

if uplo == rocblas_fill_lower:
    The lower triangular portion of each Hermitian matrix  $A_i$  is
    ↪supplied.
    The upper triangular portion of each  $A_i$  will not be touched.
```

Note that the imaginary part of the diagonal elements are not accessed and are assumed to be 0.

- **lda** – [in] [rocblas_int] specifies the leading dimension of each A_i .
- **stride_A** – [in] [rocblas_stride] stride from the start of one (A_i) and the next (A_{i+1}).
- **batch_count** – [in] [rocblas_int] number of instances in the batch.

5.5.21 rocblas_Xher2 + batched, strided_batched

```
rocblas_status rocblas_cher2(rocblas_handle handle, rocblas_fill uplo, rocblas_int n, const
    rocblas_float_complex *alpha, const rocblas_float_complex *x, rocblas_int incx,
    const rocblas_float_complex *y, rocblas_int incy, rocblas_float_complex *A,
    rocblas_int lda)
```

```
rocblas_status rocblas_zher2(rocblas_handle handle, rocblas_fill uplo, rocblas_int n, const
    rocblas_double_complex *alpha, const rocblas_double_complex *x, rocblas_int
    incx, const rocblas_double_complex *y, rocblas_int incy, rocblas_double_complex
    *A, rocblas_int lda)
```

BLAS Level 2 API

her2 performs the matrix-vector operations:

$A := A + \alpha x y^* H + \text{conj}(\alpha) y x^* H$
 where α is a complex scalar, x and y are vectors, and A is an
 n by n Hermitian matrix.

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill] specifies whether the upper ‘rocblas_fill_upper’ or lower ‘rocblas_fill_lower’
 - rocblas_fill_upper: The upper triangular part of A is supplied.
 - rocblas_fill_lower: The lower triangular part of A is supplied.
- **n** – [in] [rocblas_int] the number of rows and columns of matrix A . Must be at least 0.
- **alpha** – [in] device pointer or host pointer to scalar α .
- **x** – [in] device pointer storing vector x .
- **incx** – [in] [rocblas_int] specifies the increment for the elements of x .
- **y** – [in] device pointer storing vector y .
- **incy** – [in] [rocblas_int] specifies the increment for the elements of y .
- **A** – [inout] device pointer storing the specified triangular portion of the Hermitian matrix A .
 Of size (lda, n) .

```
if uplo == rocblas_fill_upper:
    The upper triangular portion of the Hermitian matrix  $A$  is supplied.
    The lower triangular portion of  $A$  will not be touched.

if uplo == rocblas_fill_lower:
    The lower triangular portion of the Hermitian matrix  $A$  is supplied.
    The upper triangular portion of  $A$  will not be touched.
```

Note that the imaginary part of the diagonal elements are not accessed and are assumed to be 0.

- **lda** – [in] [rocblas_int] specifies the leading dimension of A . Must be at least $\max(lda, 1)$.

```
rocblas_status rocblas_cher2_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_int n, const
    rocblas_float_complex *alpha, const rocblas_float_complex *const x[],
    rocblas_int incx, const rocblas_float_complex *const y[], rocblas_int
    incy, rocblas_float_complex *const A[], rocblas_int lda, rocblas_int
    batch_count)
```

```
rocblas_status rocblas_zher2_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_int n, const
    rocblas_double_complex *alpha, const rocblas_double_complex *const
    x[], rocblas_int incx, const rocblas_double_complex *const y[],
    rocblas_int incy, rocblas_double_complex *const A[], rocblas_int lda,
    rocblas_int batch_count)
```

BLAS Level 2 API

her2_batched performs the matrix-vector operations:

$A_i := A_i + \alpha x_i y_i^H + \text{conj}(\alpha) y_i x_i^H$
 where α is a complex scalar, x_i and y_i are vectors, and A_i is an
 n by n Hermitian matrix for each batch $i = [1, \text{batch_count}]$.

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill] specifies whether the upper ‘rocblas_fill_upper’ or lower ‘rocblas_fill_lower’
 - rocblas_fill_upper: The upper triangular part of each A_i is supplied.
 - rocblas_fill_lower: The lower triangular part of each A_i is supplied.
- **n** – [in] [rocblas_int] the number of rows and columns of each matrix A_i . Must be at least 0.
- **alpha** – [in] device pointer or host pointer to scalar α .
- **x** – [in] device array of device pointers storing each vector x_i .
- **incx** – [in] [rocblas_int] specifies the increment for the elements of x .
- **y** – [in] device array of device pointers storing each vector y_i .
- **incy** – [in] [rocblas_int] specifies the increment for the elements of each y_i .
- **A** – [inout] device array of device pointers storing the specified triangular portion of each Hermitian matrix A_i of size (lda, n) .

```
if uplo == rocblas_fill_upper:
    The upper triangular portion of each Hermitian matrix  $A_i$  is
    supplied.
    The lower triangular portion of each  $A_i$  will not be touched.

if uplo == rocblas_fill_lower:
    The lower triangular portion of each Hermitian matrix  $A_i$  is
    supplied.
    The upper triangular portion of each  $A_i$  will not be touched.
```

Note that the imaginary part of the diagonal elements are not accessed and are assumed to be 0.

- **lda** – [in] [rocblas_int] specifies the leading dimension of each A_i . Must be at least $\max(\text{lda}, 1)$.
- **batch_count** – [in] [rocblas_int] number of instances in the batch.

```
rocblas_status rocblas_cher2_strided_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_int n, const
rocblas_float_complex *alpha, const rocblas_float_complex
*x, rocblas_int incx, rocblas_stride stride_x, const
rocblas_float_complex *y, rocblas_int incy, rocblas_stride
stride_y, rocblas_float_complex *A, rocblas_int lda,
rocblas_stride stride_A, rocblas_int batch_count)
```

```
rocblas_status rocblas_zher2_strided_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_int n, const
rocblas_double_complex *alpha, const
rocblas_double_complex *x, rocblas_int incx, rocblas_stride
stride_x, const rocblas_double_complex *y, rocblas_int incy,
rocblas_stride stride_y, rocblas_double_complex *A,
rocblas_int lda, rocblas_stride stride_A, rocblas_int
batch_count)
```

BLAS Level 2 API

her2_strided_batched performs the matrix-vector operations:

$A_i := A_i + \alpha x_i y_i^H + \text{conj}(\alpha) y_i x_i^H$
 where α is a complex scalar, x_i and y_i are vectors, and A_i is an
 n by n Hermitian matrix for each batch $i = [1, \text{batch_count}]$.

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill] specifies whether the upper ‘rocblas_fill_upper’ or lower ‘rocblas_fill_lower’
 - rocblas_fill_upper: The upper triangular part of each A_i is supplied.
 - rocblas_fill_lower: The lower triangular part of each A_i is supplied.
- **n** – [in] [rocblas_int] the number of rows and columns of each matrix A_i . Must be at least 0.
- **alpha** – [in] device pointer or host pointer to scalar α .
- **x** – [in] device pointer pointing to the first vector x_1 .
- **incx** – [in] [rocblas_int] specifies the increment for the elements of each x_i .
- **stride_x** – [in] [rocblas_stride] specifies the stride between the beginning of one vector (x_i) and the next (x_{i+1}).
- **y** – [in] device pointer pointing to the first vector y_1 .
- **incy** – [in] [rocblas_int] specifies the increment for the elements of each y_i .
- **stride_y** – [in] [rocblas_stride] specifies the stride between the beginning of one vector (y_i) and the next (y_{i+1}).
- **A** – [inout] device pointer pointing to the first matrix (A_1). Stores the specified triangular portion of each Hermitian matrix A_i .

```

if uplo == rocblas_fill_upper:
    The upper triangular portion of each Hermitian matrix A_i is
    supplied.
    The lower triangular portion of each A_i will not be touched.

if uplo == rocblas_fill_lower:
    The lower triangular portion of each Hermitian matrix A_i is
    supplied.
    The upper triangular portion of each A_i will not be touched.

```

Note that the imaginary part of the diagonal elements are not accessed and are assumed to be 0.

- **lda** – [in] [rocblas_int] specifies the leading dimension of each A_i. Must be at least $\max(\text{lda}, 1)$.
- **stride_A** – [in] [rocblas_stride] specifies the stride between the beginning of one matrix (A_i) and the next (A_i+1).
- **batch_count** – [in] [rocblas_int] number of instances in the batch.

5.5.22 rocblas_Xhpr + batched, strided_batched

rocblas_status **rocblas_chpr**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_int* n, const float *alpha, const *rocblas_float_complex* *x, *rocblas_int* incx, *rocblas_float_complex* *AP)

rocblas_status **rocblas_zhpr**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_int* n, const double *alpha, const *rocblas_double_complex* *x, *rocblas_int* incx, *rocblas_double_complex* *AP)

BLAS Level 2 API

hpr performs the matrix-vector operations:

```

A := A + alpha*x*x**H
where alpha is a real scalar, x is a vector, and A is an
n by n Hermitian matrix, supplied in packed form.

```

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill] specifies whether the upper ‘rocblas_fill_upper’ or lower ‘rocblas_fill_lower’
 - rocblas_fill_upper: The upper triangular part of A is supplied in AP.
 - rocblas_fill_lower: The lower triangular part of A is supplied in AP.
- **n** – [in] [rocblas_int] the number of rows and columns of matrix A. Must be at least 0.
- **alpha** – [in] device pointer or host pointer to scalar alpha.
- **x** – [in] device pointer storing vector x.
- **incx** – [in] [rocblas_int] specifies the increment for the elements of x.
- **AP** – [inout] device pointer storing the packed version of the specified triangular portion of the Hermitian matrix A. Of at least size $((n * (n + 1)) / 2)$.

```

if uplo == rocblas_fill_upper:
    The upper triangular portion of the Hermitian matrix A is
    supplied.
    The matrix is compacted so that AP contains the triangular
    portion
    column-by-column
    so that:
    AP(0) = A(0,0)
    AP(1) = A(0,1)
    AP(2) = A(1,1), etc.
    Ex: (rocblas_fill_upper; n = 3)
        (1, 0) (2, 1) (4,9)
        (2,-1) (3, 0) (5,3) ---> [(1,0),(2,1),(3,0),(4,9),(5,3),(6,
    0)]
        (4,-9) (5,-3) (6,0)

if uplo == rocblas_fill_lower:
    The lower triangular portion of the Hermitian matrix A is
    supplied.
    The matrix is compacted so that AP contains the triangular
    portion
    column-by-column
    so that:
    AP(0) = A(0,0)
    AP(1) = A(1,0)
    AP(2) = A(2,1), etc.
    Ex: (rocblas_fill_lower; n = 3)
        (1, 0) (2, 1) (4,9)
        (2,-1) (3, 0) (5,3) ---> [(1,0),(2,-1),(4,-9),(3,0),(5,-3),
    (6,0)]
        (4,-9) (5,-3) (6,0)
    Note that the imaginary part of the diagonal elements are not
    accessed
    and are assumed to be 0.

```

```

rocblas_status rocblas_chpr_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_int n, const float *alpha,
    const rocblas_float_complex *const x[], rocblas_int incx,
    rocblas_float_complex *const AP[], rocblas_int batch_count)

```

```

rocblas_status rocblas_zhpr_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_int n, const double
    *alpha, const rocblas_double_complex *const x[], rocblas_int incx,
    rocblas_double_complex *const AP[], rocblas_int batch_count)

```

BLAS Level 2 API

hpr_batched performs the matrix-vector operations:

```

A_i := A_i + alpha*x_i*x_i**H
where alpha is a real scalar, x_i is a vector, and A_i is an
n by n symmetric matrix, supplied in packed form, for i = 1, ..., batch_count.

```

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.

- **uplo** – [in] [rocblas_fill] specifies whether the upper ‘rocblas_fill_upper’ or lower ‘rocblas_fill_lower’
 - rocblas_fill_upper: The upper triangular part of each A_i is supplied in AP.
 - rocblas_fill_lower: The lower triangular part of each A_i is supplied in AP.
- **n** – [in] [rocblas_int] the number of rows and columns of each matrix A_i. Must be at least 0.
- **alpha** – [in] device pointer or host pointer to scalar alpha.
- **x** – [in] device array of device pointers storing each vector x_i.
- **incx** – [in] [rocblas_int] specifies the increment for the elements of each x_i.
- **AP** – [inout] device array of device pointers storing the packed version of the specified triangular portion of each Hermitian matrix A_i of at least size $((n * (n + 1)) / 2)$. Array is of at least size batch_count.

```

if uplo == rocblas_fill_upper:
    The upper triangular portion of each Hermitian matrix A_i is
    supplied.
    The matrix is compacted so that AP contains the triangular
    portion
    column-by-column
    so that:
    AP(0) = A(0,0)
    AP(1) = A(0,1)
    AP(2) = A(1,1), etc.
    Ex: (rocblas_fill_upper; n = 3)
        (1, 0) (2, 1) (4,9)
        (2,-1) (3, 0) (5,3) ----> [(1,0),(2,1),(3,0),(4,9),(5,3),(6,
    0)]
        (4,-9) (5,-3) (6,0)

if uplo == rocblas_fill_lower:
    The lower triangular portion of each Hermitian matrix A_i is
    supplied.
    The matrix is compacted so that AP contains the triangular
    portion
    column-by-column
    so that:
    AP(0) = A(0,0)
    AP(1) = A(1,0)
    AP(2) = A(2,1), etc.
    Ex: (rocblas_fill_lower; n = 3)
        (1, 0) (2, 1) (4,9)
        (2,-1) (3, 0) (5,3) ----> [(1,0),(2,-1),(4,-9),(3,0),(5,-3),
    (6,0)]
        (4,-9) (5,-3) (6,0)
    Note that the imaginary part of the diagonal elements are not
    accessed
    and are assumed to be 0.
  
```

- **batch_count** – [in] [rocblas_int] number of instances in the batch.


```
rocblas_status rocblas_chpr_strided_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_int n, const
float *alpha, const rocblas_float_complex *x, rocblas_int incx,
rocblas_stride stride_x, rocblas_float_complex *AP,
rocblas_stride stride_A, rocblas_int batch_count)
```

```
rocblas_status rocblas_zhpr_strided_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_int n, const
double *alpha, const rocblas_double_complex *x, rocblas_int incx, rocblas_stride stride_x, rocblas_double_complex *AP,
rocblas_stride stride_A, rocblas_int batch_count)
```

BLAS Level 2 API

hpr_strided_batched performs the matrix-vector operations:

```
A_i := A_i + alpha*x_i*x_i**H
where alpha is a real scalar, x_i is a vector, and A_i is an
n by n symmetric matrix, supplied in packed form, for i = 1, ..., batch_count.
```

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill] specifies whether the upper ‘rocblas_fill_upper’ or lower ‘rocblas_fill_lower’
 - rocblas_fill_upper: The upper triangular part of each A_i is supplied in AP.
 - rocblas_fill_lower: The lower triangular part of each A_i is supplied in AP.
- **n** – [in] [rocblas_int] the number of rows and columns of each matrix A_i. Must be at least 0.
- **alpha** – [in] device pointer or host pointer to scalar alpha.
- **x** – [in] device pointer pointing to the first vector (x_1).
- **incx** – [in] [rocblas_int] specifies the increment for the elements of each x_i.
- **stride_x** – [in] [rocblas_stride] stride from the start of one vector (x_i) and the next one (x_{i+1}).
- **AP** – [inout] device array of device pointers storing the packed version of the specified triangular portion of each Hermitian matrix A_i. Points to the first matrix (A_1).

```
if uplo == rocblas_fill_upper:
    The upper triangular portion of each Hermitian matrix A_i is
    supplied.
    The matrix is compacted so that AP contains the triangular
    portion
    column-by-column
    so that:
    AP(0) = A(0,0)
    AP(1) = A(0,1)
    AP(2) = A(1,1), etc.
    Ex: (rocblas_fill_upper; n = 3)
        (1, 0) (2, 1) (4,9)
        (2,-1) (3, 0) (5,3) ---> [(1,0), (2,1), (3,0), (4,9), (5,3), (6,
        0)]
        (4,-9) (5,-3) (6,0)
```

(continues on next page)

(continued from previous page)

```

if uplo == rocblas_fill_lower:
    The lower triangular portion of each Hermitian matrix A_i is
    supplied.
    The matrix is compacted so that AP contains the triangular
    portion
    column-by-column
    so that:
    AP(0) = A(0,0)
    AP(1) = A(1,0)
    AP(2) = A(2,1), etc.
    Ex: (rocblas_fill_lower; n = 3)
        (1, 0) (2, 1) (4,9)
        (2,-1) (3, 0) (5,3) ----> [(1,0), (2,-1), (4,-9), (3,0), (5,-3),
    (6,0)]
        (4,-9) (5,-3) (6,0)
    Note that the imaginary part of the diagonal elements are not
    accessed
    and are assumed to be 0.

```

- **stride_A** – [in] [rocblas_stride] stride from the start of one (A_i) and the next (A_i+1).
- **batch_count** – [in] [rocblas_int] number of instances in the batch.

5.5.23 rocblas_Xhpr2 + batched, strided_batched

```

rocblas_status rocblas_chpr2(rocblas_handle handle, rocblas_fill uplo, rocblas_int n, const
    rocblas_float_complex *alpha, const rocblas_float_complex *x, rocblas_int incx,
    const rocblas_float_complex *y, rocblas_int incy, rocblas_float_complex *AP)

```

```

rocblas_status rocblas_zhpr2(rocblas_handle handle, rocblas_fill uplo, rocblas_int n, const
    rocblas_double_complex *alpha, const rocblas_double_complex *x, rocblas_int
    incx, const rocblas_double_complex *y, rocblas_int incy, rocblas_double_complex
    *AP)

```

BLAS Level 2 API

hpr2 performs the matrix-vector operations:

```

A := A + alpha*x*y**H + conj(alpha)*y*x**H
where alpha is a complex scalar, x and y are vectors, and A is an
n by n Hermitian matrix, supplied in packed form.

```

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill] specifies whether the upper ‘rocblas_fill_upper’ or lower ‘rocblas_fill_lower’
 - rocblas_fill_upper: The upper triangular part of A is supplied in AP.
 - rocblas_fill_lower: The lower triangular part of A is supplied in AP.
- **n** – [in] [rocblas_int] the number of rows and columns of matrix A. Must be at least 0.

- **alpha** – [in] device pointer or host pointer to scalar alpha.
- **x** – [in] device pointer storing vector x.
- **incx** – [in] [rocblas_int] specifies the increment for the elements of x.
- **y** – [in] device pointer storing vector y.
- **incy** – [in] [rocblas_int] specifies the increment for the elements of y.
- **AP** – [inout] device pointer storing the packed version of the specified triangular portion of the Hermitian matrix A. Of at least size $((n * (n + 1)) / 2)$.

```

if uplo == rocblas_fill_upper:
    The upper triangular portion of the Hermitian matrix A is
    supplied.
    The matrix is compacted so that AP contains the triangular
    portion
    column-by-column
    so that:
    AP(0) = A(0,0)
    AP(1) = A(0,1)
    AP(2) = A(1,1), etc.
    Ex: (rocblas_fill_upper; n = 3)
        (1, 0) (2, 1) (4,9)
        (2,-1) (3, 0) (5,3) ----> [(1,0),(2,1),(3,0),(4,9),(5,3),(6,
    (0)]
        (4,-9) (5,-3) (6,0)

if uplo == rocblas_fill_lower:
    The lower triangular portion of the Hermitian matrix A is
    supplied.
    The matrix is compacted so that AP contains the triangular
    portion
    column-by-column
    so that:
    AP(0) = A(0,0)
    AP(1) = A(1,0)
    AP(2) = A(2,1), etc.
    Ex: (rocblas_fill_lower; n = 3)
        (1, 0) (2, 1) (4,9)
        (2,-1) (3, 0) (5,3) ----> [(1,0),(2,-1),(4,-9),(3,0),(5,-3),
    (6,0)]
        (4,-9) (5,-3) (6,0)
    Note that the imaginary part of the diagonal elements are not
    accessed
    and are assumed to be 0.

```

```

rocblas_status rocblas_chpr2_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_int n, const
rocblas_float_complex *alpha, const rocblas_float_complex *const x[],
rocblas_int incx, const rocblas_float_complex *const y[], rocblas_int
incy, rocblas_float_complex *const AP[], rocblas_int batch_count)

```

```

rocblas_status rocblas_zhpr2_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_int n, const
    rocblas_double_complex *alpha, const rocblas_double_complex *const
    x[], rocblas_int incx, const rocblas_double_complex *const y[],
    rocblas_int incy, rocblas_double_complex *const AP[], rocblas_int
    batch_count)

```

BLAS Level 2 API

hpr2_batched performs the matrix-vector operations:

$A_i := A_i + \alpha x_i y_i^* H + \text{conj}(\alpha) y_i x_i^* H$
 where α is a complex scalar, x_i and y_i are vectors, and A_i is an
 n by n symmetric matrix, supplied in packed form, for $i = 1, \dots, \text{batch_count}$.

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill] specifies whether the upper ‘rocblas_fill_upper’ or lower ‘rocblas_fill_lower’
 - rocblas_fill_upper: The upper triangular part of each A_i is supplied in AP.
 - rocblas_fill_lower: The lower triangular part of each A_i is supplied in AP.
- **n** – [in] [rocblas_int] the number of rows and columns of each matrix A_i . Must be at least 0.
- **alpha** – [in] device pointer or host pointer to scalar α .
- **x** – [in] device array of device pointers storing each vector x_i .
- **incx** – [in] [rocblas_int] specifies the increment for the elements of each x_i .
- **y** – [in] device array of device pointers storing each vector y_i .
- **incy** – [in] [rocblas_int] specifies the increment for the elements of each y_i .
- **AP** – [inout] device array of device pointers storing the packed version of the specified triangular portion of each Hermitian matrix A_i of at least size $((n * (n + 1)) / 2)$. Array is of at least size batch_count.

```

if uplo == rocblas_fill_upper:
    The upper triangular portion of each Hermitian matrix  $A_i$  is
    supplied.
    The matrix is compacted so that AP contains the triangular
    portion
    column-by-column
    so that:
    AP(0) = A(0,0)
    AP(1) = A(0,1)
    AP(2) = A(1,1), etc.
    Ex: (rocblas_fill_upper; n = 3)
        (1, 0) (2, 1) (4,9)
        (2,-1) (3, 0) (5,3) ---> [(1,0), (2,1), (3,0), (4,9), (5,3), (6,
    0)]
        (4,-9) (5,-3) (6,0)

```

```

if uplo == rocblas_fill_lower:
    The lower triangular portion of each Hermitian matrix  $A_i$  is

```

(continues on next page)

(continued from previous page)

```

↪supplied.
    The matrix is compacted so that AP contains the triangular_
↪portion
    column-by-column
    so that:
    AP(0) = A(0,0)
    AP(1) = A(1,0)
    AP(2) = A(2,1), etc.
    Ex: (rocblas_fill_lower; n = 3)
        (1, 0) (2, 1) (4,9)
        (2,-1) (3, 0) (5,3) --> [(1,0),(2,-1),(4,-9),(3,0),(5,-3),
↪(6,0)]
        (4,-9) (5,-3) (6,0)
    Note that the imaginary part of the diagonal elements are not_
↪accessed
    and are assumed to be 0.

```

- **batch_count** – [in] [rocblas_int] number of instances in the batch.

```

rocblas_status rocblas_chpr2_strided_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_int n, const
rocblas_float_complex *alpha, const rocblas_float_complex
*x, rocblas_int incx, rocblas_stride stride_x, const
rocblas_float_complex *y, rocblas_int incy, rocblas_stride
stride_y, rocblas_float_complex *AP, rocblas_stride stride_A,
rocblas_int batch_count)

```

```

rocblas_status rocblas_zhpr2_strided_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_int n, const
rocblas_double_complex *alpha, const
rocblas_double_complex *x, rocblas_int incx, rocblas_stride
stride_x, const rocblas_double_complex *y, rocblas_int incy,
rocblas_stride stride_y, rocblas_double_complex *AP,
rocblas_stride stride_A, rocblas_int batch_count)

```

BLAS Level 2 API

hpr2_strided_batched performs the matrix-vector operations:

```

A_i := A_i + alpha*x_i*y_i**H + conj(alpha)*y_i*x_i**H
where alpha is a complex scalar, x_i and y_i are vectors, and A_i is an
n by n symmetric matrix, supplied in packed form, for i = 1, ..., batch_count.

```

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill] specifies whether the upper ‘rocblas_fill_upper’ or lower ‘rocblas_fill_lower’
 - rocblas_fill_upper: The upper triangular part of each A_i is supplied in AP.
 - rocblas_fill_lower: The lower triangular part of each A_i is supplied in AP.
- **n** – [in] [rocblas_int] the number of rows and columns of each matrix A_i. Must be at least 0.
- **alpha** – [in] device pointer or host pointer to scalar alpha.

- **x** – [in] device pointer pointing to the first vector (x_1).
- **incx** – [in] [rocblas_int] specifies the increment for the elements of each x_i.
- **stride_x** – [in] [rocblas_stride] stride from the start of one vector (x_i) and the next one (x_i+1).
- **y** – [in] device pointer pointing to the first vector (y_1).
- **incy** – [in] [rocblas_int] specifies the increment for the elements of each y_i.
- **stride_y** – [in] [rocblas_stride] stride from the start of one vector (y_i) and the next one (y_i+1).
- **AP** – [inout] device array of device pointers storing the packed version of the specified triangular portion of each Hermitian matrix A_i. Points to the first matrix (A_1).

```

if uplo == rocblas_fill_upper:
    The upper triangular portion of each Hermitian matrix A_i is
    supplied.
    The matrix is compacted so that AP contains the triangular
    portion
    column-by-column
    so that:
    AP(0) = A(0,0)
    AP(1) = A(0,1)
    AP(2) = A(1,1), etc.
    Ex: (rocblas_fill_upper; n = 3)
        (1, 0) (2, 1) (4,9)
        (2,-1) (3, 0) (5,3) ---> [(1,0),(2,1),(3,0),(4,9),(5,3),(6,
    0)]
        (4,-9) (5,-3) (6,0)

if uplo == rocblas_fill_lower:
    The lower triangular portion of each Hermitian matrix A_i is
    supplied.
    The matrix is compacted so that AP contains the triangular
    portion
    column-by-column
    so that:
    AP(0) = A(0,0)
    AP(1) = A(1,0)
    AP(2) = A(2,1), etc.
    Ex: (rocblas_fill_lower; n = 3)
        (1, 0) (2, 1) (4,9)
        (2,-1) (3, 0) (5,3) ---> [(1,0),(2,-1),(4,-9),(3,0),(5,
    -3),(6,0)]
        (4,-9) (5,-3) (6,0)
    Note that the imaginary part of the diagonal elements are not
    accessed
    and are assumed to be 0.

```

- **stride_A** – [in] [rocblas_stride] stride from the start of one (A_i) and the next (A_i+1).
- **batch_count** – [in] [rocblas_int] number of instances in the batch.

5.6 rocBLAS Level-3 functions

5.6.1 rocblas_Xgemm + batched, strided_batched

```
rocblas_status rocblas_sgemm(rocblas_handle handle, rocblas_operation transA, rocblas_operation transB,
                               rocblas_int m, rocblas_int n, rocblas_int k, const float *alpha, const float *A,
                               rocblas_int lda, const float *B, rocblas_int ldb, const float *beta, float *C,
                               rocblas_int ldc)
```

```
rocblas_status rocblas_dgemm(rocblas_handle handle, rocblas_operation transA, rocblas_operation transB,
                               rocblas_int m, rocblas_int n, rocblas_int k, const double *alpha, const double *A,
                               rocblas_int lda, const double *B, rocblas_int ldb, const double *beta, double *C,
                               rocblas_int ldc)
```

```
rocblas_status rocblas_hgemm(rocblas_handle handle, rocblas_operation transA, rocblas_operation transB,
                               rocblas_int m, rocblas_int n, rocblas_int k, const rocblas_half *alpha, const
                               rocblas_half *A, rocblas_int lda, const rocblas_half *B, rocblas_int ldb, const
                               rocblas_half *beta, rocblas_half *C, rocblas_int ldc)
```

```
rocblas_status rocblas_cgemm(rocblas_handle handle, rocblas_operation transA, rocblas_operation transB,
                               rocblas_int m, rocblas_int n, rocblas_int k, const rocblas_float_complex *alpha,
                               const rocblas_float_complex *A, rocblas_int lda, const rocblas_float_complex *B,
                               rocblas_int ldb, const rocblas_float_complex *beta, rocblas_float_complex *C,
                               rocblas_int ldc)
```

```
rocblas_status rocblas_zgemm(rocblas_handle handle, rocblas_operation transA, rocblas_operation transB,
                               rocblas_int m, rocblas_int n, rocblas_int k, const rocblas_double_complex *alpha,
                               const rocblas_double_complex *A, rocblas_int lda, const rocblas_double_complex
                               *B, rocblas_int ldb, const rocblas_double_complex *beta,
                               rocblas_double_complex *C, rocblas_int ldc)
```

BLAS Level 3 API

gemm performs one of the matrix-matrix operations:

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

where $\text{op}(X)$ is one of

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

alpha **and** beta are scalars, **and** A, B **and** C are matrices, **with**
 $\text{op}(A)$ an m by k matrix, $\text{op}(B)$ a k by n matrix **and** C an m by n matrix.

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **transA** – [in] [rocblas_operation] specifies the form of $\text{op}(A)$.
- **transB** – [in] [rocblas_operation] specifies the form of $\text{op}(B)$.
- **m** – [in] [rocblas_int] number of rows of matrices $\text{op}(A)$ and C.
- **n** – [in] [rocblas_int] number of columns of matrices $\text{op}(B)$ and C.

- **k** – [in] [rocblas_int] number of columns of matrix op(A) and number of rows of matrix op(B).
- **alpha** – [in] device pointer or host pointer specifying the scalar alpha.
- **A** – [in] device pointer storing matrix A.
- **lda** – [in] [rocblas_int] specifies the leading dimension of A.
- **B** – [in] device pointer storing matrix B.
- **ldb** – [in] [rocblas_int] specifies the leading dimension of B.
- **beta** – [in] device pointer or host pointer specifying the scalar beta.
- **C** – [inout] device pointer storing matrix C on the GPU.
- **ldc** – [in] [rocblas_int] specifies the leading dimension of C.

rocblas_status **rocblas_sgemv_batched**(*rocblas_handle* handle, *rocblas_operation* transA, *rocblas_operation* transB, *rocblas_int* m, *rocblas_int* n, *rocblas_int* k, const float *alpha, const float *const A[], *rocblas_int* lda, const float *const B[], *rocblas_int* ldb, const float *beta, float *const C[], *rocblas_int* ldc, *rocblas_int* batch_count)

rocblas_status **rocblas_dgemv_batched**(*rocblas_handle* handle, *rocblas_operation* transA, *rocblas_operation* transB, *rocblas_int* m, *rocblas_int* n, *rocblas_int* k, const double *alpha, const double *const A[], *rocblas_int* lda, const double *const B[], *rocblas_int* ldb, const double *beta, double *const C[], *rocblas_int* ldc, *rocblas_int* batch_count)

rocblas_status **rocblas_hgemv_batched**(*rocblas_handle* handle, *rocblas_operation* transA, *rocblas_operation* transB, *rocblas_int* m, *rocblas_int* n, *rocblas_int* k, const *rocblas_half* *alpha, const *rocblas_half* *const A[], *rocblas_int* lda, const *rocblas_half* *const B[], *rocblas_int* ldb, const *rocblas_half* *beta, *rocblas_half* *const C[], *rocblas_int* ldc, *rocblas_int* batch_count)

rocblas_status **rocblas_cgemv_batched**(*rocblas_handle* handle, *rocblas_operation* transA, *rocblas_operation* transB, *rocblas_int* m, *rocblas_int* n, *rocblas_int* k, const *rocblas_float_complex* *alpha, const *rocblas_float_complex* *const A[], *rocblas_int* lda, const *rocblas_float_complex* *const B[], *rocblas_int* ldb, const *rocblas_float_complex* *beta, *rocblas_float_complex* *const C[], *rocblas_int* ldc, *rocblas_int* batch_count)

rocblas_status **rocblas_zgemv_batched**(*rocblas_handle* handle, *rocblas_operation* transA, *rocblas_operation* transB, *rocblas_int* m, *rocblas_int* n, *rocblas_int* k, const *rocblas_double_complex* *alpha, const *rocblas_double_complex* *const A[], *rocblas_int* lda, const *rocblas_double_complex* *const B[], *rocblas_int* ldb, const *rocblas_double_complex* *beta, *rocblas_double_complex* *const C[], *rocblas_int* ldc, *rocblas_int* batch_count)

BLAS Level 3 API

gemm_batched performs one of the batched matrix-matrix operations:

$C_i = \alpha * op(A_i) * op(B_i) + \beta * C_i$, for $i = 1, \dots, \text{batch_count}$,

where $op(X)$ is one of

(continues on next page)

(continued from previous page)

```

op( X ) = X      or
op( X ) = X**T   or
op( X ) = X**H,

```

alpha and beta are scalars, and A, B and C are strided batched matrices, with

op(A) an m by k by batch_count strided_batched matrix,
 op(B) an k by n by batch_count strided_batched matrix and
 C an m by n by batch_count strided_batched matrix.

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **transA** – [in] [rocblas_operation] specifies the form of op(A).
- **transB** – [in] [rocblas_operation] specifies the form of op(B).
- **m** – [in] [rocblas_int] matrix dimension m.
- **n** – [in] [rocblas_int] matrix dimension n.
- **k** – [in] [rocblas_int] matrix dimension k.
- **alpha** – [in] device pointer or host pointer specifying the scalar alpha.
- **A** – [in] device array of device pointers storing each matrix A_i.
- **lda** – [in] [rocblas_int] specifies the leading dimension of each A_i.
- **B** – [in] device array of device pointers storing each matrix B_i.
- **ldb** – [in] [rocblas_int] specifies the leading dimension of each B_i.
- **beta** – [in] device pointer or host pointer specifying the scalar beta.
- **C** – [inout] device array of device pointers storing each matrix C_i.
- **ldc** – [in] [rocblas_int] specifies the leading dimension of each C_i.
- **batch_count** – [in] [rocblas_int] number of gemm operations in the batch.

```

rocblas_status rocblas_sgemv_strided_batched(rocblas_handle handle, rocblas_operation transA,
                                              rocblas_operation transB, rocblas_int m, rocblas_int n,
                                              rocblas_int k, const float *alpha, const float *A, rocblas_int
                                              lda, rocblas_stride stride_a, const float *B, rocblas_int ldb,
                                              rocblas_stride stride_b, const float *beta, float *C, rocblas_int
                                              ldc, rocblas_stride stride_c, rocblas_int batch_count)

```

```

rocblas_status rocblas_dgemv_strided_batched(rocblas_handle handle, rocblas_operation transA,
                                              rocblas_operation transB, rocblas_int m, rocblas_int n,
                                              rocblas_int k, const double *alpha, const double *A,
                                              rocblas_int lda, rocblas_stride stride_a, const double *B,
                                              rocblas_int ldb, rocblas_stride stride_b, const double *beta,
                                              double *C, rocblas_int ldc, rocblas_stride stride_c,
                                              rocblas_int batch_count)

```

```
rocblas_status rocblas_hgemv_strided_batched(rocblas_handle handle, rocblas_operation transA,
rocblas_operation transB, rocblas_int m, rocblas_int n,
rocblas_int k, const rocblas_half *alpha, const rocblas_half
*A, rocblas_int lda, rocblas_stride stride_a, const
rocblas_half *B, rocblas_int ldb, rocblas_stride stride_b,
const rocblas_half *beta, rocblas_half *C, rocblas_int ldc,
rocblas_stride stride_c, rocblas_int batch_count)
```

```
rocblas_status rocblas_cgemv_strided_batched(rocblas_handle handle, rocblas_operation transA,
rocblas_operation transB, rocblas_int m, rocblas_int n,
rocblas_int k, const rocblas_float_complex *alpha, const
rocblas_float_complex *A, rocblas_int lda, rocblas_stride
stride_a, const rocblas_float_complex *B, rocblas_int ldb,
rocblas_stride stride_b, const rocblas_float_complex *beta,
rocblas_float_complex *C, rocblas_int ldc, rocblas_stride
stride_c, rocblas_int batch_count)
```

```
rocblas_status rocblas_zgemv_strided_batched(rocblas_handle handle, rocblas_operation transA,
rocblas_operation transB, rocblas_int m, rocblas_int n,
rocblas_int k, const rocblas_double_complex *alpha, const
rocblas_double_complex *A, rocblas_int lda, rocblas_stride
stride_a, const rocblas_double_complex *B, rocblas_int ldb,
rocblas_stride stride_b, const rocblas_double_complex *beta,
rocblas_double_complex *C, rocblas_int ldc, rocblas_stride
stride_c, rocblas_int batch_count)
```

BLAS Level 3 API

gemv_strided_batched performs one of the strided batched matrix-matrix operations:

```
C_i = alpha*op( A_i )*op( B_i ) + beta*C_i, for i = 1, ..., batch_count,
```

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 strided batched matrices, with
op(A) an m by k by batch_count strided_batched matrix,
op(B) an k by n by batch_count strided_batched matrix and
C an m by n by batch_count strided_batched matrix.

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **transA** – [in] [rocblas_operation] specifies the form of op(A).
- **transB** – [in] [rocblas_operation] specifies the form of op(B).
- **m** – [in] [rocblas_int] matrix dimension m.
- **n** – [in] [rocblas_int] matrix dimension n.
- **k** – [in] [rocblas_int] matrix dimension k.
- **alpha** – [in] device pointer or host pointer specifying the scalar alpha.

- **A** – [in] device pointer pointing to the first matrix A₁.
- **lda** – [in] [rocblas_int] specifies the leading dimension of each A_i.
- **stride_a** – [in] [rocblas_stride] stride from the start of one A_i matrix to the next A_(i + 1).
- **B** – [in] device pointer pointing to the first matrix B₁.
- **ldb** – [in] [rocblas_int] specifies the leading dimension of each B_i.
- **stride_b** – [in] [rocblas_stride] stride from the start of one B_i matrix to the next B_(i + 1).
- **beta** – [in] device pointer or host pointer specifying the scalar beta.
- **C** – [inout] device pointer pointing to the first matrix C₁.
- **ldc** – [in] [rocblas_int] specifies the leading dimension of each C_i.
- **stride_c** – [in] [rocblas_stride] stride from the start of one C_i matrix to the next C_(i + 1).
- **batch_count** – [in] [rocblas_int] number of gemm operations in the batch.

5.6.2 rocblas_Xsymm + batched, strided_batched

rocblas_status **rocblas_ssymm**(*rocblas_handle* handle, *rocblas_side* side, *rocblas_fill* uplo, *rocblas_int* m, *rocblas_int* n, const float *alpha, const float *A, *rocblas_int* lda, const float *B, *rocblas_int* ldb, const float *beta, float *C, *rocblas_int* ldc)

rocblas_status **rocblas_dsymm**(*rocblas_handle* handle, *rocblas_side* side, *rocblas_fill* uplo, *rocblas_int* m, *rocblas_int* n, const double *alpha, const double *A, *rocblas_int* lda, const double *B, *rocblas_int* ldb, const double *beta, double *C, *rocblas_int* ldc)

rocblas_status **rocblas_csymm**(*rocblas_handle* handle, *rocblas_side* side, *rocblas_fill* uplo, *rocblas_int* m, *rocblas_int* n, const *rocblas_float_complex* *alpha, const *rocblas_float_complex* *A, *rocblas_int* lda, const *rocblas_float_complex* *B, *rocblas_int* ldb, const *rocblas_float_complex* *beta, *rocblas_float_complex* *C, *rocblas_int* ldc)

rocblas_status **rocblas_zsymm**(*rocblas_handle* handle, *rocblas_side* side, *rocblas_fill* uplo, *rocblas_int* m, *rocblas_int* n, const *rocblas_double_complex* *alpha, const *rocblas_double_complex* *A, *rocblas_int* lda, const *rocblas_double_complex* *B, *rocblas_int* ldb, const *rocblas_double_complex* *beta, *rocblas_double_complex* *C, *rocblas_int* ldc)

BLAS Level 3 API

symm performs one of the matrix-matrix operations:

```
C := alpha*A*B + beta*C if side == rocblas_side_left,
C := alpha*B*A + beta*C if side == rocblas_side_right,
```

where alpha and beta are scalars, B and C are m by n matrices, and A is a symmetric matrix stored as either upper or lower.

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.

- **side** – [in] [rocblas_side]
 - rocblas_side_left: $C := \alpha * A * B + \beta * C$
 - rocblas_side_right: $C := \alpha * B * A + \beta * C$
- **uplo** – [in] [rocblas_fill]
 - rocblas_fill_upper: A is an upper triangular matrix
 - rocblas_fill_lower: A is a lower triangular matrix
- **m** – [in] [rocblas_int] m specifies the number of rows of B and C. $m \geq 0$.
- **n** – [in] [rocblas_int] n specifies the number of columns of B and C. $n \geq 0$.
- **alpha** – [in] alpha specifies the scalar alpha. When alpha is zero then A and B are not referenced.
- **A** – [in] pointer storing matrix A on the GPU.
 - A is m by m if side == rocblas_side_left
 - A is n by n if side == rocblas_side_right only the upper/lower triangular part is accessed.
- **lda** – [in] [rocblas_int] lda specifies the first dimension of A.

```
if side == rocblas_side_left, lda >= max( 1, m ),
otherwise lda >= max( 1, n ).
```

- **B** – [in] pointer storing matrix B on the GPU. Matrix dimension is m by n
- **ldb** – [in] [rocblas_int] ldb specifies the first dimension of B. $ldb \geq \max(1, m)$.
- **beta** – [in] beta specifies the scalar beta. When beta is zero then C need not be set before entry.
- **C** – [in] pointer storing matrix C on the GPU. Matrix dimension is m by n
- **ldc** – [in] [rocblas_int] ldc specifies the first dimension of C. $ldc \geq \max(1, m)$.

```
rocblas_status rocblas_ssymm_batched(rocblas_handle handle, rocblas_side side, rocblas_fill uplo, rocblas_int
m, rocblas_int n, const float *alpha, const float *const A[], rocblas_int
lda, const float *const B[], rocblas_int ldb, const float *beta, float *const
C[], rocblas_int ldc, rocblas_int batch_count)
```

```
rocblas_status rocblas_dsymm_batched(rocblas_handle handle, rocblas_side side, rocblas_fill uplo, rocblas_int
m, rocblas_int n, const double *alpha, const double *const A[],
rocblas_int lda, const double *const B[], rocblas_int ldb, const double
*beta, double *const C[], rocblas_int ldc, rocblas_int batch_count)
```

```
rocblas_status rocblas_csymm_batched(rocblas_handle handle, rocblas_side side, rocblas_fill uplo, rocblas_int
m, rocblas_int n, const rocblas_float_complex *alpha, const
rocblas_float_complex *const A[], rocblas_int lda, const
rocblas_float_complex *const B[], rocblas_int ldb, const
rocblas_float_complex *beta, rocblas_float_complex *const C[],
rocblas_int ldc, rocblas_int batch_count)
```

```
rocblas_status rocblas_zsymm_batched(rocblas_handle handle, rocblas_side side, rocblas_fill uplo, rocblas_int
m, rocblas_int n, const rocblas_double_complex *alpha, const
rocblas_double_complex *const A[], rocblas_int lda, const
rocblas_double_complex *const B[], rocblas_int ldb, const
rocblas_double_complex *beta, rocblas_double_complex *const C[],
rocblas_int ldc, rocblas_int batch_count)
```

BLAS Level 3 API

`symm_batched` performs a batch of the matrix-matrix operations:

```
C_i := alpha*A_i*B_i + beta*C_i if side == rocblas_side_left,
C_i := alpha*B_i*A_i + beta*C_i if side == rocblas_side_right,
```

where `alpha` and `beta` are scalars, `B_i` and `C_i` are `m` by `n` matrices, and `A_i` is a symmetric matrix stored as either upper or lower.

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **side** – [in] [rocblas_side]
 - `rocblas_side_left`: $C_i := \alpha A_i B_i + \beta C_i$
 - `rocblas_side_right`: $C_i := \alpha B_i A_i + \beta C_i$
- **uplo** – [in] [rocblas_fill]
 - `rocblas_fill_upper`: `A_i` is an upper triangular matrix
 - `rocblas_fill_lower`: `A_i` is a lower triangular matrix
- **m** – [in] [rocblas_int] `m` specifies the number of rows of `B_i` and `C_i`. $m \geq 0$.
- **n** – [in] [rocblas_int] `n` specifies the number of columns of `B_i` and `C_i`. $n \geq 0$.
- **alpha** – [in] `alpha` specifies the scalar `alpha`. When `alpha` is zero then `A_i` and `B_i` are not referenced.
- **A** – [in] device array of device pointers storing each matrix `A_i` on the GPU.
 - `A_i` is `m` by `m` if `side == rocblas_side_left`
 - `A_i` is `n` by `n` if `side == rocblas_side_right` only the upper/lower triangular part is accessed.
- **lda** – [in] [rocblas_int] `lda` specifies the first dimension of `A_i`.


```
if side == rocblas_side_left, lda >= max( 1, m ),
otherwise lda >= max( 1, n ).
```
- **B** – [in] device array of device pointers storing each matrix `B_i` on the GPU. Matrix dimension is `m` by `n`
- **ldb** – [in] [rocblas_int] `ldb` specifies the first dimension of `B_i`. $ldb \geq \max(1, m)$.
- **beta** – [in] `beta` specifies the scalar `beta`. When `beta` is zero then `C_i` need not be set before entry.
- **C** – [in] device array of device pointers storing each matrix `C_i` on the GPU. Matrix dimension is `m` by `n`.
- **ldc** – [in] [rocblas_int] `ldc` specifies the first dimension of `C_i`. $ldc \geq \max(1, m)$.
- **batch_count** – [in] [rocblas_int] number of instances in the batch.

```
rocblas_status rocblas_ssymm_strided_batched(rocblas_handle handle, rocblas_side side, rocblas_fill uplo,
rocblas_int m, rocblas_int n, const float *alpha, const float
*A, rocblas_int lda, rocblas_stride stride_A, const float *B,
rocblas_int ldb, rocblas_stride stride_B, const float *beta,
float *C, rocblas_int ldc, rocblas_stride stride_C, rocblas_int
batch_count)
```

```
rocblas_status rocblas_dsymm_strided_batched(rocblas_handle handle, rocblas_side side, rocblas_fill uplo,
rocblas_int m, rocblas_int n, const double *alpha, const
double *A, rocblas_int lda, rocblas_stride stride_A, const
double *B, rocblas_int ldb, rocblas_stride stride_B, const
double *beta, double *C, rocblas_int ldc, rocblas_stride
stride_C, rocblas_int batch_count)
```

```
rocblas_status rocblas_csymm_strided_batched(rocblas_handle handle, rocblas_side side, rocblas_fill uplo,
rocblas_int m, rocblas_int n, const rocblas_float_complex
*alpha, const rocblas_float_complex *A, rocblas_int lda,
rocblas_stride stride_A, const rocblas_float_complex *B,
rocblas_int ldb, rocblas_stride stride_B, const
rocblas_float_complex *beta, rocblas_float_complex *C,
rocblas_int ldc, rocblas_stride stride_C, rocblas_int
batch_count)
```

```
rocblas_status rocblas_zsymm_strided_batched(rocblas_handle handle, rocblas_side side, rocblas_fill uplo,
rocblas_int m, rocblas_int n, const rocblas_double_complex
*alpha, const rocblas_double_complex *A, rocblas_int lda,
rocblas_stride stride_A, const rocblas_double_complex *B,
rocblas_int ldb, rocblas_stride stride_B, const
rocblas_double_complex *beta, rocblas_double_complex *C,
rocblas_int ldc, rocblas_stride stride_C, rocblas_int
batch_count)
```

BLAS Level 3 API

symm_strided_batched performs a batch of the matrix-matrix operations:

```
C_i := alpha*A_i*B_i + beta*C_i if side == rocblas_side_left,
C_i := alpha*B_i*A_i + beta*C_i if side == rocblas_side_right,
```

where alpha **and** beta are scalars, B_i **and** C_i are m by n matrices, **and** A_i **is** a symmetric matrix stored **as** either upper **or** lower.

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **side** – [in] [rocblas_side]
 - rocblas_side_left: $C_i := \alpha A_i B_i + \beta C_i$
 - rocblas_side_right: $C_i := \alpha B_i A_i + \beta C_i$
- **uplo** – [in] [rocblas_fill]
 - rocblas_fill_upper: A_i is an upper triangular matrix
 - rocblas_fill_lower: A_i is a lower triangular matrix
- **m** – [in] [rocblas_int] m specifies the number of rows of B_i and C_i. $m \geq 0$.

- **n** – [in] [rocblas_int] n specifies the number of columns of B_i and C_i. $n \geq 0$.
- **alpha** – [in] alpha specifies the scalar alpha. When alpha is zero then A_i and B_i are not referenced.
- **A** – [in] device pointer to first matrix A_1
 - A_i is m by m if side == rocblas_side_left
 - A_i is n by n if side == rocblas_side_right only the upper/lower triangular part is accessed.
- **lda** – [in] [rocblas_int] lda specifies the first dimension of A_i.

```
if side == rocblas_side_left, lda >= max( 1, m ),
otherwise lda >= max( 1, n ).
```

- **stride_A** – [in] [rocblas_stride] stride from the start of one matrix (A_i) and the next one (A_{i+1}).
- **B** – [in] device pointer to first matrix B_1 of dimension (ldb, n) on the GPU.
- **ldb** – [in] [rocblas_int] ldb specifies the first dimension of B_i. $ldb \geq \max(1, m)$.
- **stride_B** – [in] [rocblas_stride] stride from the start of one matrix (B_i) and the next one (B_{i+1}).
- **beta** – [in] beta specifies the scalar beta. When beta is zero then C need not be set before entry.
- **C** – [in] device pointer to first matrix C_1 of dimension (ldc, n) on the GPU.
- **ldc** – [in] [rocblas_int] ldc specifies the first dimension of C. $ldc \geq \max(1, m)$.
- **stride_C** – [inout] [rocblas_stride] stride from the start of one matrix (C_i) and the next one (C_{i+1}).
- **batch_count** – [in] [rocblas_int] number of instances in the batch.

5.6.3 rocblas_Xsyrk + batched, strided_batched

rocblas_status **rocblas_ssyrk**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_int* n, *rocblas_int* k, const float *alpha, const float *A, *rocblas_int* lda, const float *beta, float *C, *rocblas_int* ldc)

rocblas_status **rocblas_dsyrk**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_int* n, *rocblas_int* k, const double *alpha, const double *A, *rocblas_int* lda, const double *beta, double *C, *rocblas_int* ldc)

rocblas_status **rocblas_csyrk**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_int* n, *rocblas_int* k, const *rocblas_float_complex* *alpha, const *rocblas_float_complex* *A, *rocblas_int* lda, const *rocblas_float_complex* *beta, *rocblas_float_complex* *C, *rocblas_int* ldc)

rocblas_status **rocblas_zsyrk**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_int* n, *rocblas_int* k, const *rocblas_double_complex* *alpha, const *rocblas_double_complex* *A, *rocblas_int* lda, const *rocblas_double_complex* *beta, *rocblas_double_complex* *C, *rocblas_int* ldc)

BLAS Level 3 API

syrk performs one of the matrix-matrix operations for a symmetric rank-k update:

```

C := alpha*op( A )*op( A )^T + beta*C,

where alpha and beta are scalars, op(A) is an n by k matrix, and
C is a symmetric n x n matrix stored as either upper or lower.

op( A ) = A, and A is n by k if transA == rocblas_operation_none
op( A ) = A^T and A is k by n if transA == rocblas_operation_transpose

```

rocblas_operation_conjugate_transpose is not supported for complex types. See cherk and zherk.

if transA = rocblas_operation_none, lda >= max(1, n), otherwise lda >= max(1, k).

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill]
 - rocblas_fill_upper: C is an upper triangular matrix
 - rocblas_fill_lower: C is a lower triangular matrix
- **transA** – [in] [rocblas_operation]
 - rocblas_operation_transpose: $\text{op}(A) = A^T$
 - rocblas_operation_none: $\text{op}(A) = A$
 - rocblas_operation_conjugate_transpose: $\text{op}(A) = A^T$
- **n** – [in] [rocblas_int] n specifies the number of rows and columns of C. $n \geq 0$.
- **k** – [in] [rocblas_int] k specifies the number of columns of op(A). $k \geq 0$.
- **alpha** – [in] alpha specifies the scalar alpha. When alpha is zero then A is not referenced and A need not be set before entry.
- **A** – [in] pointer storing matrix A on the GPU. Matrix dimension is (lda, k) when if transA = rocblas_operation_none, otherwise (lda, n)
- **lda** – [in] [rocblas_int] lda specifies the first dimension of A.
- **beta** – [in] beta specifies the scalar beta. When beta is zero then C need not be set before entry.
- **C** – [in] pointer storing matrix C on the GPU. only the upper/lower triangular part is accessed.
- **ldc** – [in] [rocblas_int] ldc specifies the first dimension of C. $\text{ldc} \geq \max(1, n)$.

rocblas_status **rocblas_ssyrrk_batched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_int* n, *rocblas_int* k, const float *alpha, const float *const A[], *rocblas_int* lda, const float *beta, float *const C[], *rocblas_int* ldc, *rocblas_int* batch_count)

rocblas_status **rocblas_dsyrrk_batched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_int* n, *rocblas_int* k, const double *alpha, const double *const A[], *rocblas_int* lda, const double *beta, double *const C[], *rocblas_int* ldc, *rocblas_int* batch_count)


```
rocblas_status rocblas_csyrk_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_operation transA,
                                     rocblas_int n, rocblas_int k, const rocblas_float_complex *alpha, const
                                     rocblas_float_complex *const A[], rocblas_int lda, const
                                     rocblas_float_complex *beta, rocblas_float_complex *const C[],
                                     rocblas_int ldc, rocblas_int batch_count)
```

```
rocblas_status rocblas_zsyrk_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_operation transA,
                                     rocblas_int n, rocblas_int k, const rocblas_double_complex *alpha,
                                     const rocblas_double_complex *const A[], rocblas_int lda, const
                                     rocblas_double_complex *beta, rocblas_double_complex *const C[],
                                     rocblas_int ldc, rocblas_int batch_count)
```

BLAS Level 3 API

syrk_batched performs a batch of the matrix-matrix operations for a symmetric rank-k update:

```
C_i := alpha*op( A_i )*op( A_i )^T + beta*C_i,

where alpha and beta are scalars, op(A_i) is an n by k matrix, and
C_i is a symmetric n x n matrix stored as either upper or lower.

op( A_i ) = A_i, and A_i is n by k if transA == rocblas_operation_none
op( A_i ) = A_i^T and A_i is k by n if transA == rocblas_operation_transpose
```

rocblas_operation_conjugate_transpose is not supported for complex types. See cherk and zherk.

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill]
 - rocblas_fill_upper: C_i is an upper triangular matrix
 - rocblas_fill_lower: C_i is a lower triangular matrix
- **transA** – [in] [rocblas_operation]
 - rocblas_operation_transpose: $\text{op}(A) = A^T$
 - rocblas_operation_none: $\text{op}(A) = A$
 - rocblas_operation_conjugate_transpose: $\text{op}(A) = A^H$
- **n** – [in] [rocblas_int] n specifies the number of rows and columns of C_i. $n \geq 0$.
- **k** – [in] [rocblas_int] k specifies the number of columns of op(A). $k \geq 0$.
- **alpha** – [in] alpha specifies the scalar alpha. When alpha is zero then A is not referenced and A need not be set before entry.
- **A** – [in] device array of device pointers storing each matrix_i A of dimension (lda, k) when transA is rocblas_operation_none, otherwise of dimension (lda, n).
- **lda** – [in] [rocblas_int] lda specifies the first dimension of A_i.

```
if transA == rocblas_operation_none, lda >= max( 1, n ),
otherwise lda >= max( 1, k ).
```

- **beta** – [in] beta specifies the scalar beta. When beta is zero then C need not be set before entry.

- **C** – [in] device array of device pointers storing each matrix C_i on the GPU. only the upper/lower triangular part of each C_i is accessed.
- **ldc** – [in] [rocblas_int] ldc specifies the first dimension of C. $ldc \geq \max(1, n)$.
- **batch_count** – [in] [rocblas_int] number of instances in the batch.

rocblas_status **rocblas_ssyrk_strided_batched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_int* n, *rocblas_int* k, const float *alpha, const float *A, *rocblas_int* lda, *rocblas_stride* stride_A, const float *beta, float *C, *rocblas_int* ldc, *rocblas_stride* stride_C, *rocblas_int* batch_count)

rocblas_status **rocblas_dsyrk_strided_batched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_int* n, *rocblas_int* k, const double *alpha, const double *A, *rocblas_int* lda, *rocblas_stride* stride_A, const double *beta, double *C, *rocblas_int* ldc, *rocblas_stride* stride_C, *rocblas_int* batch_count)

rocblas_status **rocblas_csyrk_strided_batched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_int* n, *rocblas_int* k, const *rocblas_float_complex* *alpha, const *rocblas_float_complex* *A, *rocblas_int* lda, *rocblas_stride* stride_A, const *rocblas_float_complex* *beta, *rocblas_float_complex* *C, *rocblas_int* ldc, *rocblas_stride* stride_C, *rocblas_int* batch_count)

rocblas_status **rocblas_zsyrk_strided_batched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_int* n, *rocblas_int* k, const *rocblas_double_complex* *alpha, const *rocblas_double_complex* *A, *rocblas_int* lda, *rocblas_stride* stride_A, const *rocblas_double_complex* *beta, *rocblas_double_complex* *C, *rocblas_int* ldc, *rocblas_stride* stride_C, *rocblas_int* batch_count)

BLAS Level 3 API

`syrk_strided_batched` performs a batch of the matrix-matrix operations for a symmetric rank-k update:

```
C_i := alpha*op( A_i )*op( A_i )^T + beta*C_i,
```

where `alpha` and `beta` are scalars, `op(A_i)` is an n by k matrix, and C_i is a symmetric $n \times n$ matrix stored as either upper or lower.

```
op( A_i ) = A_i, and A_i is n by k if transA == rocblas_operation_none
op( A_i ) = A_i^T and A_i is k by n if transA == rocblas_operation_transpose
```

`rocblas_operation_conjugate_transpose` is not supported for complex types. See `cherk` and `zherk`.

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill]
 - `rocblas_fill_upper`: C_i is an upper triangular matrix
 - `rocblas_fill_lower`: C_i is a lower triangular matrix

- **transA** – [in] [rocblas_operation]
 - rocblas_operation_transpose: $op(A) = A^T$
 - rocblas_operation_none: $op(A) = A$
 - rocblas_operation_conjugate_transpose: $op(A) = A^H$
- **n** – [in] [rocblas_int] n specifies the number of rows and columns of C_i. $n \geq 0$.
- **k** – [in] [rocblas_int] k specifies the number of columns of op(A). $k \geq 0$.
- **alpha** – [in] alpha specifies the scalar alpha. When alpha is zero then A is not referenced and A need not be set before entry.
- **A** – [in] Device pointer to the first matrix A_1 on the GPU of dimension (lda, k) when transA is rocblas_operation_none, otherwise of dimension (lda, n).
- **lda** – [in] [rocblas_int] lda specifies the first dimension of A_i.

```
if transA == rocblas_operation_none, lda >= max( 1, n ),
otherwise lda >= max( 1, k ).
```

- **stride_A** – [in] [rocblas_stride] stride from the start of one matrix (A_i) and the next one (A_i+1).
- **beta** – [in] beta specifies the scalar beta. When beta is zero then C need not be set before entry.
- **C** – [in] Device pointer to the first matrix C_1 on the GPU. on the GPU. only the upper/lower triangular part of each C_i is accessed.
- **ldc** – [in] [rocblas_int] ldc specifies the first dimension of C. $ldc \geq \max(1, n)$.
- **stride_C** – [inout] [rocblas_stride] stride from the start of one matrix (C_i) and the next one (C_i+1)
- **batch_count** – [in] [rocblas_int] number of instances in the batch.

5.6.4 rocblas_Xsyr2k + batched, strided_batched

rocblas_status **rocblas_ssr2k**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* trans, *rocblas_int* n, *rocblas_int* k, const float *alpha, const float *A, *rocblas_int* lda, const float *B, *rocblas_int* ldb, const float *beta, float *C, *rocblas_int* ldc)

rocblas_status **rocblas_dsr2k**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* trans, *rocblas_int* n, *rocblas_int* k, const double *alpha, const double *A, *rocblas_int* lda, const double *B, *rocblas_int* ldb, const double *beta, double *C, *rocblas_int* ldc)

rocblas_status **rocblas_csr2k**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* trans, *rocblas_int* n, *rocblas_int* k, const *rocblas_float_complex* *alpha, const *rocblas_float_complex* *A, *rocblas_int* lda, const *rocblas_float_complex* *B, *rocblas_int* ldb, const *rocblas_float_complex* *beta, *rocblas_float_complex* *C, *rocblas_int* ldc)

rocblas_status **rocblas_zsr2k**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* trans, *rocblas_int* n, *rocblas_int* k, const *rocblas_double_complex* *alpha, const *rocblas_double_complex* *A, *rocblas_int* lda, const *rocblas_double_complex* *B, *rocblas_int* ldb, const *rocblas_double_complex* *beta, *rocblas_double_complex* *C, *rocblas_int* ldc)

BLAS Level 3 API

syr2k performs one of the matrix-matrix operations for a symmetric rank-2k update:

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

where α and β are scalars, $\text{op}(A)$ and $\text{op}(B)$ are n by k matrix, and C is a symmetric $n \times n$ matrix stored as either upper or lower.

$\text{op}(A) = A$, $\text{op}(B) = B$, and A and B are n by k if $\text{trans} == \text{rocblas_operation_none}$
 $\text{op}(A) = A^T$, $\text{op}(B) = B^T$, and A and B are k by n if $\text{trans} == \text{rocblas_operation_}\rightarrow\text{transpose}$
 or for ssyr2k and dsyr2k when $\text{trans} == \text{rocblas_operation_conjugate_transpose}$

`rocblas_operation_conjugate_transpose` is not supported for complex types in `csyr2k` and `zsyr2k`.

if $\text{trans} = \text{rocblas_operation_none}$, $\text{lda} \geq \max(1, n)$, otherwise $\text{lda} \geq \max(1, k)$.

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill]
 - `rocblas_fill_upper`: C is an upper triangular matrix
 - `rocblas_fill_lower`: C is a lower triangular matrix
- **trans** – [in] [rocblas_operation]
 - `rocblas_operation_transpose`: $\text{op}(A) = A^T$, $\text{op}(B) = B^T$
 - `rocblas_operation_none`: $\text{op}(A) = A$, $\text{op}(B) = B$
 - `rocblas_operation_conjugate_transpose`: $\text{op}(A) = A^T$, $\text{op}(B) = B^T$
- **n** – [in] [rocblas_int] n specifies the number of rows and columns of C . $n \geq 0$.
- **k** – [in] [rocblas_int] k specifies the number of columns of $\text{op}(A)$ and $\text{op}(B)$. $k \geq 0$.
- **alpha** – [in] α specifies the scalar α . When α is zero then A is not referenced and A need not be set before entry.
- **A** – [in] pointer storing matrix A on the GPU. Matrix dimension is (lda, k) when if $\text{trans} = \text{rocblas_operation_none}$, otherwise (lda, n) only the upper/lower triangular part is accessed.
- **lda** – [in] [rocblas_int] lda specifies the first dimension of A .
- **B** – [in] pointer storing matrix B on the GPU. Matrix dimension is (ldb, k) when if $\text{trans} = \text{rocblas_operation_none}$, otherwise (ldb, n) only the upper/lower triangular part is accessed.
- **ldb** – [in] [rocblas_int] ldb specifies the first dimension of B . if $\text{trans} = \text{rocblas_operation_none}$, $\text{ldb} \geq \max(1, n)$, otherwise $\text{ldb} \geq \max(1, k)$.
- **beta** – [in] β specifies the scalar β . When β is zero then C need not be set before entry.
- **C** – [in] pointer storing matrix C on the GPU.
- **ldc** – [in] [rocblas_int] ldc specifies the first dimension of C . $\text{ldc} \geq \max(1, n)$.

```
rocblas_status rocblas_ssy2k_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_operation trans,
                                     rocblas_int n, rocblas_int k, const float *alpha, const float *const A[],
                                     rocblas_int lda, const float *const B[], rocblas_int ldb, const float
                                     *beta, float *const C[], rocblas_int ldc, rocblas_int batch_count)
```

```
rocblas_status rocblas_dsyr2k_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_operation trans,
                                       rocblas_int n, rocblas_int k, const double *alpha, const double *const
                                       A[], rocblas_int lda, const double *const B[], rocblas_int ldb, const
                                       double *beta, double *const C[], rocblas_int ldc, rocblas_int
                                       batch_count)
```

```
rocblas_status rocblas_csyr2k_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_operation trans,
                                       rocblas_int n, rocblas_int k, const rocblas_float_complex *alpha, const
                                       rocblas_float_complex *const A[], rocblas_int lda, const
                                       rocblas_float_complex *const B[], rocblas_int ldb, const
                                       rocblas_float_complex *beta, rocblas_float_complex *const C[],
                                       rocblas_int ldc, rocblas_int batch_count)
```

```
rocblas_status rocblas_zsyr2k_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_operation trans,
                                       rocblas_int n, rocblas_int k, const rocblas_double_complex *alpha,
                                       const rocblas_double_complex *const A[], rocblas_int lda, const
                                       rocblas_double_complex *const B[], rocblas_int ldb, const
                                       rocblas_double_complex *beta, rocblas_double_complex *const C[],
                                       rocblas_int ldc, rocblas_int batch_count)
```

BLAS Level 3 API

ssyr2k_batched performs a batch of the matrix-matrix operations for a symmetric rank-2k update:

```
C_i := alpha*(op( A_i )*op( B_i )^T + op( B_i )*op( A_i )^T) + beta*C_i,

where alpha and beta are scalars, op(A_i) and op(B_i) are n by k matrix, and
C_i is a symmetric n x n matrix stored as either upper or lower.

op( A_i ) = A_i, op( B_i ) = B_i, and A_i and B_i are n by k if trans == rocblas_
↪operation_none
op( A_i ) = A_i^T, op( B_i ) = B_i^T, and A_i and B_i are k by n if trans ==
↪rocblas_operation_transpose
or for ssyr2k_batched and dsyr2k_batched when trans == rocblas_operation_conjugate_
↪transpose
```

rocblas_operation_conjugate_transpose is not supported for complex types in csyr2k_batched and zsyr2k_batched.

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill]
 - rocblas_fill_upper: C_i is an upper triangular matrix
 - rocblas_fill_lower: C_i is a lower triangular matrix
- **trans** – [in] [rocblas_operation]
 - rocblas_operation_transpose: op(A_i) = A_i^T, op(B_i) = B_i^T

- `rocblas_operation_none`: $\text{op}(A_i) = A_i$, $\text{op}(B_i) = B_i$
- `rocblas_operation_conjugate_transpose`: $\text{op}(A_i) = A_i^T$, $\text{op}(B_i) = B_i^T$
- **n** – [in] [rocblas_int] n specifies the number of rows and columns of C_i . $n \geq 0$.
- **k** – [in] [rocblas_int] k specifies the number of columns of $\text{op}(A)$. $k \geq 0$.
- **alpha** – [in] alpha specifies the scalar alpha. When alpha is zero then A is not referenced and A need not be set before entry.
- **A** – [in] device array of device pointers storing each matrix A_i of dimension (lda, k) when trans is `rocblas_operation_none`, otherwise of dimension (lda, n).
- **lda** – [in] [rocblas_int] lda specifies the first dimension of A_i . if trans = `rocblas_operation_none`, $\text{lda} \geq \max(1, n)$, otherwise $\text{lda} \geq \max(1, k)$.
- **B** – [in] device array of device pointers storing each matrix B_i of dimension (ldb, k) when trans is `rocblas_operation_none`, otherwise of dimension (ldb, n).
- **ldb** – [in] [rocblas_int] ldb specifies the first dimension of B.

```
if trans == rocblas_operation_none,  ldb >= max( 1, n ),
otherwise ldb >= max( 1, k ).
```

- **beta** – [in] beta specifies the scalar beta. When beta is zero then C need not be set before entry.
- **C** – [in] device array of device pointers storing each matrix C_i on the GPU.
- **ldc** – [in] [rocblas_int] ldc specifies the first dimension of C. $\text{ldc} \geq \max(1, n)$.
- **batch_count** – [in] [rocblas_int] number of instances in the batch.

rocblas_status **rocblas_ssyr2k_strided_batched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* trans, *rocblas_int* n, *rocblas_int* k, const float *alpha, const float *A, *rocblas_int* lda, *rocblas_stride* stride_A, const float *B, *rocblas_int* ldb, *rocblas_stride* stride_B, const float *beta, float *C, *rocblas_int* ldc, *rocblas_stride* stride_C, *rocblas_int* batch_count)

rocblas_status **rocblas_dsyr2k_strided_batched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* trans, *rocblas_int* n, *rocblas_int* k, const double *alpha, const double *A, *rocblas_int* lda, *rocblas_stride* stride_A, const double *B, *rocblas_int* ldb, *rocblas_stride* stride_B, const double *beta, double *C, *rocblas_int* ldc, *rocblas_stride* stride_C, *rocblas_int* batch_count)

rocblas_status **rocblas_csyr2k_strided_batched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* trans, *rocblas_int* n, *rocblas_int* k, const *rocblas_float_complex* *alpha, const *rocblas_float_complex* *A, *rocblas_int* lda, *rocblas_stride* stride_A, const *rocblas_float_complex* *B, *rocblas_int* ldb, *rocblas_stride* stride_B, const *rocblas_float_complex* *beta, *rocblas_float_complex* *C, *rocblas_int* ldc, *rocblas_stride* stride_C, *rocblas_int* batch_count)

```

rocblas_status rocblas_zsyr2k_strided_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_operation
trans, rocblas_int n, rocblas_int k, const
rocblas_double_complex *alpha, const
rocblas_double_complex *A, rocblas_int lda, rocblas_stride
stride_A, const rocblas_double_complex *B, rocblas_int ldb,
rocblas_stride stride_B, const rocblas_double_complex
*beta, rocblas_double_complex *C, rocblas_int ldc,
rocblas_stride stride_C, rocblas_int batch_count)

```

BLAS Level 3 API

syr2k_strided_batched performs a batch of the matrix-matrix operations for a symmetric rank-2k update:

```

C_i := alpha*(op( A_i )*op( B_i )^T + op( B_i )*op( A_i )^T) + beta*C_i,

where alpha and beta are scalars, op(A_i) and op(B_i) are n by k matrix, and
C_i is a symmetric n x n matrix stored as either upper or lower.

op( A_i ) = A_i, op( B_i ) = B_i, and A_i and B_i are n by k if trans == rocblas_
↪operation_none
op( A_i ) = A_i^T, op( B_i ) = B_i^T, and A_i and B_i are k by n if trans ==
↪rocblas_operation_transpose
or for ssyr2k_strided_batched and dsyr2k_strided_batched when trans == rocblas_
↪operation_conjugate_transpose

```

rocblas_operation_conjugate_transpose is not supported for complex types in csyr2k_strided_batched and zsyr2k_strided_batched.

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill]
 - rocblas_fill_upper: C_i is an upper triangular matrix
 - rocblas_fill_lower: C_i is a lower triangular matrix
- **trans** – [in] [rocblas_operation]
 - rocblas_operation_transpose: op(A_i) = A_i^T, op(B_i) = B_i^T
 - rocblas_operation_none: op(A_i) = A_i, op(B_i) = B_i
 - rocblas_operation_conjugate_transpose: op(A_i) = A_i^T, op(B_i) = B_i^T
- **n** – [in] [rocblas_int] n specifies the number of rows and columns of C_i. n >= 0.
- **k** – [in] [rocblas_int] k specifies the number of columns of op(A). k >= 0.
- **alpha** – [in] alpha specifies the scalar alpha. When alpha is zero then A is not referenced and A need not be set before entry.
- **A** – [in] Device pointer to the first matrix A_1 on the GPU of dimension (lda, k) when trans is rocblas_operation_none, otherwise of dimension (lda, n).
- **lda** – [in] [rocblas_int] lda specifies the first dimension of A_i.

```

if trans == rocblas_operation_none, lda >= max( 1, n ),
otherwise lda >= max( 1, k ).

```

- **stride_A** – [in] [rocblas_stride] stride from the start of one matrix (A_i) and the next one (A_i+1)
- **B** – [in] Device pointer to the first matrix B_1 on the GPU of dimension (ldb, k) when trans is rocblas_operation_none, otherwise of dimension (ldb, n)
- **ldb** – [in] [rocblas_int] ldb specifies the first dimension of B_i.

```
if trans == rocblas_operation_none,  ldb >= max( 1, n ),
otherwise ldb >= max( 1, k ).
```

- **stride_B** – [in] [rocblas_stride] stride from the start of one matrix (B_i) and the next one (B_i+1)
- **beta** – [in] beta specifies the scalar beta. When beta is zero then C need not be set before entry.
- **C** – [in] Device pointer to the first matrix C_1 on the GPU.
- **ldc** – [in] [rocblas_int] ldc specifies the first dimension of C. ldc >= max(1, n).
- **stride_C** – [inout] [rocblas_stride] stride from the start of one matrix (C_i) and the next one (C_i+1).
- **batch_count** – [in] [rocblas_int] number of instances in the batch.

5.6.5 rocblas_Xsyrkx + batched, strided_batched

rocblas_status **rocblas_ssyrkx**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* trans, *rocblas_int* n, *rocblas_int* k, const float *alpha, const float *A, *rocblas_int* lda, const float *B, *rocblas_int* ldb, const float *beta, float *C, *rocblas_int* ldc)

rocblas_status **rocblas_dsyrkx**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* trans, *rocblas_int* n, *rocblas_int* k, const double *alpha, const double *A, *rocblas_int* lda, const double *B, *rocblas_int* ldb, const double *beta, double *C, *rocblas_int* ldc)

rocblas_status **rocblas_csyrkx**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* trans, *rocblas_int* n, *rocblas_int* k, const *rocblas_float_complex* *alpha, const *rocblas_float_complex* *A, *rocblas_int* lda, const *rocblas_float_complex* *B, *rocblas_int* ldb, const *rocblas_float_complex* *beta, *rocblas_float_complex* *C, *rocblas_int* ldc)

rocblas_status **rocblas_zsyrkx**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* trans, *rocblas_int* n, *rocblas_int* k, const *rocblas_double_complex* *alpha, const *rocblas_double_complex* *A, *rocblas_int* lda, const *rocblas_double_complex* *B, *rocblas_int* ldb, const *rocblas_double_complex* *beta, *rocblas_double_complex* *C, *rocblas_int* ldc)

BLAS Level 3 API

syrkx performs one of the matrix-matrix operations for a symmetric rank-k update:

```
C := alpha*op( A )*op( B )^T + beta*C,
```

where alpha and beta are scalars, op(A) and op(B) are n by k matrix, and C is a symmetric n x n matrix stored as either upper or lower.

This routine should only be used when the caller can guarantee that the result of op(A)*op(B)^T will be symmetric.


```

op( A ) = A, op( B ) = B, and A and B are n by k if trans == rocblas_operation_none
op( A ) = A^T, op( B ) = B^T, and A and B are k by n if trans == rocblas_operation_
↪ transpose
or for ssyrkx and dsyrkx when trans == rocblas_operation_conjugate_transpose

```

rocblas_operation_conjugate_transpose is not supported for complex types in csyrkx and zsyrkx.

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill]
 - rocblas_fill_upper: C is an upper triangular matrix
 - rocblas_fill_lower: C is a lower triangular matrix
- **trans** – [in] [rocblas_operation]
 - rocblas_operation_transpose: $\text{op}(A) = A^T$, $\text{op}(B) = B^T$
 - rocblas_operation_none: $\text{op}(A) = A$, $\text{op}(B) = B$
 - rocblas_operation_conjugate_transpose: $\text{op}(A) = A^T$, $\text{op}(B) = B^T$
- **n** – [in] [rocblas_int] n specifies the number of rows and columns of C. $n \geq 0$.
- **k** – [in] [rocblas_int] k specifies the number of columns of op(A) and op(B). $k \geq 0$.
- **alpha** – [in] alpha specifies the scalar alpha. When alpha is zero then A is not referenced and A need not be set before entry.
- **A** – [in] pointer storing matrix A on the GPU. Matrix dimension is (lda, k) when if trans = rocblas_operation_none, otherwise (lda, n)
- **lda** – [in] [rocblas_int] lda specifies the first dimension of A.

```

if trans == rocblas_operation_none, lda >= max( 1, n ),
otherwise lda >= max( 1, k ).

```

- **B** – [in] pointer storing matrix B on the GPU. Matrix dimension is (ldb, k) when if trans = rocblas_operation_none, otherwise (ldb, n)
- **ldb** – [in] [rocblas_int] ldb specifies the first dimension of B.

```

if trans == rocblas_operation_none, ldb >= max( 1, n ),
otherwise ldb >= max( 1, k ).

```

- **beta** – [in] beta specifies the scalar beta. When beta is zero then C need not be set before entry.
- **C** – [in] pointer storing matrix C on the GPU. only the upper/lower triangular part is accessed.
- **ldc** – [in] [rocblas_int] ldc specifies the first dimension of C. $\text{ldc} \geq \max(1, n)$.

rocblas_status **rocblas_ssyrkx_batched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* trans, *rocblas_int* n, *rocblas_int* k, const float *alpha, const float *const A[], *rocblas_int* lda, const float *const B[], *rocblas_int* ldb, const float *beta, float *const C[], *rocblas_int* ldc, *rocblas_int* batch_count)

```
rocblas_status rocblas_dsyrrkx_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_operation trans,
                                       rocblas_int n, rocblas_int k, const double *alpha, const double *const
                                       A[], rocblas_int lda, const double *const B[], rocblas_int ldb, const
                                       double *beta, double *const C[], rocblas_int ldc, rocblas_int
                                       batch_count)
```

```
rocblas_status rocblas_csyrrkx_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_operation trans,
                                       rocblas_int n, rocblas_int k, const rocblas_float_complex *alpha, const
                                       rocblas_float_complex *const A[], rocblas_int lda, const
                                       rocblas_float_complex *const B[], rocblas_int ldb, const
                                       rocblas_float_complex *beta, rocblas_float_complex *const C[],
                                       rocblas_int ldc, rocblas_int batch_count)
```

```
rocblas_status rocblas_zsyrrkx_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_operation trans,
                                       rocblas_int n, rocblas_int k, const rocblas_double_complex *alpha,
                                       const rocblas_double_complex *const A[], rocblas_int lda, const
                                       rocblas_double_complex *const B[], rocblas_int ldb, const
                                       rocblas_double_complex *beta, rocblas_double_complex *const C[],
                                       rocblas_int ldc, rocblas_int batch_count)
```

BLAS Level 3 API

syrrkx_batched performs a batch of the matrix-matrix operations for a symmetric rank-k update:

$$C_i := \alpha * \text{op}(A_i) * \text{op}(B_i)^T + \beta * C_i,$$

where α and β are scalars, $\text{op}(A_i)$ and $\text{op}(B_i)$ are n by k matrix, and C_i is a symmetric $n \times n$ matrix stored as either upper or lower.

This routine should only be used when the caller can guarantee that the result of $\text{op}(A_i) * \text{op}(B_i)^T$ will be symmetric.

```
op( A_i ) = A_i, op( B_i ) = B_i, and A_i and B_i are n by k if trans == rocblas_
↳ operation_none
op( A_i ) = A_i^T, op( B_i ) = B_i^T, and A_i and B_i are k by n if trans ==
↳ rocblas_operation_transpose
or for ssyrrkx_batched and dsyrrkx_batched when trans == rocblas_operation_conjugate_
↳ transpose
```

rocblas_operation_conjugate_transpose is not supported for complex types in csyrrkx_batched and zsyrrkx_batched.

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill]
 - rocblas_fill_upper: C_i is an upper triangular matrix
 - rocblas_fill_lower: C_i is a lower triangular matrix
- **trans** – [in] [rocblas_operation]
 - rocblas_operation_transpose: $\text{op}(A_i) = A_i^T$, $\text{op}(B_i) = B_i^T$
 - rocblas_operation_none: $\text{op}(A_i) = A_i$, $\text{op}(B_i) = B_i$
 - rocblas_operation_conjugate_transpose: $\text{op}(A_i) = A_i^T$, $\text{op}(B_i) = B_i^T$

- **n** – [in] [rocblas_int] n specifies the number of rows and columns of C_i. n >= 0.
- **k** – [in] [rocblas_int] k specifies the number of columns of op(A). k >= 0.
- **alpha** – [in] alpha specifies the scalar alpha. When alpha is zero then A is not referenced and A need not be set before entry.
- **A** – [in] device array of device pointers storing each matrix_i A of dimension (lda, k) when trans is rocblas_operation_none, otherwise of dimension (lda, n)
- **lda** – [in] [rocblas_int] lda specifies the first dimension of A_i.

```
if trans == rocblas_operation_none, lda >= max( 1, n ),
otherwise lda >= max( 1, k ).
```

- **B** – [in] device array of device pointers storing each matrix_i B of dimension (ldb, k) when trans is rocblas_operation_none, otherwise of dimension (ldb, n)
- **ldb** – [in] [rocblas_int] ldb specifies the first dimension of B.

```
if trans == rocblas_operation_none, ldb >= max( 1, n ),
otherwise ldb >= max( 1, k ).
```

- **beta** – [in] beta specifies the scalar beta. When beta is zero then C need not be set before entry.
- **C** – [in] device array of device pointers storing each matrix C_i on the GPU. only the upper/lower triangular part of each C_i is accessed.
- **ldc** – [in] [rocblas_int] ldc specifies the first dimension of C. ldc >= max(1, n).
- **batch_count** – [in] [rocblas_int] number of instances in the batch.

rocblas_status **rocblas_ssyrrkx_strided_batched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* trans, *rocblas_int* n, *rocblas_int* k, const float *alpha, const float *A, *rocblas_int* lda, *rocblas_stride* stride_A, const float *B, *rocblas_int* ldb, *rocblas_stride* stride_B, const float *beta, float *C, *rocblas_int* ldc, *rocblas_stride* stride_C, *rocblas_int* batch_count)

rocblas_status **rocblas_dsyrkx_strided_batched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* trans, *rocblas_int* n, *rocblas_int* k, const double *alpha, const double *A, *rocblas_int* lda, *rocblas_stride* stride_A, const double *B, *rocblas_int* ldb, *rocblas_stride* stride_B, const double *beta, double *C, *rocblas_int* ldc, *rocblas_stride* stride_C, *rocblas_int* batch_count)

rocblas_status **rocblas_csyrrkx_strided_batched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* trans, *rocblas_int* n, *rocblas_int* k, const *rocblas_float_complex* *alpha, const *rocblas_float_complex* *A, *rocblas_int* lda, *rocblas_stride* stride_A, const *rocblas_float_complex* *B, *rocblas_int* ldb, *rocblas_stride* stride_B, const *rocblas_float_complex* *beta, *rocblas_float_complex* *C, *rocblas_int* ldc, *rocblas_stride* stride_C, *rocblas_int* batch_count)

```

rocblas_status rocblas_zsyrkx_strided_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_operation
trans, rocblas_int n, rocblas_int k, const
rocblas_double_complex *alpha, const
rocblas_double_complex *A, rocblas_int lda, rocblas_stride
stride_A, const rocblas_double_complex *B, rocblas_int ldb,
rocblas_stride stride_B, const rocblas_double_complex
*beta, rocblas_double_complex *C, rocblas_int ldc,
rocblas_stride stride_C, rocblas_int batch_count)

```

BLAS Level 3 API

syrrkx_strided_batched performs a batch of the matrix-matrix operations for a symmetric rank-k update:

```
C_i := alpha*op( A_i )*op( B_i )^T + beta*C_i,
```

where alpha and beta are scalars, op(A_i) and op(B_i) are n by k matrix, and C_i is a symmetric n x n matrix stored as either upper or lower.

This routine should only be used when the caller can guarantee that the result of op(A_i)*op(B_i)^T will be symmetric.

```

op( A_i ) = A_i, op( B_i ) = B_i, and A_i and B_i are n by k if trans == rocblas_
↪operation_none
op( A_i ) = A_i^T, op( B_i ) = B_i^T, and A_i and B_i are k by n if trans ==
↪rocblas_operation_transpose
or for ssyrkx_strided_batched and dsyrkx_strided_batched when trans == rocblas_
↪operation_conjugate_transpose

```

rocblas_operation_conjugate_transpose is not supported for complex types in csyrkx_strided_batched and zsyrkx_strided_batched.

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill]
 - rocblas_fill_upper: C_i is an upper triangular matrix
 - rocblas_fill_lower: C_i is a lower triangular matrix
- **trans** – [in] [rocblas_operation]
 - rocblas_operation_transpose: op(A_i) = A_i^T, op(B_i) = B_i^T
 - rocblas_operation_none: op(A_i) = A_i, op(B_i) = B_i
 - rocblas_operation_conjugate_transpose: op(A_i) = A_i^T, op(B_i) = B_i^T
- **n** – [in] [rocblas_int] n specifies the number of rows and columns of C_i. n >= 0.
- **k** – [in] [rocblas_int] k specifies the number of columns of op(A). k >= 0.
- **alpha** – [in] alpha specifies the scalar alpha. When alpha is zero then A is not referenced and A need not be set before entry.
- **A** – [in] Device pointer to the first matrix A_1 on the GPU of dimension (lda, k) when trans is rocblas_operation_none, otherwise of dimension (lda, n)
- **lda** – [in] [rocblas_int] lda specifies the first dimension of A_i.

```
if trans == rocblas_operation_none, lda >= max( 1, n ),
otherwise lda >= max( 1, k ).
```

- **stride_A** – [in] [rocblas_stride] stride from the start of one matrix (A_i) and the next one (A_i+1).
- **B** – [in] Device pointer to the first matrix B_1 on the GPU of dimension (ldb, k) when trans is rocblas_operation_none, otherwise of dimension (ldb, n).
- **ldb** – [in] [rocblas_int] ldb specifies the first dimension of B_i.

```
if trans == rocblas_operation_none, ldb >= max( 1, n ),
otherwise ldb >= max( 1, k ).
```

- **stride_B** – [in] [rocblas_stride] stride from the start of one matrix (B_i) and the next one (B_i+1).
- **beta** – [in] beta specifies the scalar beta. When beta is zero then C need not be set before entry.
- **C** – [in] Device pointer to the first matrix C_1 on the GPU. only the upper/lower triangular part of each C_i is accessed.
- **ldc** – [in] [rocblas_int] ldc specifies the first dimension of C. ldc >= max(1, n).
- **stride_C** – [inout] [rocblas_stride] stride from the start of one matrix (C_i) and the next one (C_i+1).
- **batch_count** – [in] [rocblas_int] number of instances in the batch.

5.6.6 rocblas_Xtrmm + batched, strided_batched

```
rocblas_status rocblas_strmm(rocblas_handle handle, rocblas_side side, rocblas_fill uplo, rocblas_operation
transA, rocblas_diagonal diag, rocblas_int m, rocblas_int n, const float *alpha,
const float *A, rocblas_int lda, float *B, rocblas_int ldb)
```

```
rocblas_status rocblas_dtrmm(rocblas_handle handle, rocblas_side side, rocblas_fill uplo, rocblas_operation
transA, rocblas_diagonal diag, rocblas_int m, rocblas_int n, const double *alpha,
const double *A, rocblas_int lda, double *B, rocblas_int ldb)
```

```
rocblas_status rocblas_ctrmm(rocblas_handle handle, rocblas_side side, rocblas_fill uplo, rocblas_operation
transA, rocblas_diagonal diag, rocblas_int m, rocblas_int n, const
rocblas_float_complex *alpha, const rocblas_float_complex *A, rocblas_int lda,
rocblas_float_complex *B, rocblas_int ldb)
```

```
rocblas_status rocblas_ztrmm(rocblas_handle handle, rocblas_side side, rocblas_fill uplo, rocblas_operation
transA, rocblas_diagonal diag, rocblas_int m, rocblas_int n, const
rocblas_double_complex *alpha, const rocblas_double_complex *A, rocblas_int
lda, rocblas_double_complex *B, rocblas_int ldb)
```

BLAS Level 3 API

The rocBLAS trmm API is from Legacy BLAS and it supports only in-place functionality. It is deprecated and it will be replaced with an API that supports both in-place and out-of-place functionality. The new API is available in rocBLAS versions 3.x.x and later. To get the new API compile with the directive -DROCBLAS_V3

```

#ifdef ROCBLAS_V3 // deprecated
    rocbblas_status rocbblas_strmm(rocbblas_handle      handle,
                                   rocbblas_side        side,
                                   rocbblas_fill        uplo,
                                   rocbblas_operation    transA,
                                   rocbblas_diagonal    diag,
                                   rocbblas_int         m,
                                   rocbblas_int         n,
                                   const float*        alpha,
                                   const float*        A,
                                   rocbblas_int         lda,
                                   float*             B,
                                   rocbblas_int         ldb)

#else // available in rocBLAS version 3.x.x and later with -DROCBLAS_V3

    rocbblas_status rocbblas_strmm(rocbblas_handle      handle,
                                   rocbblas_side        side,
                                   rocbblas_fill        uplo,
                                   rocbblas_operation    transA,
                                   rocbblas_diagonal    diag,
                                   rocbblas_int         m,
                                   rocbblas_int         n,
                                   const float*        alpha,
                                   const float*        A,
                                   rocbblas_int         lda,
                                   const float*        B,
                                   rocbblas_int         ldb,
                                   float*             C,
                                   rocbblas_int         ldc)

#endif

#ifdef ROCBLAS_V3 // deprecated
    rocbblas_status rocbblas_dtrmm(rocbblas_handle      handle,
                                   rocbblas_side        side,
                                   rocbblas_fill        uplo,
                                   rocbblas_operation    transA,
                                   rocbblas_diagonal    diag,
                                   rocbblas_int         m,
                                   rocbblas_int         n,
                                   const double*       alpha,
                                   const double*       A,
                                   rocbblas_int         lda,
                                   double*             B,
                                   rocbblas_int         ldb)

#else // available in rocBLAS version 3.x.x and later with -DROCBLAS_V3

    rocbblas_status rocbblas_dtrmm(rocbblas_handle      handle,
                                   rocbblas_side        side,
                                   rocbblas_fill        uplo,
                                   rocbblas_operation    transA,
                                   rocbblas_diagonal    diag,

```

(continues on next page)

(continued from previous page)

```

        rocbblas_int          m,
        rocbblas_int          n,
        const double*         alpha,
        const double*         A,
        rocbblas_int          lda,
        const double*         B,
        rocbblas_int          ldb,
        double*                C,
        rocbblas_int          ldc)

#endif

#ifdef ROCBLAS_V3 // deprecated
    rocbblas_status rocbblas_ctrmm(rocbblas_handle handle,
        rocbblas_side side,
        rocbblas_fill uplo,
        rocbblas_operation transA,
        rocbblas_diagonal diag,
        rocbblas_int m,
        rocbblas_int n,
        const rocbblas_float_complex* alpha,
        const rocbblas_float_complex* A,
        rocbblas_int lda,
        rocbblas_float_complex* B,
        rocbblas_int ldb)

#else // available in rocBLAS version 3.x.x and later with -DROCBLAS_V3

    rocbblas_status rocbblas_ctrmm(rocbblas_handle handle,
        rocbblas_side side,
        rocbblas_fill uplo,
        rocbblas_operation transA,
        rocbblas_diagonal diag,
        rocbblas_int m,
        rocbblas_int n,
        const rocbblas_float_complex* alpha,
        const rocbblas_float_complex* A,
        rocbblas_int lda,
        const rocbblas_float_complex* B,
        rocbblas_int ldb,
        rocbblas_float_complex* C,
        rocbblas_int ldc)

#endif

#ifdef ROCBLAS_V3 // deprecated
    rocbblas_status rocbblas_ztrmm(rocbblas_handle handle,
        rocbblas_side side,
        rocbblas_fill uplo,
        rocbblas_operation transA,
        rocbblas_diagonal diag,
        rocbblas_int m,
        rocbblas_int n,
        const rocbblas_double_complex* alpha,

```

(continues on next page)

(continued from previous page)

```

const rocblas_double_complex* A,
rocblas_int lda,
rocblas_double_complex* B,
rocblas_int ldb)

#else // available in rocBLAS version 3.x.x and later with -DROCBLAS_V3

rocblas_status rocblas_ztrmm(rocblas_handle handle,
                             rocblas_side side,
                             rocblas_fill uplo,
                             rocblas_operation transA,
                             rocblas_diagonal diag,
                             rocblas_int m,
                             rocblas_int n,
                             const rocblas_double_complex* alpha,
                             const rocblas_double_complex* A,
                             rocblas_int lda,
                             const rocblas_double_complex* B,
                             rocblas_int ldb,
                             rocblas_double_complex* C,
                             rocblas_int ldc)

#endif

```

The deprecated Legacy BLAS in-place trmm performs one of the matrix-matrix operations:

```

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

```

The new trmm performs one of the matrix-matrix operations:

```

C := alpha*op( A )*B,   or
C := alpha*B*op( A ),

```

The in-place functionality is still available in the new trmm by setting pointer C equal to pointer B, and ldc equal to ldb.

alpha is a scalar, B is an m by n matrix, C 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.

```

When uplo == rocblas_fill_upper the leading k by k upper triangular part of the array A must contain the upper triangular matrix and the strictly lower triangular part of A is not referenced. Here k is m when side == rocblas_side_left and is n when side == rocblas_side_right.

When uplo == rocblas_fill_lower the leading k by k lower triangular part of the array A must contain the lower triangular matrix and the strictly upper triangular part of

(continues on next page)

(continued from previous page)

A is not referenced. Here k is m when side == rocblas_side_left and is n when side == rocblas_side_right.

Note that when diag == rocblas_diagonal_unit the diagonal elements of A are not referenced either, but are assumed to be unity.

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **side** – [in] [rocblas_side] Specifies whether op(A) multiplies B from the left or right as follows:
 - rocblas_side_left: $B := \alpha * \text{op}(A) * B$
 - rocblas_side_right: $B := \alpha * B * \text{op}(A)$
- **uplo** – [in] [rocblas_fill] Specifies whether the matrix A is an upper or lower triangular matrix as follows:
 - rocblas_fill_upper: A is an upper triangular matrix.
 - rocblas_fill_lower: A is a lower triangular matrix.
- **transA** – [in] [rocblas_operation] Specifies the form of op(A) to be used in the matrix multiplication as follows:
 - rocblas_operation_none: $\text{op}(A) = A$
 - rocblas_operation_transpose: $\text{op}(A) = A^T$
 - rocblas_operation_conjugate_transpose: $\text{op}(A) = A^H$
- **diag** – [in] [rocblas_diagonal] Specifies whether or not A is unit triangular as follows:
 - rocblas_diagonal_unit: A is assumed to be unit triangular.
 - rocblas_diagonal_non_unit: A is not assumed to be unit triangular.
- **m** – [in] [rocblas_int] m specifies the number of rows of B. $m \geq 0$.
- **n** – [in] [rocblas_int] n specifies the number of columns of B. $n \geq 0$.
- **alpha** – [in] alpha specifies the scalar alpha. When alpha is zero then A is not referenced and B need not be set before entry.
- **A** – [in] Device pointer to matrix A on the GPU. A has dimension (lda, k), where k is m when side == rocblas_side_left and is n when side == rocblas_side_right.
- **lda** – [in] [rocblas_int] lda specifies the first dimension of A.


```
if side == rocblas_side_left, lda >= max( 1, m ),
if side == rocblas_side_right, lda >= max( 1, n ).
```
- **B** – [inout] Device pointer to the first matrix B_0 on the GPU. On entry, the leading m by n part of the array B must contain the matrix B, and on exit is overwritten by the transformed matrix.
- **ldb** – [in] [rocblas_int] ldb specifies the first dimension of B. $ldb \geq \max(1, m)$.

```

rocblas_status rocblas_strmm_batched(rocblas_handle handle, rocblas_side side, rocblas_fill uplo,
                                     rocblas_operation transA, rocblas_diagonal diag, rocblas_int m,
                                     rocblas_int n, const float *alpha, const float *const A[], rocblas_int lda,
                                     float *const B[], rocblas_int ldb, rocblas_int batch_count)

rocblas_status rocblas_dtrmm_batched(rocblas_handle handle, rocblas_side side, rocblas_fill uplo,
                                     rocblas_operation transA, rocblas_diagonal diag, rocblas_int m,
                                     rocblas_int n, const double *alpha, const double *const A[], rocblas_int
                                     lda, double *const B[], rocblas_int ldb, rocblas_int batch_count)

rocblas_status rocblas_ctrmm_batched(rocblas_handle handle, rocblas_side side, rocblas_fill uplo,
                                     rocblas_operation transA, rocblas_diagonal diag, rocblas_int m,
                                     rocblas_int n, const rocblas_float_complex *alpha, const
                                     rocblas_float_complex *const A[], rocblas_int lda,
                                     rocblas_float_complex *const B[], rocblas_int ldb, rocblas_int
                                     batch_count)

rocblas_status rocblas_ztrmm_batched(rocblas_handle handle, rocblas_side side, rocblas_fill uplo,
                                     rocblas_operation transA, rocblas_diagonal diag, rocblas_int m,
                                     rocblas_int n, const rocblas_double_complex *alpha, const
                                     rocblas_double_complex *const A[], rocblas_int lda,
                                     rocblas_double_complex *const B[], rocblas_int ldb, rocblas_int
                                     batch_count)

```

BLAS Level 3 API

The rocBLAS trmm_batched API is from Legacy BLAS and it supports only in-place functionality. It is deprecated and it will be replaced with an API that supports both in-place and out-of-place functionality. The new API is available in rocBLAS versions 3.x.x and later. To get the new API compile with the directive -DROCBLAS_V3.

```

#ifdef ROCBLAS_V3 // deprecated
    rocblas_status rocblas_strmm_batched(rocblas_handle    handle,
                                         rocblas_side      side,
                                         rocblas_fill      uplo,
                                         rocblas_operation  transA,
                                         rocblas_diagonal   diag,
                                         rocblas_int        m,
                                         rocblas_int        n,
                                         const float*      alpha,
                                         const float* const A[],
                                         rocblas_int        lda,
                                         float* const      B[],
                                         rocblas_int        ldb,
                                         rocblas_int        batch_count)

#else // available in rocBLAS version 3.x.x and later with -DROCBLAS_V3

    rocblas_status rocblas_strmm_batched(rocblas_handle    handle,
                                         rocblas_side      side,
                                         rocblas_fill      uplo,
                                         rocblas_operation  transA,
                                         rocblas_diagonal   diag,
                                         rocblas_int        m,
                                         rocblas_int        n,

```

(continues on next page)

(continued from previous page)

```

const float*      alpha,
const float* const A[],
rocbblas_int      lda,
const float* const B[],
rocbblas_int      ldb,
float* const      C[],
rocbblas_int      ldc,
rocbblas_int      batch_count)

#endif

#ifdef ROCBLAS_V3 // deprecated
    rocbblas_status rocbblas_dtrmm_batched(rocbblas_handle handle,
        rocbblas_side      side,
        rocbblas_fill      upto,
        rocbblas_operation transA,
        rocbblas_diagonal  diag,
        rocbblas_int      m,
        rocbblas_int      n,
        const double*      alpha,
        const double* const A[],
        rocbblas_int      lda,
        double* const      B[],
        rocbblas_int      ldb,
        rocbblas_int      batch_count)

#else // available in rocBLAS version 3.x.x and later with -DROCBLAS_V3

    rocbblas_status rocbblas_dtrmm_batched(rocbblas_handle handle,
        rocbblas_side      side,
        rocbblas_fill      upto,
        rocbblas_operation transA,
        rocbblas_diagonal  diag,
        rocbblas_int      m,
        rocbblas_int      n,
        const double*      alpha,
        const double* const A[],
        rocbblas_int      lda,
        const double* const B[],
        rocbblas_int      ldb,
        double* const      C[],
        rocbblas_int      ldc,
        rocbblas_int      batch_count)

#endif

#ifdef ROCBLAS_V3 // deprecated
    rocbblas_status rocbblas_ctrmm_batched(rocbblas_handle handle,
        rocbblas_side      side,
        rocbblas_fill      upto,
        rocbblas_operation transA,
        rocbblas_diagonal  diag,
        rocbblas_int      m,
        rocbblas_int      n,

```

(continues on next page)

(continued from previous page)

```

const rocblas_float_complex* alpha,
const rocblas_float_complex* const A[],
rocblas_int lda,
rocblas_float_complex* const B[],
rocblas_int ldb,
rocblas_int batch_

↪count)

#else // available in rocBLAS version 3.x.x and later with -DROCBLAS_V3

rocblas_status rocblas_ctrmm_batched(rocblas_handle handle,
rocblas_side side,
rocblas_fill uplo,
rocblas_operation transA,
rocblas_diagonal diag,
rocblas_int m,
rocblas_int n,
const rocblas_float_complex* alpha,
const rocblas_float_complex* const A[],
rocblas_int lda,
const rocblas_float_complex* const B[],
rocblas_int ldb,
rocblas_float_complex* const C[],
rocblas_int ldc,
rocblas_int batch_

↪count)
#endif

#ifndef ROCBLAS_V3 // deprecated
rocblas_status rocblas_ztrmm_batched(rocblas_handle handle,
rocblas_side side,
rocblas_fill uplo,
rocblas_operation transA,
rocblas_diagonal diag,
rocblas_int m,
rocblas_int n,
const rocblas_double_complex* alpha,
const rocblas_double_complex* const A[],
rocblas_int lda,
const rocblas_double_complex* const B[],
rocblas_int ldb,
rocblas_int batch_

↪count)

#else // available in rocBLAS version 3.x.x and later with -DROCBLAS_V3

rocblas_status rocblas_ztrmm_batched(rocblas_handle handle,
rocblas_side side,
rocblas_fill uplo,
rocblas_operation transA,
rocblas_diagonal diag,
rocblas_int m,

```

(continues on next page)

(continued from previous page)

```

rocbblas_int          n,
const rocbblas_double_complex* alpha,
const rocbblas_double_complex* const A[],
rocbblas_int          lda,
const rocbblas_double_complex* const B[],
rocbblas_int          ldb,
rocbblas_double_complex* const C[],
rocbblas_int          ldc,
rocbblas_int          batch_

↪count)
#endif

```

The deprecated Legacy BLAS in-place trmm_batched performs one of the batched matrix-matrix operations:

```

B_i := alpha*op( A_i )*B_i,    or
B_i := alpha*B_i*op( A_i )    for i = 0, 1, ... batch_count -1,

```

The new trmm_batched performs one of the matrix-matrix operations:

```

C_i := alpha*op( A_i )*B_i,    or
C_i := alpha*B_i*op( A_i )    for i = 0, 1, ... batch_count -1,

```

The in-place functionality is still available in the new trmm_batched by setting pointer C equal to pointer B and ldc equal to ldb.

```

alpha is a scalar, B_i is an m by n matrix, C_i is an m by n matrix, A_i is a
↪unit, or
non-unit, upper or lower triangular matrix and op( A_i ) is one of

op( A_i ) = A_i    or    op( A_i ) = A_i^T    or    op( A_i ) = A_i^H.

```

Note that when diag == rocbblas_diagonal_unit the diagonal elements of A_i are not referenced either, but are assumed to be unity.

Parameters

- **handle** – [in] [rocbblas_handle] handle to the rocbblas library context queue.
- **side** – [in] [rocbblas_side] Specifies whether op(A_i) multiplies B_i from the left or right as follows:
 - rocbblas_side_left: $B_i := \alpha * \text{op}(A_i) * B_i$
 - rocbblas_side_right: $B_i := \alpha * B_i * \text{op}(A_i)$
- **uplo** – [in] [rocbblas_fill] Specifies whether the matrix A is an upper or lower triangular matrix as follows:
 - rocbblas_fill_upper: A is an upper triangular matrix.
 - rocbblas_fill_lower: A is a lower triangular matrix.
- **transA** – [in] [rocbblas_operation] Specifies the form of op(A_i) to be used in the matrix multiplication as follows:
 - rocbblas_operation_none: $\text{op}(A_i) = A_i$

- `rocblas_operation_transpose`: $\text{op}(A_i) = A_i^T$
- `rocblas_operation_conjugate_transpose`: $\text{op}(A_i) = A_i^H$
- **diag** – [in] [rocblas_diagonal] Specifies whether or not A_i is unit triangular as follows:
 - `rocblas_diagonal_unit`: A_i is assumed to be unit triangular.
 - `rocblas_diagonal_non_unit`: A_i is not assumed to be unit triangular.
- **m** – [in] [rocblas_int] m specifies the number of rows of B_i . $m \geq 0$.
- **n** – [in] [rocblas_int] n specifies the number of columns of B_i . $n \geq 0$.
- **alpha** – [in] α specifies the scalar α . When α is zero then A_i is not referenced and B_i need not be set before entry.
- **A** – [in] Device array of device pointers storing each matrix A_i on the GPU. Each A_i is of dimension (lda, k) , where k is m when `side == rocblas_side_left` and is n when `side == rocblas_side_right`.

When `uplo == rocblas_fill_upper` the leading k by k upper triangular part of the array A must contain the upper triangular matrix **and** the strictly lower triangular part of A **is not** referenced.

When `uplo == rocblas_fill_lower` the leading k by k lower triangular part of the array A must contain the lower triangular matrix **and** the strictly upper triangular part of A **is not** referenced.

- **lda** – [in] [rocblas_int] lda specifies the first dimension of A .

```
if side == rocblas_side_left, lda >= max( 1, m ),
if side == rocblas_side_right, lda >= max( 1, n ).
```

- **B** – [inout] device array of device pointers storing each matrix B_i on the GPU. On entry, the leading m by n part of the array B_i must contain the matrix B_i , and on exit is overwritten by the transformed matrix.
- **ldb** – [in] [rocblas_int] ldb specifies the first dimension of B_i . $ldb \geq \max(1, m)$.
- **batch_count** – [in] [rocblas_int] number of instances i in the batch.

```
rocblas_status rocblas_strmm_strided_batched(rocblas_handle handle, rocblas_side side, rocblas_fill uplo,
rocblas_operation transA, rocblas_diagonal diag, rocblas_int m, rocblas_int n, const float *alpha, const float *A,
rocblas_int lda, rocblas_stride stride_A, float *B, rocblas_int ldb, rocblas_stride stride_B, rocblas_int batch_count)
```

```
rocblas_status rocblas_dtrmm_strided_batched(rocblas_handle handle, rocblas_side side, rocblas_fill uplo,
rocblas_operation transA, rocblas_diagonal diag, rocblas_int m, rocblas_int n, const double *alpha, const double *A,
rocblas_int lda, rocblas_stride stride_A, double *B, rocblas_int ldb, rocblas_stride stride_B, rocblas_int batch_count)
```

```
rocblas_status rocblas_ctrmm_strided_batched(rocblas_handle handle, rocblas_side side, rocblas_fill uplo,
rocblas_operation transA, rocblas_diagonal diag, rocblas_int
m, rocblas_int n, const rocblas_float_complex *alpha, const
rocblas_float_complex *A, rocblas_int lda, rocblas_stride
stride_A, rocblas_float_complex *B, rocblas_int ldb,
rocblas_stride stride_B, rocblas_int batch_count)
```

```
rocblas_status rocblas_ztrmm_strided_batched(rocblas_handle handle, rocblas_side side, rocblas_fill uplo,
rocblas_operation transA, rocblas_diagonal diag, rocblas_int
m, rocblas_int n, const rocblas_double_complex *alpha, const
rocblas_double_complex *A, rocblas_int lda, rocblas_stride
stride_A, rocblas_double_complex *B, rocblas_int ldb,
rocblas_stride stride_B, rocblas_int batch_count)
```

BLAS Level 3 API

The rocBLAS trmm_strided_batched API is from Legacy BLAS and it supports only in-place functionality. It is deprecated and it will be replaced with an API that supports both in-place and out-of-place functionality. The new API is available in rocBLAS versions 3.x.x and later. To get the new API compile with the directive -DROCBLAS_V3.

```
#ifndef ROCBLAS_V3 // deprecated
    rocblas_status rocblas_strmm_strided_batched(rocblas_handle    handle,
                                                rocblas_side      side,
                                                rocblas_fill      uplo,
                                                rocblas_operation transA,
                                                rocblas_diagonal diag,
                                                rocblas_int       m,
                                                rocblas_int       n,
                                                const float*      alpha,
                                                const float*      A,
                                                rocblas_int       lda,
                                                rocblas_stride    stride_A,
                                                float*           B,
                                                rocblas_int       ldb,
                                                rocblas_stride    stride_B,
                                                rocblas_int       batch_count)

#else // available in rocBLAS version 3.x.x and later with -DROCBLAS_V3

    rocblas_status rocblas_strmm_strided_batched(rocblas_handle    handle,
                                                rocblas_side      side,
                                                rocblas_fill      uplo,
                                                rocblas_operation transA,
                                                rocblas_diagonal diag,
                                                rocblas_int       m,
                                                rocblas_int       n,
                                                const float*      alpha,
                                                const float*      A,
                                                rocblas_int       lda,
                                                rocblas_stride    stride_A,
                                                const float*      B,
                                                rocblas_int       ldb,
                                                rocblas_stride    stride_B,
                                                float*           C,
```

(continues on next page)

(continued from previous page)

```

rocblas_int      ldc,
rocblas_stride   stride_C,
rocblas_int      batch_count)

#endif

#ifdef ROCBLAS_V3 // deprecated
rocblas_status rocblas_dtrmm_strided_batched(rocblas_handle handle,
rocblas_side     side,
rocblas_fill     uplo,
rocblas_operation transA,
rocblas_diagonal diag,
rocblas_int      m,
rocblas_int      n,
const double*    alpha,
const double*    A,
rocblas_int      lda,
rocblas_stride   stride_A,
double*          B,
rocblas_int      ldb,
rocblas_stride   stride_B,
rocblas_int      batch_count)

#else // available in rocBLAS version 3.x.x and later with -DROCBLAS_V3

rocblas_status rocblas_dtrmm_strided_batched(rocblas_handle handle,
rocblas_side     side,
rocblas_fill     uplo,
rocblas_operation transA,
rocblas_diagonal diag,
rocblas_int      m,
rocblas_int      n,
const double*    alpha,
const double*    A,
rocblas_int      lda,
rocblas_stride   stride_A,
const double*    B,
rocblas_int      ldb,
rocblas_stride   stride_B,
double*          C,
rocblas_int      ldc,
rocblas_stride   stride_C,
rocblas_int      batch_count)

#endif

#ifdef ROCBLAS_V3 // deprecated
rocblas_status rocblas_ctrmm_strided_batched(rocblas_handle
↪ handle,
rocblas_side     side,
rocblas_fill     uplo,
rocblas_operation transA,
↪ transA,
rocblas_diagonal diag,

```

(continues on next page)

(continued from previous page)

```

        rocbldas_int          m,
        rocbldas_int          n,
        const rocbldas_float_complex* alpha,
        const rocbldas_float_complex* A,
        rocbldas_int          lda,
        rocbldas_stride       ↵
↵stride_A,

        rocbldas_float_complex* B,
        rocbldas_int          ldb,
        rocbldas_stride       ↵
↵stride_B,

        rocbldas_int batch_count)

#else // available in rocBLAS version 3.x.x and later with -DROCBLAS_V3

    rocbldas_status rocbldas_ctrmm_strided_batched(rocbldas_handle ↵
↵handle,

        rocbldas_side         side,
        rocbldas_fill         uplo,
        rocbldas_operation    ↵
↵transA,

        rocbldas_diagonal     diag,
        rocbldas_int          m,
        rocbldas_int          n,
        const rocbldas_float_complex* alpha,
        const rocbldas_float_complex* A,
        rocbldas_int          lda,
        rocbldas_stride       ↵
↵stride_A,

        const rocbldas_float_complex* B,
        rocbldas_int          ldb,
        rocbldas_stride       ↵
↵stride_B,

        rocbldas_float_complex* C,
        rocbldas_int          ldc,
        rocbldas_stride       ↵
↵stride_C,

        rocbldas_int batch_count)

#endif

#ifdef ROCBLAS_V3 // deprecated
    rocbldas_status rocbldas_ztrmm_strided_batched(rocbldas_handle ↵
↵handle,

        rocbldas_side         side,
        rocbldas_fill         uplo,
        rocbldas_operation    ↵
↵transA,

        rocbldas_diagonal     diag,
        rocbldas_int          m,
        rocbldas_int          n,
        const rocbldas_double_complex* ↵
↵alpha,

```

(continues on next page)

(continued from previous page)

```

const rocblas_double_complex* A,
rocblas_int lda,
rocblas_stride ↵
↵stride_A,

rocblas_double_complex* B,
rocblas_int ldb,
rocblas_stride ↵
↵stride_B,

rocblas_int batch_count)

#else // available in rocBLAS version 3.x.x and later with -DROCBLAS_V3

rocblas_status rocblas_ztrmm_strided_batched(rocblas_handle ↵
↵handle,

rocblas_side side,
rocblas_fill uplo,
rocblas_operation ↵
↵transA,

rocblas_diagonal diag,
rocblas_int m,
rocblas_int n,
const rocblas_double_complex* ↵
↵alpha,

const rocblas_double_complex* A,
rocblas_int lda,
rocblas_stride ↵
↵stride_A,

const rocblas_double_complex* B,
rocblas_int ldb,
rocblas_stride ↵
↵stride_B,

rocblas_double_complex* C,
rocblas_int ldc,
rocblas_stride ↵
↵stride_C,

rocblas_int batch_count)

#endif

```

The deprecated Legacy BLAS in-place trmm_strided_batched performs one of the strided_batched matrix-matrix operations:

```

B_i := alpha*op( A_i )*B_i,    or
B_i := alpha*B_i*op( A_i )   for i = 0, 1, ... batch_count -1,

```

The new trmm_batched performs one of the matrix-matrix operations:

```

C_i := alpha*op( A_i )*B_i,    or
C_i := alpha*B_i*op( A_i )   for i = 0, 1, ... batch_count -1,

```

The in-place functionality is still available in the new trmm_batched by setting pointer C equal to pointer B, setting ldc equal to ldb, and setting stride_C equal to stride_B.

alpha **is** a scalar, B_i **is** an m by n matrix, C_i **is** an m by n matrix, A_i **is** a
 ↪ unit, **or**
 non-unit, upper **or** lower triangular matrix **and** op(A_i) **is** one of

op(A_i) = A_i **or**
 op(A_i) = A_i^T **or**
 op(A_i) = A_i^H.

Note that when diag == rocblas_diagonal_unit the diagonal elements of A_i are not referenced either, but are assumed to be unity.

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **side** – [in] [rocblas_side] Specifies whether op(A_i) multiplies B_i from the left or right as follows:
 - rocblas_side_left: $B_i := \alpha * \text{op}(A_i) * B_i$
 - rocblas_side_right: $B_i := \alpha * B_i * \text{op}(A_i)$
- **uplo** – [in] [rocblas_fill] Specifies whether the matrix A is an upper or lower triangular matrix as follows:
 - rocblas_fill_upper: A is an upper triangular matrix
 - rocblas_fill_lower: A is a lower triangular matrix
- **transA** – [in] [rocblas_operation] Specifies the form of op(A_i) to be used in the matrix multiplication as follows:
 - rocblas_operation_none: $\text{op}(A_i) = A_i$
 - rocblas_operation_transpose: $\text{op}(A_i) = A_i^T$
 - rocblas_operation_conjugate_transpose: $\text{op}(A_i) = A_i^H$
- **diag** – [in] [rocblas_diagonal] Specifies whether or not A_i is unit triangular as follows:
 - rocblas_diagonal_unit: A_i is assumed to be unit triangular.
 - rocblas_diagonal_non_unit: A_i is not assumed to be unit triangular.
- **m** – [in] [rocblas_int] m specifies the number of rows of B_i. $m \geq 0$.
- **n** – [in] [rocblas_int] n specifies the number of columns of B_i. $n \geq 0$.
- **alpha** – [in] alpha specifies the scalar alpha. When alpha is zero then A_i is not referenced and B_i need not be set before entry.
- **A** – [in] Device pointer to the first matrix A_0 on the GPU. Each A_i is of dimension (lda, k), where k is m when side == rocblas_side_left and is n when side == rocblas_side_right.

When uplo == rocblas_fill_upper the leading k by k upper triangular part of the array A must contain the upper triangular matrix **and** the strictly lower triangular part of A **is not** referenced.

When uplo == rocblas_fill_lower the leading k by k lower triangular part of the array A must contain the lower

(continues on next page)

(continued from previous page)

triangular matrix **and** the strictly upper triangular part of A **is not** referenced.

- **lda** – [in] [rocblas_int] lda specifies the first dimension of A.

```
if side == rocblas_side_left, lda >= max( 1, m ),
if side == rocblas_side_right, lda >= max( 1, n ).
```

- **stride_A** – [in] [rocblas_stride] stride from the start of one matrix (A_i) and the next one (A_i+1).
- **B** – [inout] Device pointer to the first matrix B_0 on the GPU. On entry, the leading m by n part of the array B_i must contain the matrix B_i, and on exit is overwritten by the transformed matrix.
- **ldb** – [in] [rocblas_int] ldb specifies the first dimension of B_i. $ldb \geq \max(1, m)$.
- **stride_B** – [in] [rocblas_stride] stride from the start of one matrix (B_i) and the next one (B_i+1).
- **batch_count** – [in] [rocblas_int] number of instances i in the batch.

5.6.7 rocblas_Xtrsm + batched, strided_batched

rocblas_status **rocblas_strsm**(*rocblas_handle* handle, *rocblas_side* side, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_diagonal* diag, *rocblas_int* m, *rocblas_int* n, const float *alpha, const float *A, *rocblas_int* lda, float *B, *rocblas_int* ldb)

rocblas_status **rocblas_dtrsm**(*rocblas_handle* handle, *rocblas_side* side, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_diagonal* diag, *rocblas_int* m, *rocblas_int* n, const double *alpha, const double *A, *rocblas_int* lda, double *B, *rocblas_int* ldb)

rocblas_status **rocblas_ctrsm**(*rocblas_handle* handle, *rocblas_side* side, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_diagonal* diag, *rocblas_int* m, *rocblas_int* n, const *rocblas_float_complex* *alpha, const *rocblas_float_complex* *A, *rocblas_int* lda, *rocblas_float_complex* *B, *rocblas_int* ldb)

rocblas_status **rocblas_ztrsm**(*rocblas_handle* handle, *rocblas_side* side, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_diagonal* diag, *rocblas_int* m, *rocblas_int* n, const *rocblas_double_complex* *alpha, const *rocblas_double_complex* *A, *rocblas_int* lda, *rocblas_double_complex* *B, *rocblas_int* ldb)

BLAS Level 3 API

trsm solves:

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

where alpha **is** a scalar, X **and** B are m by n matrices,

A **is** triangular matrix **and** op(A) **is** one of

$\text{op}(A) = A$ **or** $\text{op}(A) = A^T$ **or** $\text{op}(A) = A^H$.

The matrix X **is** overwritten on B.

Note about memory allocation: When `trsm` is launched with a `k` evenly divisible by the internal block size of 128, and is no larger than 10 of these blocks, the API takes advantage of utilizing pre-allocated memory found in the handle to increase overall performance. This memory can be managed by using the environment variable `WORKBUF_TRSM_B_CHNK`. When this variable is not set the device memory used for temporary storage will default to 1 MB and may result in chunking, which in turn may reduce performance. Under these circumstances it is recommended that `WORKBUF_TRSM_B_CHNK` be set to the desired chunk of right hand sides to be used at a time (where `k` is `m` when `roclblas_side_left` and is `n` when `roclblas_side_right`).

Parameters

- **handle** – [in] [`roclblas_handle`] handle to the roclblas library context queue.
- **side** – [in] [`roclblas_side`]
 - `roclblas_side_left`: $\text{op}(A) * X = \alpha * B$
 - `roclblas_side_right`: $X * \text{op}(A) = \alpha * B$
- **uplo** – [in] [`roclblas_fill`]
 - `roclblas_fill_upper`: `A` is an upper triangular matrix.
 - `roclblas_fill_lower`: `A` is a lower triangular matrix.
- **transA** – [in] [`roclblas_operation`]
 - `transB`: $\text{op}(A) = A$.
 - `roclblas_operation_transpose`: $\text{op}(A) = A^T$
 - `roclblas_operation_conjugate_transpose`: $\text{op}(A) = A^H$
- **diag** – [in] [`roclblas_diagonal`]
 - `roclblas_diagonal_unit`: `A` is assumed to be unit triangular.
 - `roclblas_diagonal_non_unit`: `A` is not assumed to be unit triangular.
- **m** – [in] [`roclblas_int`] `m` specifies the number of rows of `B`. $m \geq 0$.
- **n** – [in] [`roclblas_int`] `n` specifies the number of columns of `B`. $n \geq 0$.
- **alpha** – [in] device pointer or host pointer specifying the scalar `alpha`. When `alpha` is `&zero` then `A` is not referenced and `B` need not be set before entry.
- **A** – [in] device pointer storing matrix `A`. of dimension (`lda`, `k`), where `k` is `m` when `roclblas_side_left` and is `n` when `roclblas_side_right` only the upper/lower triangular part is accessed.
- **lda** – [in] [`roclblas_int`] `lda` specifies the first dimension of `A`.


```
if side == roclblas_side_left, lda >= max( 1, m ),
if side == roclblas_side_right, lda >= max( 1, n ).
```
- **B** – [inout] device pointer storing matrix `B`.
- **ldb** – [in] [`roclblas_int`] `ldb` specifies the first dimension of `B`. $ldb \geq \max(1, m)$.

roclblas_status **roclblas_strsm_batched**(*roclblas_handle* handle, *roclblas_side* side, *roclblas_fill* uplo, *roclblas_operation* transA, *roclblas_diagonal* diag, *roclblas_int* m, *roclblas_int* n, const float *alpha, const float *const A[], *roclblas_int* lda, float *const B[], *roclblas_int* ldb, *roclblas_int* batch_count)

```
rocblas_status rocblas_dtrsm_batched(rocblas_handle handle, rocblas_side side, rocblas_fill uplo,
                                     rocblas_operation transA, rocblas_diagonal diag, rocblas_int m,
                                     rocblas_int n, const double *alpha, const double *const A[], rocblas_int
                                     lda, double *const B[], rocblas_int ldb, rocblas_int batch_count)
```

```
rocblas_status rocblas_ctrsm_batched(rocblas_handle handle, rocblas_side side, rocblas_fill uplo,
                                     rocblas_operation transA, rocblas_diagonal diag, rocblas_int m,
                                     rocblas_int n, const rocblas_float_complex *alpha, const
                                     rocblas_float_complex *const A[], rocblas_int lda,
                                     rocblas_float_complex *const B[], rocblas_int ldb, rocblas_int
                                     batch_count)
```

```
rocblas_status rocblas_ztrsm_batched(rocblas_handle handle, rocblas_side side, rocblas_fill uplo,
                                     rocblas_operation transA, rocblas_diagonal diag, rocblas_int m,
                                     rocblas_int n, const rocblas_double_complex *alpha, const
                                     rocblas_double_complex *const A[], rocblas_int lda,
                                     rocblas_double_complex *const B[], rocblas_int ldb, rocblas_int
                                     batch_count)
```

BLAS Level 3 API

trsm_batched performs the following batched operation:

```
op(A_i)*X_i = alpha*B_i or
X_i*op(A_i) = alpha*B_i, for i = 1, ..., batch_count,

where alpha is a scalar, X and B are batched m by n matrices,

A is triangular batched matrix and op(A) is one of

op( A ) = A or
op( A ) = A^T or
op( A ) = A^H.
```

Each matrix X_i is overwritten on B_i for i = 1, ..., batch_count.

Note about memory allocation: When trsm is launched with a k evenly divisible by the internal block size of 128, and is no larger than 10 of these blocks, the API takes advantage of utilizing pre-allocated memory found in the handle to increase overall performance. This memory can be managed by using the environment variable WORKBUF_TRSM_B_CHNK. When this variable is not set the device memory used for temporary storage will default to 1 MB and may result in chunking, which in turn may reduce performance. Under these circumstances it is recommended that WORKBUF_TRSM_B_CHNK be set to the desired chunk of right hand sides to be used at a time (where k is m when rocblas_side_left and is n when rocblas_side_right).

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **side** – [in] [rocblas_side]
 - rocblas_side_left: $\text{op}(A) * X = \alpha * B$
 - rocblas_side_right: $X * \text{op}(A) = \alpha * B$
- **uplo** – [in] [rocblas_fill]
 - rocblas_fill_upper: each A_i is an upper triangular matrix.
 - rocblas_fill_lower: each A_i is a lower triangular matrix.

- **transA** – [in] [rocblas_operation]
 - transB: $\text{op}(A) = A$
 - rocblas_operation_transpose: $\text{op}(A) = A^T$
 - rocblas_operation_conjugate_transpose: $\text{op}(A) = A^H$
- **diag** – [in] [rocblas_diagonal]
 - rocblas_diagonal_unit: each A_i is assumed to be unit triangular.
 - rocblas_diagonal_non_unit: each A_i is not assumed to be unit triangular.
- **m** – [in] [rocblas_int] m specifies the number of rows of each B_i . $m \geq 0$.
- **n** – [in] [rocblas_int] n specifies the number of columns of each B_i . $n \geq 0$.
- **alpha** – [in] device pointer or host pointer specifying the scalar alpha. When alpha is &zero then A is not referenced and B need not be set before entry.
- **A** – [in] device array of device pointers storing each matrix A_i on the GPU. Matrices are of dimension (lda, k) , where k is m when rocblas_side_left and is n when rocblas_side_right only the upper/lower triangular part is accessed.
- **lda** – [in] [rocblas_int] lda specifies the first dimension of each A_i .


```
if side = rocblas_side_left,  lda >= max( 1, m ),
if side = rocblas_side_right, lda >= max( 1, n ).
```
- **B** – [inout] device array of device pointers storing each matrix B_i on the GPU.
- **ldb** – [in] [rocblas_int] ldb specifies the first dimension of each B_i . $ldb \geq \max(1, m)$.
- **batch_count** – [in] [rocblas_int] number of trsm operations in the batch.

rocblas_status rocblas_strsm_strided_batched(*rocblas_handle* handle, *rocblas_side* side, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_diagonal* diag, *rocblas_int* m, *rocblas_int* n, const float *alpha, const float *A, *rocblas_int* lda, *rocblas_stride* stride_a, float *B, *rocblas_int* ldb, *rocblas_stride* stride_b, *rocblas_int* batch_count)

rocblas_status rocblas_dtrsm_strided_batched(*rocblas_handle* handle, *rocblas_side* side, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_diagonal* diag, *rocblas_int* m, *rocblas_int* n, const double *alpha, const double *A, *rocblas_int* lda, *rocblas_stride* stride_a, double *B, *rocblas_int* ldb, *rocblas_stride* stride_b, *rocblas_int* batch_count)

rocblas_status rocblas_ctrsm_strided_batched(*rocblas_handle* handle, *rocblas_side* side, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_diagonal* diag, *rocblas_int* m, *rocblas_int* n, const *rocblas_float_complex* *alpha, const *rocblas_float_complex* *A, *rocblas_int* lda, *rocblas_stride* stride_a, *rocblas_float_complex* *B, *rocblas_int* ldb, *rocblas_stride* stride_b, *rocblas_int* batch_count)

rocblas_status rocblas_ztrsm_strided_batched(*rocblas_handle* handle, *rocblas_side* side, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_diagonal* diag, *rocblas_int* m, *rocblas_int* n, const *rocblas_double_complex* *alpha, const *rocblas_double_complex* *A, *rocblas_int* lda, *rocblas_stride* stride_a, *rocblas_double_complex* *B, *rocblas_int* ldb, *rocblas_stride* stride_b, *rocblas_int* batch_count)

BLAS Level 3 API

trsm_srided_batched performs the following strided batched operation:

```

op(A_i)*X_i = alpha*B_i or
X_i*op(A_i) = alpha*B_i, for i = 1, ..., batch_count,

where alpha is a scalar, X and B are strided batched m by n matrices,

A is triangular strided batched matrix and op(A) is one of

op( A ) = A or
op( A ) = A^T or
op( A ) = A^H.

Each matrix X_i is overwritten on B_i for i = 1, ..., batch_count.

```

Note about memory allocation: When trsm is launched with a k evenly divisible by the internal block size of 128, and is no larger than 10 of these blocks, the API takes advantage of utilizing pre-allocated memory found in the handle to increase overall performance. This memory can be managed by using the environment variable WORKBUF_TRSM_B_CHNK. When this variable is not set the device memory used for temporary storage will default to 1 MB and may result in chunking, which in turn may reduce performance. Under these circumstances it is recommended that WORKBUF_TRSM_B_CHNK be set to the desired chunk of right hand sides to be used at a time (where k is m when rocblas_side_left and is n when rocblas_side_right).

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **side** – [in] [rocblas_side]
 - rocblas_side_left: $\text{op}(A)*X = \alpha*B$.
 - rocblas_side_right: $X*\text{op}(A) = \alpha*B$.
- **uplo** – [in] [rocblas_fill]
 - rocblas_fill_upper: each A_i is an upper triangular matrix.
 - rocblas_fill_lower: each A_i is a lower triangular matrix.
- **transA** – [in] [rocblas_operation]
 - transB: $\text{op}(A) = A$.
 - rocblas_operation_transpose: $\text{op}(A) = A^T$.
 - rocblas_operation_conjugate_transpose: $\text{op}(A) = A^H$.
- **diag** – [in] [rocblas_diagonal]
 - rocblas_diagonal_unit: each A_i is assumed to be unit triangular.
 - rocblas_diagonal_non_unit: each A_i is not assumed to be unit triangular.
- **m** – [in] [rocblas_int] m specifies the number of rows of each B_i . $m \geq 0$.
- **n** – [in] [rocblas_int] n specifies the number of columns of each B_i . $n \geq 0$.
- **alpha** – [in] device pointer or host pointer specifying the scalar alpha. When alpha is &zero then A is not referenced and B need not be set before entry.

- **A** – [in] device pointer pointing to the first matrix A₁. of dimension (lda, k), where k is m when rocblas_side_left and is n when rocblas_side_right only the upper/lower triangular part is accessed.
- **lda** – [in] [rocblas_int] lda specifies the first dimension of each A_i.

```
if side == rocblas_side_left, lda >= max( 1, m ).
if side == rocblas_side_right, lda >= max( 1, n ).
```

- **stride_a** – [in] [rocblas_stride] stride from the start of one A_i matrix to the next A_(i + 1).
- **B** – [inout] device pointer pointing to the first matrix B₁.
- **ldb** – [in] [rocblas_int] ldb specifies the first dimension of each B_i. ldb >= max(1, m).
- **stride_b** – [in] [rocblas_stride] stride from the start of one B_i matrix to the next B_(i + 1).
- **batch_count** – [in] [rocblas_int] number of trsm operations in the batch.

5.6.8 rocblas_Xhemm + batched, strided_batched

*rocblas_status rocblas_chemm(rocblas_handle handle, rocblas_side side, rocblas_fill uplo, rocblas_int m, rocblas_int n, const rocblas_float_complex *alpha, const rocblas_float_complex *A, rocblas_int lda, const rocblas_float_complex *B, rocblas_int ldb, const rocblas_float_complex *beta, rocblas_float_complex *C, rocblas_int ldc)*

*rocblas_status rocblas_zhemm(rocblas_handle handle, rocblas_side side, rocblas_fill uplo, rocblas_int m, rocblas_int n, const rocblas_double_complex *alpha, const rocblas_double_complex *A, rocblas_int lda, const rocblas_double_complex *B, rocblas_int ldb, const rocblas_double_complex *beta, rocblas_double_complex *C, rocblas_int ldc)*

BLAS Level 3 API

hemm performs one of the matrix-matrix operations:

```
C := alpha*A*B + beta*C if side == rocblas_side_left,
C := alpha*B*A + beta*C if side == rocblas_side_right,
```

where alpha and beta are scalars, B and C are m by n matrices, and A is a Hermitian matrix stored as either upper or lower.

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **side** – [in] [rocblas_side]
 - rocblas_side_left: C := alpha*A*B + beta*C
 - rocblas_side_right: C := alpha*B*A + beta*C
- **uplo** – [in] [rocblas_fill]
 - rocblas_fill_upper: A is an upper triangular matrix
 - rocblas_fill_lower: A is a lower triangular matrix
- **m** – [in] [rocblas_int] m specifies the number of rows of B and C. m >= 0.

- **n** – [in] [rocblas_int] n specifies the number of columns of B and C. $n \geq 0$.
- **alpha** – [in] alpha specifies the scalar alpha. When alpha is zero then A and B are not referenced.
- **A** – [in] pointer storing matrix A on the GPU.
 - A is m by m if `side == rocblas_side_left`
 - A is n by n if `side == rocblas_side_right` Only the upper/lower triangular part is accessed. The imaginary component of the diagonal elements is not used.
- **lda** – [in] [rocblas_int] lda specifies the first dimension of A.

```
if side == rocblas_side_left, lda >= max( 1, m ),
otherwise lda >= max( 1, n ).
```

- **B** – [in] pointer storing matrix B on the GPU. Matrix dimension is m by n
- **ldb** – [in] [rocblas_int] ldb specifies the first dimension of B. $ldb \geq \max(1, m)$.
- **beta** – [in] beta specifies the scalar beta. When beta is zero then C need not be set before entry.
- **C** – [in] pointer storing matrix C on the GPU. Matrix dimension is m by n
- **ldc** – [in] [rocblas_int] ldc specifies the first dimension of C. $ldc \geq \max(1, m)$.

```
rocblas_status rocblas_chemm_batched(rocblas_handle handle, rocblas_side side, rocblas_fill uplo, rocblas_int
m, rocblas_int n, const rocblas_float_complex *alpha, const
rocblas_float_complex *const A[], rocblas_int lda, const
rocblas_float_complex *const B[], rocblas_int ldb, const
rocblas_float_complex *beta, rocblas_float_complex *const C[],
rocblas_int ldc, rocblas_int batch_count)
```

```
rocblas_status rocblas_zhemm_batched(rocblas_handle handle, rocblas_side side, rocblas_fill uplo, rocblas_int
m, rocblas_int n, const rocblas_double_complex *alpha, const
rocblas_double_complex *const A[], rocblas_int lda, const
rocblas_double_complex *const B[], rocblas_int ldb, const
rocblas_double_complex *beta, rocblas_double_complex *const C[],
rocblas_int ldc, rocblas_int batch_count)
```

BLAS Level 3 API

hemm_batched performs a batch of the matrix-matrix operations:

```
C_i := alpha*A_i*B_i + beta*C_i if side == rocblas_side_left,
C_i := alpha*B_i*A_i + beta*C_i if side == rocblas_side_right,
```

where alpha and beta are scalars, B_i and C_i are m by n matrices, and A_i is a Hermitian matrix stored as either upper or lower.

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **side** – [in] [rocblas_side]
 - `rocblas_side_left`: $C_i := \alpha A_i B_i + \beta C_i$
 - `rocblas_side_right`: $C_i := \alpha B_i A_i + \beta C_i$

- **uplo** – [in] [rocblas_fill]
 - rocblas_fill_upper: A_i is an upper triangular matrix
 - rocblas_fill_lower: A_i is a lower triangular matrix
- **m** – [in] [rocblas_int] m specifies the number of rows of B_i and C_i. $m \geq 0$.
- **n** – [in] [rocblas_int] n specifies the number of columns of B_i and C_i. $n \geq 0$.
- **alpha** – [in] alpha specifies the scalar alpha. When alpha is zero then A_i and B_i are not referenced.
- **A** – [in] device array of device pointers storing each matrix A_i on the GPU.
 - A_i is m by m if side == rocblas_side_left
 - A_i is n by n if side == rocblas_side_right Only the upper/lower triangular part is accessed. The imaginary component of the diagonal elements is not used.
- **lda** – [in] [rocblas_int] lda specifies the first dimension of A_i.

```
if side == rocblas_side_left, lda >= max( 1, m ),
otherwise lda >= max( 1, n ).
```

- **B** – [in] device array of device pointers storing each matrix B_i on the GPU. Matrix dimension is m by n
- **ldb** – [in] [rocblas_int] ldb specifies the first dimension of B_i. $ldb \geq \max(1, m)$.
- **beta** – [in] beta specifies the scalar beta. When beta is zero then C_i need not be set before entry.
- **C** – [in] device array of device pointers storing each matrix C_i on the GPU. Matrix dimension is m by n
- **ldc** – [in] [rocblas_int] ldc specifies the first dimension of C_i. $ldc \geq \max(1, m)$.
- **batch_count** – [in] [rocblas_int] number of instances in the batch.

```
rocblas_status rocblas_chemm_strided_batched(rocblas_handle handle, rocblas_side side, rocblas_fill uplo,
rocblas_int m, rocblas_int n, const rocblas_float_complex
*alpha, const rocblas_float_complex *A, rocblas_int lda,
rocblas_stride stride_A, const rocblas_float_complex *B,
rocblas_int ldb, rocblas_stride stride_B, const
rocblas_float_complex *beta, rocblas_float_complex *C,
rocblas_int ldc, rocblas_stride stride_C, rocblas_int
batch_count)
```

```
rocblas_status rocblas_zhemm_strided_batched(rocblas_handle handle, rocblas_side side, rocblas_fill uplo,
rocblas_int m, rocblas_int n, const rocblas_double_complex
*alpha, const rocblas_double_complex *A, rocblas_int lda,
rocblas_stride stride_A, const rocblas_double_complex *B,
rocblas_int ldb, rocblas_stride stride_B, const
rocblas_double_complex *beta, rocblas_double_complex *C,
rocblas_int ldc, rocblas_stride stride_C, rocblas_int
batch_count)
```

BLAS Level 3 API

hemm_strided_batched performs a batch of the matrix-matrix operations:

```
C_i := alpha*A_i*B_i + beta*C_i if side == rocblas_side_left,
C_i := alpha*B_i*A_i + beta*C_i if side == rocblas_side_right,
```

where alpha and beta are scalars, B_i and C_i are m by n matrices, and A_i is a Hermitian matrix stored as either upper or lower.

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **side** – [in] [rocblas_side]
 - rocblas_side_left: $C_i := \alpha A_i B_i + \beta C_i$
 - rocblas_side_right: $C_i := \alpha B_i A_i + \beta C_i$
- **uplo** – [in] [rocblas_fill]
 - rocblas_fill_upper: A_i is an upper triangular matrix
 - rocblas_fill_lower: A_i is a lower triangular matrix
- **m** – [in] [rocblas_int] m specifies the number of rows of B_i and C_i. $m \geq 0$.
- **n** – [in] [rocblas_int] n specifies the number of columns of B_i and C_i. $n \geq 0$.
- **alpha** – [in] alpha specifies the scalar alpha. When alpha is zero then A_i and B_i are not referenced.
- **A** – [in] device pointer to first matrix A_1
 - A_i is m by m if side == rocblas_side_left
 - A_i is n by n if side == rocblas_side_right Only the upper/lower triangular part is accessed. The imaginary component of the diagonal elements is not used.
- **lda** – [in] [rocblas_int] lda specifies the first dimension of A_i.

```
if side = rocblas_side_left,  lda >= max( 1, m ),
otherwise lda >= max( 1, n ).
```

- **stride_A** – [in] [rocblas_stride] stride from the start of one matrix (A_i) and the next one (A_{i+1}).
- **B** – [in] device pointer to first matrix B_1 of dimension (ldb, n) on the GPU
- **ldb** – [in] [rocblas_int] ldb specifies the first dimension of B_i.

```
if side = rocblas_operation_none,  ldb >= max( 1, m ),
otherwise ldb >= max( 1, n ).
```

- **stride_B** – [in] [rocblas_stride] stride from the start of one matrix (B_i) and the next one (B_{i+1}).
- **beta** – [in] beta specifies the scalar beta. When beta is zero then C need not be set before entry.
- **C** – [in] device pointer to first matrix C_1 of dimension (ldc, n) on the GPU.
- **ldc** – [in] [rocblas_int] ldc specifies the first dimension of C. $ldc \geq \max(1, m)$.
- **stride_C** – [inout] [rocblas_stride] stride from the start of one matrix (C_i) and the next one (C_{i+1}).

- **batch_count** – [in] [rocblas_int] number of instances in the batch.

5.6.9 rocblas_Xherk + batched, strided_batched

rocblas_status **rocblas_cherk**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_int* n, *rocblas_int* k, const float *alpha, const *rocblas_float_complex* *A, *rocblas_int* lda, const float *beta, *rocblas_float_complex* *C, *rocblas_int* ldc)

rocblas_status **rocblas_zherk**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_int* n, *rocblas_int* k, const double *alpha, const *rocblas_double_complex* *A, *rocblas_int* lda, const double *beta, *rocblas_double_complex* *C, *rocblas_int* ldc)

BLAS Level 3 API

herk performs one of the matrix-matrix operations for a Hermitian rank-k update:

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

where α and β are scalars, $\text{op}(A)$ is an n by k matrix, and C is a $n \times n$ Hermitian matrix stored as either upper or lower.

$\text{op}(A) = A$, and A is n by k if $\text{transA} == \text{rocblas_operation_none}$

$\text{op}(A) = A^H$ and A is k by n if $\text{transA} == \text{rocblas_operation_conjugate_transpose}$

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill]
 - *rocblas_fill_upper*: C is an upper triangular matrix
 - *rocblas_fill_lower*: C is a lower triangular matrix
- **transA** – [in] [rocblas_operation]
 - *rocblas_operation_conjugate_transpose*: $\text{op}(A) = A^H$
 - *rocblas_operation_none*: $\text{op}(A) = A$
- **n** – [in] [rocblas_int] n specifies the number of rows and columns of C . $n \geq 0$.
- **k** – [in] [rocblas_int] k specifies the number of columns of $\text{op}(A)$. $k \geq 0$.
- **alpha** – [in] α specifies the scalar α . When α is zero then A is not referenced and A need not be set before entry.
- **A** – [in] pointer storing matrix A on the GPU. Matrix dimension is (lda, k) when if $\text{transA} = \text{rocblas_operation_none}$, otherwise (lda, n)
- **lda** – [in] [rocblas_int] lda specifies the first dimension of A .

```
if transA == rocblas_operation_none, lda >= max(1, n),
otherwise lda >= max(1, k).
```

- **beta** – [in] β specifies the scalar β . When β is zero then C need not be set before entry.
- **C** – [in] pointer storing matrix C on the GPU. The imaginary component of the diagonal elements are not used but are set to zero unless quick return. only the upper/lower triangular part is accessed.

- **ldc** – [in] [rocblas_int] ldc specifies the first dimension of C. $\text{ldc} \geq \max(1, n)$.

```
rocblas_status rocblas_cherk_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_operation transA,
                                     rocblas_int n, rocblas_int k, const float *alpha, const
                                     rocblas_float_complex *const A[], rocblas_int lda, const float *beta,
                                     rocblas_float_complex *const C[], rocblas_int ldc, rocblas_int
                                     batch_count)
```

```
rocblas_status rocblas_zherk_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_operation transA,
                                     rocblas_int n, rocblas_int k, const double *alpha, const
                                     rocblas_double_complex *const A[], rocblas_int lda, const double *beta,
                                     rocblas_double_complex *const C[], rocblas_int ldc, rocblas_int
                                     batch_count)
```

BLAS Level 3 API

herk_batched performs a batch of the matrix-matrix operations for a Hermitian rank-k update:

```
C_i := alpha*op( A_i )*op( A_i )^H + beta*C_i,
```

where **alpha** and **beta** are scalars, **op(A)** is an n by k matrix, and **C_i** is a n x n Hermitian matrix stored as either upper or lower.

```
op( A_i ) = A_i, and A_i is n by k if transA == rocblas_operation_none
op( A_i ) = A_i^H and A_i is k by n if transA == rocblas_operation_conjugate_
↪transpose
```

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill]
 - rocblas_fill_upper: C_i is an upper triangular matrix
 - rocblas_fill_lower: C_i is a lower triangular matrix
- **transA** – [in] [rocblas_operation]
 - rocblas_operation_conjugate_transpose: $\text{op}(A) = A^H$
 - rocblas_operation_none: $\text{op}(A) = A$
- **n** – [in] [rocblas_int] n specifies the number of rows and columns of C_i. $n \geq 0$.
- **k** – [in] [rocblas_int] k specifies the number of columns of op(A). $k \geq 0$.
- **alpha** – [in] alpha specifies the scalar alpha. When alpha is zero then A is not referenced and A need not be set before entry.
- **A** – [in] device array of device pointers storing each matrix_i A of dimension (lda, k) when transA is rocblas_operation_none, otherwise of dimension (lda, n).
- **lda** – [in] [rocblas_int] lda specifies the first dimension of A_i.

```
if transA == rocblas_operation_none, lda >= max( 1, n ),
otherwise lda >= max( 1, k ).
```

- **beta** – [in] beta specifies the scalar beta. When beta is zero then C need not be set before entry.

- **C** – [in] device array of device pointers storing each matrix C_i on the GPU. The imaginary component of the diagonal elements are not used but are set to zero unless quick return. only the upper/lower triangular part of each C_i is accessed.
- **ldc** – [in] [rocbas_int] ldc specifies the first dimension of C. $ldc \geq \max(1, n)$.
- **batch_count** – [in] [rocbas_int] number of instances in the batch.

```
rocbas_status rocbas_cherk_strided_batched(rocbas_handle handle, rocbas_fill uplo, rocbas_operation
transA, rocbas_int n, rocbas_int k, const float *alpha, const
rocbas_float_complex *A, rocbas_int lda, rocbas_stride
stride_A, const float *beta, rocbas_float_complex *C,
rocbas_int ldc, rocbas_stride stride_C, rocbas_int
batch_count)
```

```
rocbas_status rocbas_zherk_strided_batched(rocbas_handle handle, rocbas_fill uplo, rocbas_operation
transA, rocbas_int n, rocbas_int k, const double *alpha,
const rocbas_double_complex *A, rocbas_int lda,
rocbas_stride stride_A, const double *beta,
rocbas_double_complex *C, rocbas_int ldc, rocbas_stride
stride_C, rocbas_int batch_count)
```

BLAS Level 3 API

herk_strided_batched performs a batch of the matrix-matrix operations for a Hermitian rank-k update:

```
C_i := alpha*op( A_i )*op( A_i )^H + beta*C_i,
```

where **alpha** and **beta** are scalars, **op(A)** is an n by k matrix, and **C_i** is a $n \times n$ Hermitian matrix stored as either upper or lower.

```
op( A_i ) = A_i, and A_i is n by k if transA == rocbas_operation_none
op( A_i ) = A_i^H and A_i is k by n if transA == rocbas_operation_conjugate_
↪ transpose
```

Parameters

- **handle** – [in] [rocbas_handle] handle to the rocbas library context queue.
- **uplo** – [in] [rocbas_fill]
 - rocbas_fill_upper: C_i is an upper triangular matrix
 - rocbas_fill_lower: C_i is a lower triangular matrix
- **transA** – [in] [rocbas_operation]
 - rocbas_operation_conjugate_transpose: $op(A) = A^H$
 - rocbas_operation_none: $op(A) = A$
- **n** – [in] [rocbas_int] n specifies the number of rows and columns of C_i . $n \geq 0$.
- **k** – [in] [rocbas_int] k specifies the number of columns of $op(A)$. $k \geq 0$.
- **alpha** – [in] α specifies the scalar α . When α is zero then A is not referenced and A need not be set before entry.
- **A** – [in] Device pointer to the first matrix A_1 on the GPU of dimension (lda, k) when $transA$ is rocbas_operation_none, otherwise of dimension (lda, n)
- **lda** – [in] [rocbas_int] lda specifies the first dimension of A_i .

```
if transA == rocblas_operation_none, lda >= max( 1, n ),
otherwise lda >= max( 1, k ).
```

- **stride_A** – [in] [rocblas_stride] stride from the start of one matrix (A_i) and the next one (A_i+1).
- **beta** – [in] beta specifies the scalar beta. When beta is zero then C need not be set before entry.
- **C** – [in] Device pointer to the first matrix C₁ on the GPU. The imaginary component of the diagonal elements are not used but are set to zero unless quick return. only the upper/lower triangular part of each C_i is accessed.
- **ldc** – [in] [rocblas_int] ldc specifies the first dimension of C. ldc >= max(1, n).
- **stride_C** – [inout] [rocblas_stride] stride from the start of one matrix (C_i) and the next one (C_i+1).
- **batch_count** – [in] [rocblas_int] number of instances in the batch.

5.6.10 rocblas_Xher2k + batched, strided_batched

rocblas_status **rocblas_cher2k**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* trans, *rocblas_int* n, *rocblas_int* k, const *rocblas_float_complex* *alpha, const *rocblas_float_complex* *A, *rocblas_int* lda, const *rocblas_float_complex* *B, *rocblas_int* ldb, const float *beta, *rocblas_float_complex* *C, *rocblas_int* ldc)

rocblas_status **rocblas_zher2k**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* trans, *rocblas_int* n, *rocblas_int* k, const *rocblas_double_complex* *alpha, const *rocblas_double_complex* *A, *rocblas_int* lda, const *rocblas_double_complex* *B, *rocblas_int* ldb, const double *beta, *rocblas_double_complex* *C, *rocblas_int* ldc)

BLAS Level 3 API

her2k performs one of the matrix-matrix operations for a Hermitian rank-2k update:

```
C := alpha*op( A )*op( B )^H + conj(alpha)*op( B )*op( A )^H + beta*C,
```

where alpha and beta are scalars, op(A) and op(B) are n by k matrices, and C is a n x n Hermitian matrix stored as either upper or lower.

op(A) = A, op(B) = B, and A and B are n by k if trans == rocblas_operation_none
 op(A) = A^H, op(B) = B^H, and A and B are k by n if trans == rocblas_operation_conjugate_transpose

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill]
 - rocblas_fill_upper: C is an upper triangular matrix
 - rocblas_fill_lower: C is a lower triangular matrix
- **trans** – [in] [rocblas_operation]
 - rocblas_operation_conjugate_transpose: op(A) = A^H, op(B) = B^H

- `rocblas_operation_none`: $\text{op}(A) = A$, $\text{op}(B) = B$
- **n** – [in] [rocblas_int] n specifies the number of rows and columns of C. $n \geq 0$.
- **k** – [in] [rocblas_int] k specifies the number of columns of $\text{op}(A)$. $k \geq 0$.
- **alpha** – [in] alpha specifies the scalar alpha. When alpha is zero then A is not referenced and A need not be set before entry.
- **A** – [in] pointer storing matrix A on the GPU. Matrix dimension is (lda, k) when if `trans = rocblas_operation_none`, otherwise (lda, n)
- **lda** – [in] [rocblas_int] lda specifies the first dimension of A.

```
if trans == rocblas_operation_none, lda >= max( 1, n ),
otherwise lda >= max( 1, k ).
```

- **B** – [in] pointer storing matrix B on the GPU. Matrix dimension is (ldb, k) when if `trans = rocblas_operation_none`, otherwise (ldb, n)
- **ldb** – [in] [rocblas_int] ldb specifies the first dimension of B.

```
if trans == rocblas_operation_none, ldb >= max( 1, n ),
otherwise ldb >= max( 1, k ).
```

- **beta** – [in] beta specifies the scalar beta. When beta is zero then C need not be set before entry.
- **C** – [in] pointer storing matrix C on the GPU. The imaginary component of the diagonal elements are not used but are set to zero unless quick return. only the upper/lower triangular part is accessed.
- **ldc** – [in] [rocblas_int] ldc specifies the first dimension of C. $\text{ldc} \geq \max(1, n)$.

```
rocblas_status rocblas_cher2k_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_operation trans,
rocblas_int n, rocblas_int k, const rocblas_float_complex *alpha, const
rocblas_float_complex *const A[], rocblas_int lda, const
rocblas_float_complex *const B[], rocblas_int ldb, const float *beta,
rocblas_float_complex *const C[], rocblas_int ldc, rocblas_int
batch_count)
```

```
rocblas_status rocblas_zher2k_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_operation trans,
rocblas_int n, rocblas_int k, const rocblas_double_complex *alpha,
const rocblas_double_complex *const A[], rocblas_int lda, const
rocblas_double_complex *const B[], rocblas_int ldb, const double
*beta, rocblas_double_complex *const C[], rocblas_int ldc, rocblas_int
batch_count)
```

BLAS Level 3 API

`her2k_batched` performs a batch of the matrix-matrix operations for a Hermitian rank-2k update:

```
C_i := alpha*op( A_i )*op( B_i )^H + conj(alpha)*op( B_i )*op( A_i )^H + beta*C_i,
where alpha and beta are scalars, op(A_i) and op(B_i) are n by k matrices, and
C_i is a n x n Hermitian matrix stored as either upper or lower.

op( A_i ) = A_i, op( B_i ) = B_i, and A_i and B_i are n by k if trans == rocblas_
↪operation_none
```

(continues on next page)

(continued from previous page)

```
op( A_i ) = A_i^H, op( B_i ) = B_i^H,  and A_i and B_i are k by n if trans == rocblas_operation_conjugate_transpose
```

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill]
 - rocblas_fill_upper: C_i is an upper triangular matrix
 - rocblas_fill_lower: C_i is a lower triangular matrix
- **trans** – [in] [rocblas_operation]
 - rocblas_operation_conjugate_transpose: $op(A) = A^H$
 - rocblas_operation_none: $op(A) = A$
- **n** – [in] [rocblas_int] n specifies the number of rows and columns of C_i. $n \geq 0$.
- **k** – [in] [rocblas_int] k specifies the number of columns of op(A). $k \geq 0$.
- **alpha** – [in] alpha specifies the scalar alpha. When alpha is zero then A is not referenced and A need not be set before entry.
- **A** – [in] device array of device pointers storing each matrix_i A of dimension (lda, k) when trans is rocblas_operation_none, otherwise of dimension (lda, n).
- **lda** – [in] [rocblas_int] lda specifies the first dimension of A_i.

```
if trans = rocblas_operation_none,  lda >= max( 1, n ),
otherwise lda >= max( 1, k ).
```

- **B** – [in] device array of device pointers storing each matrix_i B of dimension (ldb, k) when trans is rocblas_operation_none, otherwise of dimension (ldb, n).
- **ldb** – [in] [rocblas_int] ldb specifies the first dimension of B_i.

```
if trans = rocblas_operation_none,  ldb >= max( 1, n ),
otherwise ldb >= max( 1, k ).
```

- **beta** – [in] beta specifies the scalar beta. When beta is zero then C need not be set before entry.
- **C** – [in] device array of device pointers storing each matrix C_i on the GPU. The imaginary component of the diagonal elements are not used but are set to zero unless quick return. Only the upper/lower triangular part of each C_i is accessed.
- **ldc** – [in] [rocblas_int] ldc specifies the first dimension of C. $ldc \geq \max(1, n)$.
- **batch_count** – [in] [rocblas_int] number of instances in the batch.

```
rocblas_status rocblas_cher2k_strided_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_operation
trans, rocblas_int n, rocblas_int k, const
rocblas_float_complex *alpha, const rocblas_float_complex
*A, rocblas_int lda, rocblas_stride stride_A, const
rocblas_float_complex *B, rocblas_int ldb, rocblas_stride
stride_B, const float *beta, rocblas_float_complex *C,
rocblas_int ldc, rocblas_stride stride_C, rocblas_int
batch_count)
```

```
rocblas_status rocblas_zher2k_strided_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_operation
trans, rocblas_int n, rocblas_int k, const
rocblas_double_complex *alpha, const
rocblas_double_complex *A, rocblas_int lda, rocblas_stride
stride_A, const rocblas_double_complex *B, rocblas_int ldb,
rocblas_stride stride_B, const double *beta,
rocblas_double_complex *C, rocblas_int ldc, rocblas_stride
stride_C, rocblas_int batch_count)
```

BLAS Level 3 API

her2k_strided_batched performs a batch of the matrix-matrix operations for a Hermitian rank-2k update:

```
C_i := alpha*op( A_i )*op( B_i )^H + conj(alpha)*op( B_i )*op( A_i )^H + beta*C_i,
```

where alpha and beta are scalars, op(A_i) and op(B_i) are n by k matrices, and C_i is a n x n Hermitian matrix stored as either upper or lower.

```
op( A_i ) = A_i, op( B_i ) = B_i, and A_i and B_i are n by k if trans == rocblas_
operation_none
```

```
op( A_i ) = A_i^H, op( B_i ) = B_i^H, and A_i and B_i are k by n if trans ==
rocblas_operation_conjugate_transpose
```

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill]
 - rocblas_fill_upper: C_i is an upper triangular matrix
 - rocblas_fill_lower: C_i is a lower triangular matrix
- **trans** – [in] [rocblas_operation]
 - rocblas_operation_conjugate_transpose: op(A_i) = A_i^H, op(B_i) = B_i^H
 - rocblas_operation_none: op(A_i) = A_i, op(B_i) = B_i
- **n** – [in] [rocblas_int] n specifies the number of rows and columns of C_i. n >= 0.
- **k** – [in] [rocblas_int] k specifies the number of columns of op(A). k >= 0.
- **alpha** – [in] alpha specifies the scalar alpha. When alpha is zero then A is not referenced and A need not be set before entry.
- **A** – [in] Device pointer to the first matrix A_1 on the GPU of dimension (lda, k) when trans is rocblas_operation_none, otherwise of dimension (lda, n).
- **lda** – [in] [rocblas_int] lda specifies the first dimension of A_i.

```
if trans = rocblas_operation_none, lda >= max( 1, n ),
otherwise lda >= max( 1, k ).
```

- **stride_A** – [in] [rocblas_stride] stride from the start of one matrix (A_i) and the next one (A_{i+1}).
- **B** – [in] Device pointer to the first matrix B_1 on the GPU of dimension (ldb, k) when trans is rocblas_operation_none, otherwise of dimension (ldb, n).
- **ldb** – [in] [rocblas_int] ldb specifies the first dimension of B_i.

```
if trans == rocblas_operation_none, ldb >= max( 1, n ),
otherwise ldb >= max( 1, k ).
```

- **stride_B** – [in] [rocblas_stride] stride from the start of one matrix (B_i) and the next one (B_i+1).
- **beta** – [in] beta specifies the scalar beta. When beta is zero then C need not be set before entry.
- **C** – [in] Device pointer to the first matrix C_1 on the GPU. The imaginary component of the diagonal elements are not used but are set to zero unless quick return. only the upper/lower triangular part of each C_i is accessed.
- **ldc** – [in] [rocblas_int] ldc specifies the first dimension of C. ldc >= max(1, n).
- **stride_C** – [inout] [rocblas_stride] stride from the start of one matrix (C_i) and the next one (C_i+1).
- **batch_count** – [in] [rocblas_int] number of instances in the batch.

5.6.11 rocblas_Xherkx + batched, strided_batched

rocblas_status **rocblas_cherkx**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* trans, *rocblas_int* n, *rocblas_int* k, const *rocblas_float_complex* *alpha, const *rocblas_float_complex* *A, *rocblas_int* lda, const *rocblas_float_complex* *B, *rocblas_int* ldb, const float *beta, *rocblas_float_complex* *C, *rocblas_int* ldc)

rocblas_status **rocblas_zherkx**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_operation* trans, *rocblas_int* n, *rocblas_int* k, const *rocblas_double_complex* *alpha, const *rocblas_double_complex* *A, *rocblas_int* lda, const *rocblas_double_complex* *B, *rocblas_int* ldb, const double *beta, *rocblas_double_complex* *C, *rocblas_int* ldc)

BLAS Level 3 API

herkx performs one of the matrix-matrix operations for a Hermitian rank-k update:

```
C := alpha*op( A )*op( B )^H + beta*C,
```

where alpha and beta are scalars, op(A) and op(B) are n by k matrices, and C is a n x n Hermitian matrix stored as either upper or lower.

This routine should only be used when the caller can guarantee that the result of op(A)*op(B)^T will be Hermitian.

```
op( A ) = A, op( B ) = B, and A and B are n by k if trans == rocblas_operation_none
op( A ) = A^H, op( B ) = B^H, and A and B are k by n if trans == rocblas_operation_
  ↳conjugate_transpose
```

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill]
 - rocblas_fill_upper: C is an upper triangular matrix
 - rocblas_fill_lower: C is a lower triangular matrix
- **trans** – [in] [rocblas_operation]

- `rocblas_operation_conjugate_transpose`: $\text{op}(A) = A^H$, $\text{op}(B) = B^H$
- `rocblas_operation_none`: $\text{op}(A) = A$, $\text{op}(B) = B$
- **n** – [in] [rocblas_int] n specifies the number of rows and columns of C. $n \geq 0$.
- **k** – [in] [rocblas_int] k specifies the number of columns of $\text{op}(A)$. $k \geq 0$.
- **alpha** – [in] alpha specifies the scalar alpha. When alpha is zero then A is not referenced and A need not be set before entry.
- **A** – [in] pointer storing matrix A on the GPU. Matrix dimension is (lda, k) when if `trans = rocblas_operation_none`, otherwise (lda, n)
- **lda** – [in] [rocblas_int] lda specifies the first dimension of A.

```
if trans == rocblas_operation_none, lda >= max( 1, n ),
otherwise lda >= max( 1, k ).
```

- **B** – [in] pointer storing matrix B on the GPU. Matrix dimension is (ldb, k) when if `trans = rocblas_operation_none`, otherwise (ldb, n)
- **ldb** – [in] [rocblas_int] ldb specifies the first dimension of B.

```
if trans == rocblas_operation_none, ldb >= max( 1, n ),
otherwise ldb >= max( 1, k ).
```

- **beta** – [in] beta specifies the scalar beta. When beta is zero then C need not be set before entry.
- **C** – [in] pointer storing matrix C on the GPU. The imaginary component of the diagonal elements are not used but are set to zero unless quick return. only the upper/lower triangular part is accessed.
- **ldc** – [in] [rocblas_int] ldc specifies the first dimension of C. $\text{ldc} \geq \max(1, n)$.

```
rocblas_status rocblas_cherkx_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_operation trans,
rocblas_int n, rocblas_int k, const rocblas_float_complex *alpha, const
rocblas_float_complex *const A[], rocblas_int lda, const
rocblas_float_complex *const B[], rocblas_int ldb, const float *beta,
rocblas_float_complex *const C[], rocblas_int ldc, rocblas_int
batch_count)
```

```
rocblas_status rocblas_zherkx_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_operation trans,
rocblas_int n, rocblas_int k, const rocblas_double_complex *alpha,
const rocblas_double_complex *const A[], rocblas_int lda, const
rocblas_double_complex *const B[], rocblas_int ldb, const double
*beta, rocblas_double_complex *const C[], rocblas_int ldc, rocblas_int
batch_count)
```

BLAS Level 3 API

`herkx_batched` performs a batch of the matrix-matrix operations for a Hermitian rank-k update:

```
C_i := alpha*op( A_i )*op( B_i )^H + beta*C_i,
```

where alpha and beta are scalars, $\text{op}(A_i)$ and $\text{op}(B_i)$ are n by k matrices, and C_i is a n x n Hermitian matrix stored as either upper or lower.

This routine should only be used when the caller can guarantee that the result of $\text{op}(A) * \text{op}(B)^T$ will be Hermitian.

```

op( A_i ) = A_i, op( B_i ) = B_i, and A_i and B_i are n by k if trans == rocblas_
↪operation_none
op( A_i ) = A_i^H, op( B_i ) = B_i^H, and A_i and B_i are k by n if trans ==
↪rocblas_operation_conjugate_transpose

```

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill]
 - rocblas_fill_upper: C_i is an upper triangular matrix
 - rocblas_fill_lower: C_i is a lower triangular matrix
- **trans** – [in] [rocblas_operation]
 - rocblas_operation_conjugate_transpose: $op(A) = A^H$
 - rocblas_operation_none: $op(A) = A$
- **n** – [in] [rocblas_int] n specifies the number of rows and columns of C_i. $n \geq 0$.
- **k** – [in] [rocblas_int] k specifies the number of columns of op(A). $k \geq 0$.
- **alpha** – [in] alpha specifies the scalar alpha. When alpha is zero then A is not referenced and A need not be set before entry.
- **A** – [in] device array of device pointers storing each matrix_i A of dimension (lda, k) when trans is rocblas_operation_none, otherwise of dimension (lda, n)
- **lda** – [in] [rocblas_int] lda specifies the first dimension of A_i.

```

if trans = rocblas_operation_none,  lda >= max( 1, n ),
otherwise lda >= max( 1, k ).

```

- **B** – [in] device array of device pointers storing each matrix_i B of dimension (ldb, k) when trans is rocblas_operation_none, otherwise of dimension (ldb, n)
- **ldb** – [in] [rocblas_int] ldb specifies the first dimension of B_i.

```

if trans = rocblas_operation_none,  ldb >= max( 1, n ),
otherwise ldb >= max( 1, k ).

```

- **beta** – [in] beta specifies the scalar beta. When beta is zero then C need not be set before entry.
- **C** – [in] device array of device pointers storing each matrix C_i on the GPU. The imaginary component of the diagonal elements are not used but are set to zero unless quick return. only the upper/lower triangular part of each C_i is accessed.
- **ldc** – [in] [rocblas_int] ldc specifies the first dimension of C. $ldc \geq \max(1, n)$.
- **batch_count** – [in] [rocblas_int] number of instances in the batch.

```
rocblas_status rocblas_cherkx_strided_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_operation
trans, rocblas_int n, rocblas_int k, const
rocblas_float_complex *alpha, const rocblas_float_complex
*A, rocblas_int lda, rocblas_stride stride_A, const
rocblas_float_complex *B, rocblas_int ldb, rocblas_stride
stride_B, const float *beta, rocblas_float_complex *C,
rocblas_int ldc, rocblas_stride stride_C, rocblas_int
batch_count)
```

```
rocblas_status rocblas_zherkx_strided_batched(rocblas_handle handle, rocblas_fill uplo, rocblas_operation
trans, rocblas_int n, rocblas_int k, const
rocblas_double_complex *alpha, const
rocblas_double_complex *A, rocblas_int lda, rocblas_stride
stride_A, const rocblas_double_complex *B, rocblas_int ldb,
rocblas_stride stride_B, const double *beta,
rocblas_double_complex *C, rocblas_int ldc, rocblas_stride
stride_C, rocblas_int batch_count)
```

BLAS Level 3 API

herkx_strided_batched performs a batch of the matrix-matrix operations for a Hermitian rank-k update:

$$C_i := \alpha * \text{op}(A_i) * \text{op}(B_i)^H + \beta * C_i,$$

where α and β are scalars, $\text{op}(A_i)$ and $\text{op}(B_i)$ are n by k matrices, and C_i is a $n \times n$ Hermitian matrix stored as either upper or lower.

This routine should only be used when the caller can guarantee that the result of $\text{op}(A) * \text{op}(B)^T$ will be Hermitian.

```
op( A_i ) = A_i, op( B_i ) = B_i, and A_i and B_i are n by k if trans == rocblas_
↪operation_none
op( A_i ) = A_i^H, op( B_i ) = B_i^H, and A_i and B_i are k by n if trans ==
↪rocblas_operation_conjugate_transpose
```

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill]
 - rocblas_fill_upper: C_i is an upper triangular matrix
 - rocblas_fill_lower: C_i is a lower triangular matrix
- **trans** – [in] [rocblas_operation]
 - rocblas_operation_conjugate_transpose: $\text{op}(A_i) = A_i^H$, $\text{op}(B_i) = B_i^H$
 - rocblas_operation_none: $\text{op}(A_i) = A_i$, $\text{op}(B_i) = B_i$
- **n** – [in] [rocblas_int] n specifies the number of rows and columns of C_i . $n \geq 0$.
- **k** – [in] [rocblas_int] k specifies the number of columns of $\text{op}(A)$. $k \geq 0$.
- **alpha** – [in] α specifies the scalar α . When α is zero then A is not referenced and A need not be set before entry.
- **A** – [in] Device pointer to the first matrix A_1 on the GPU of dimension (lda, k) when trans is `rocblas_operation_none`, otherwise of dimension (lda, n) .

- **lda** – [in] [rocblas_int] lda specifies the first dimension of A_i.

```
if trans == rocblas_operation_none, lda >= max( 1, n ),
otherwise lda >= max( 1, k ).
```

- **stride_A** – [in] [rocblas_stride] stride from the start of one matrix (A_i) and the next one (A_i+1)
- **B** – [in] Device pointer to the first matrix B_1 on the GPU of dimension (ldb, k) when trans is rocblas_operation_none, otherwise of dimension (ldb, n).
- **ldb** – [in] [rocblas_int] ldb specifies the first dimension of B_i.

```
if trans == rocblas_operation_none, ldb >= max( 1, n ),
otherwise ldb >= max( 1, k ).
```

- **stride_B** – [in] [rocblas_stride] stride from the start of one matrix (B_i) and the next one (B_i+1)
- **beta** – [in] beta specifies the scalar beta. When beta is zero then C need not be set before entry.
- **C** – [in] Device pointer to the first matrix C_1 on the GPU. The imaginary component of the diagonal elements are not used but are set to zero unless quick return. only the upper/lower triangular part of each C_i is accessed.
- **ldc** – [in] [rocblas_int] ldc specifies the first dimension of C. ldc >= max(1, n).
- **stride_C** – [inout] [rocblas_stride] stride from the start of one matrix (C_i) and the next one (C_i+1).
- **batch_count** – [in] [rocblas_int] number of instances in the batch.

5.6.12 rocblas_Xtrtri + batched, strided_batched

rocblas_status **rocblas_strtri**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_diagonal* diag, *rocblas_int* n, const float *A, *rocblas_int* lda, float *invA, *rocblas_int* ldinvA)

rocblas_status **rocblas_dtrtri**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_diagonal* diag, *rocblas_int* n, const double *A, *rocblas_int* lda, double *invA, *rocblas_int* ldinvA)

BLAS Level 3 API

trtri compute the inverse of a matrix A, namely, invA and write the result into invA;

if rocblas_fill_upper, the lower part of A is not referenced if rocblas_fill_lower, the upper part of A is not referenced

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill] specifies whether the upper ‘rocblas_fill_upper’ or lower ‘rocblas_fill_lower’
- **diag** – [in] [rocblas_diagonal]
 - ‘rocblas_diagonal_non_unit’, A is non-unit triangular;
 - ‘rocblas_diagonal_unit’, A is unit triangular;

- **n** – [in] [rocblas_int] size of matrix A and invA.
- **A** – [in] device pointer storing matrix A.
- **lda** – [in] [rocblas_int] specifies the leading dimension of A.
- **invA** – [out] device pointer storing matrix invA. Partial inplace operation is supported. See below: -If UPLO = ‘U’, the leading N-by-N upper triangular part of the invA will store the inverse of the upper triangular matrix, and the strictly lower triangular part of invA may be cleared.
 - If UPLO = ‘L’, the leading N-by-N lower triangular part of the invA will store the inverse of the lower triangular matrix, and the strictly upper triangular part of invA may be cleared.
- **ldinvA** – [in] [rocblas_int] specifies the leading dimension of invA.

rocblas_status **rocblas_strtri_batched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_diagonal* diag, *rocblas_int* n, const float *const A[], *rocblas_int* lda, float *const invA[], *rocblas_int* ldinvA, *rocblas_int* batch_count)

rocblas_status **rocblas_dtrtri_batched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_diagonal* diag, *rocblas_int* n, const double *const A[], *rocblas_int* lda, double *const invA[], *rocblas_int* ldinvA, *rocblas_int* batch_count)

BLAS Level 3 API

trtri_batched compute the inverse of A_i and write into invA_i where A_i and invA_i are the i-th matrices in the batch, for i = 1, ..., batch_count.

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill] specifies whether the upper ‘rocblas_fill_upper’ or lower ‘rocblas_fill_lower’
- **diag** – [in] [rocblas_diagonal]
 - ‘rocblas_diagonal_non_unit’, A is non-unit triangular;
 - ‘rocblas_diagonal_unit’, A is unit triangular;
- **n** – [in] [rocblas_int]
- **A** – [in] device array of device pointers storing each matrix A_i.
- **lda** – [in] [rocblas_int] specifies the leading dimension of each A_i.
- **invA** – [out] device array of device pointers storing the inverse of each matrix A_i. Partial inplace operation is supported. See below: -If UPLO = ‘U’, the leading N-by-N upper triangular part of the invA will store the inverse of the upper triangular matrix, and the strictly lower triangular part of invA may be cleared.
 - If UPLO = ‘L’, the leading N-by-N lower triangular part of the invA will store the inverse of the lower triangular matrix, and the strictly upper triangular part of invA may be cleared.
- **ldinvA** – [in] [rocblas_int] specifies the leading dimension of each invA_i.
- **batch_count** – [in] [rocblas_int] numbers of matrices in the batch.

rocblas_status **rocblas_strtri_strided_batched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_diagonal* diag, *rocblas_int* n, const float *A, *rocblas_int* lda, *rocblas_stride* stride_a, float *invA, *rocblas_int* ldinvA, *rocblas_stride* stride_invA, *rocblas_int* batch_count)

rocblas_status **rocblas_dtrtri_strided_batched**(*rocblas_handle* handle, *rocblas_fill* uplo, *rocblas_diagonal* diag, *rocblas_int* n, const double *A, *rocblas_int* lda, *rocblas_stride* stride_a, double *invA, *rocblas_int* ldinvA, *rocblas_stride* stride_invA, *rocblas_int* batch_count)

BLAS Level 3 API

trtri_strided_batched compute the inverse of A_i and write into invA_i where A_i and invA_i are the i-th matrices in the batch, for i = 1, ..., batch_count.

- If UPLO = 'U', the leading N-by-N upper triangular part of the invA will store the inverse of the upper triangular matrix, and the strictly lower triangular part of invA may be cleared.
- If UPLO = 'L', the leading N-by-N lower triangular part of the invA will store the inverse of the lower triangular matrix, and the strictly upper triangular part of invA may be cleared.

Parameters

- **ldinvA** – [in] [rocblas_int] specifies the leading dimension of each invA_i.
- **stride_invA** – [in] [rocblas_stride] “batch stride invA”: stride from the start of one invA_i matrix to the next invA_(i + 1).
- **batch_count** – [in] [rocblas_int] numbers of matrices in the batch.
- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **uplo** – [in] [rocblas_fill] specifies whether the upper ‘rocblas_fill_upper’ or lower ‘rocblas_fill_lower’
- **diag** – [in] [rocblas_diagonal]
 - ‘rocblas_diagonal_non_unit’, A is non-unit triangular;
 - ‘rocblas_diagonal_unit’, A is unit triangular;
- **n** – [in] [rocblas_int]
- **A** – [in] device pointer pointing to address of first matrix A₁.
- **lda** – [in] [rocblas_int] specifies the leading dimension of each A.
- **stride_a** – [in] [rocblas_stride] “batch stride a”: stride from the start of one A_i matrix to the next A_(i + 1).
- **invA** – [out] device pointer storing the inverses of each matrix A_i. Partial inplace operation is supported. See below:

5.7 rocBLAS Extension

5.7.1 rocblas_axpy_ex + batched, strided_batched

rocblas_status **rocblas_axpy_ex**(*rocblas_handle* handle, *rocblas_int* n, const void *alpha, *rocblas_datatype* alpha_type, const void *x, *rocblas_datatype* x_type, *rocblas_int* incx, void *y, *rocblas_datatype* y_type, *rocblas_int* incy, *rocblas_datatype* execution_type)

BLAS EX API

axpy_ex computes constant alpha multiplied by vector x, plus vector y.

```
y := alpha * x + y
```

Currently supported datatypes are as follows:

alpha_type	x_type	y_type	execution_type
bf16_r	bf16_r	bf16_r	f32_r
f32_r	bf16_r	bf16_r	f32_r
f16_r	f16_r	f16_r	f16_r
f16_r	f16_r	f16_r	f32_r
f32_r	f16_r	f16_r	f32_r
f32_r	f32_r	f32_r	f32_r
f64_r	f64_r	f64_r	f64_r
f32_c	f32_c	f32_c	f32_c
f64_c	f64_c	f64_c	f64_c

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **n** – [in] [rocblas_int] the number of elements in x and y.
- **alpha** – [in] device pointer or host pointer to specify the scalar alpha.
- **alpha_type** – [in] [rocblas_datatype] specifies the datatype of alpha.
- **x** – [in] device pointer storing vector x.
- **x_type** – [in] [rocblas_datatype] specifies the datatype of vector x.
- **incx** – [in] [rocblas_int] specifies the increment for the elements of x.
- **y** – [inout] device pointer storing vector y.
- **y_type** – [in] [rocblas_datatype] specifies the datatype of vector y.
- **incy** – [in] [rocblas_int] specifies the increment for the elements of y.
- **execution_type** – [in] [rocblas_datatype] specifies the datatype of computation.

```
rocblas_status rocblas_axpy_batched_ex(rocblas_handle handle, rocblas_int n, const void *alpha,
                                       rocblas_datatype alpha_type, const void *x, rocblas_datatype x_type,
                                       rocblas_int incx, void *y, rocblas_datatype y_type, rocblas_int incy,
                                       rocblas_int batch_count, rocblas_datatype execution_type)
```

BLAS EX API

axpy_batched_ex computes constant alpha multiplied by vector x, plus vector y over a set of batched vectors.

```
y := alpha * x + y
```

Currently supported datatypes are as follows:

alpha_type	x_type	y_type	execution_type
bf16_r	bf16_r	bf16_r	f32_r
f32_r	bf16_r	bf16_r	f32_r
f16_r	f16_r	f16_r	f16_r
f16_r	f16_r	f16_r	f32_r
f32_r	f16_r	f16_r	f32_r
f32_r	f32_r	f32_r	f32_r
f64_r	f64_r	f64_r	f64_r
f32_c	f32_c	f32_c	f32_c
f64_c	f64_c	f64_c	f64_c

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **n** – [in] [rocblas_int] the number of elements in each x_i and y_i .
- **alpha** – [in] device pointer or host pointer to specify the scalar alpha.
- **alpha_type** – [in] [rocblas_datatype] specifies the datatype of alpha.
- **x** – [in] device array of device pointers storing each vector x_i .
- **x_type** – [in] [rocblas_datatype] specifies the datatype of each vector x_i .
- **incx** – [in] [rocblas_int] specifies the increment for the elements of each x_i .
- **y** – [inout] device array of device pointers storing each vector y_i .
- **y_type** – [in] [rocblas_datatype] specifies the datatype of each vector y_i .
- **incy** – [in] [rocblas_int] specifies the increment for the elements of each y_i .
- **batch_count** – [in] [rocblas_int] number of instances in the batch.
- **execution_type** – [in] [rocblas_datatype] specifies the datatype of computation.

rocblas_status rocblas_axpy_strided_batched_ex(*rocblas_handle* handle, *rocblas_int* n, const void *alpha, *rocblas_datatype* alpha_type, const void *x, *rocblas_datatype* x_type, *rocblas_int* incx, *rocblas_stride* stridex, void *y, *rocblas_datatype* y_type, *rocblas_int* incy, *rocblas_stride* stridey, *rocblas_int* batch_count, *rocblas_datatype* execution_type)

BLAS EX API

axpy_strided_batched_ex computes constant alpha multiplied by vector x, plus vector y over a set of strided batched vectors.

```
y := alpha * x + y
```

Currently supported datatypes are as follows:

alpha_type	x_type	y_type	execution_type
bf16_r	bf16_r	bf16_r	f32_r
f32_r	bf16_r	bf16_r	f32_r
f16_r	f16_r	f16_r	f16_r
f16_r	f16_r	f16_r	f32_r
f32_r	f16_r	f16_r	f32_r
f32_r	f32_r	f32_r	f32_r
f64_r	f64_r	f64_r	f64_r
f32_c	f32_c	f32_c	f32_c
f64_c	f64_c	f64_c	f64_c

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **n** – [in] [rocblas_int] the number of elements in each x_i and y_i .
- **alpha** – [in] device pointer or host pointer to specify the scalar alpha.
- **alpha_type** – [in] [rocblas_datatype] specifies the datatype of alpha.
- **x** – [in] device pointer to the first vector x_1 .
- **x_type** – [in] [rocblas_datatype] specifies the datatype of each vector x_i .
- **incx** – [in] [rocblas_int] specifies the increment for the elements of each x_i .
- **stridex** – [in] [rocblas_stride] stride from the start of one vector (x_i) to the next one (x_{i+1}). There are no restrictions placed on stridex. However, ensure that stridex is of appropriate size. For a typical case this means $\text{stridex} \geq n * \text{incx}$.
- **y** – [inout] device pointer to the first vector y_1 .
- **y_type** – [in] [rocblas_datatype] specifies the datatype of each vector y_i .
- **incy** – [in] [rocblas_int] specifies the increment for the elements of each y_i .
- **stridey** – [in] [rocblas_stride] stride from the start of one vector (y_i) to the next one (y_{i+1}). There are no restrictions placed on stridey. However, ensure that stridey is of appropriate size. For a typical case this means $\text{stridey} \geq n * \text{incy}$.
- **batch_count** – [in] [rocblas_int] number of instances in the batch.
- **execution_type** – [in] [rocblas_datatype] specifies the datatype of computation.

5.7.2 rocblas_dot_ex + batched, strided_batched

rocblas_status **rocblas_dot_ex**(*rocblas_handle* handle, *rocblas_int* n, const void *x, *rocblas_datatype* x_type, *rocblas_int* incx, const void *y, *rocblas_datatype* y_type, *rocblas_int* incy, void *result, *rocblas_datatype* result_type, *rocblas_datatype* execution_type)

BLAS EX API

dot_ex performs the dot product of vectors x and y.

```
result = x * y;
```

dotc_ex performs the dot product of the conjugate of complex vector x and complex vector y

```
result = conjugate (x) * y;
```

Currently supported datatypes are as follows:

x_type	y_type	result_type	execution_type
f16_r	f16_r	f16_r	f16_r
f16_r	f16_r	f16_r	f32_r
bf16_r	bf16_r	bf16_r	f32_r
f32_r	f32_r	f32_r	f32_r
f64_r	f64_r	f64_r	f64_r
f32_c	f32_c	f32_c	f32_c
f64_c	f64_c	f64_c	f64_c

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **n** – [in] [rocblas_int] the number of elements in x and y.
- **x** – [in] device pointer storing vector x.
- **x_type** – [in] [rocblas_datatype] specifies the datatype of vector x.
- **incx** – [in] [rocblas_int] specifies the increment for the elements of y.
- **y** – [in] device pointer storing vector y.
- **y_type** – [in] [rocblas_datatype] specifies the datatype of vector y.
- **incy** – [in] [rocblas_int] specifies the increment for the elements of y.
- **result** – [inout] device pointer or host pointer to store the dot product. return is 0.0 if n <= 0.
- **result_type** – [in] [rocblas_datatype] specifies the datatype of the result.
- **execution_type** – [in] [rocblas_datatype] specifies the datatype of computation.

rocblas_status **rocblas_dot_batched_ex**(*rocblas_handle* handle, *rocblas_int* n, const void *x, *rocblas_datatype* x_type, *rocblas_int* incx, const void *y, *rocblas_datatype* y_type, *rocblas_int* incy, *rocblas_int* batch_count, void *result, *rocblas_datatype* result_type, *rocblas_datatype* execution_type)

BLAS EX API

dot_batched_ex performs a batch of dot products of vectors x and y.

```
result_i = x_i * y_i;
```

dotc_batched_ex performs a batch of dot products of the conjugate of complex vector x and complex vector y

```
result_i = conjugate (x_i) * y_i;
```

where (x_i, y_i) is the i-th instance of the batch. x_i and y_i are vectors, for i = 1, ..., batch_count

Currently supported datatypes are as follows:

x_type	y_type	result_type	execution_type
f16_r	f16_r	f16_r	f16_r
f16_r	f16_r	f16_r	f32_r
bf16_r	bf16_r	bf16_r	f32_r
f32_r	f32_r	f32_r	f32_r
f64_r	f64_r	f64_r	f64_r
f32_c	f32_c	f32_c	f32_c
f64_c	f64_c	f64_c	f64_c

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **n** – [in] [rocblas_int] the number of elements in each x_i and y_i .
- **x** – [in] device array of device pointers storing each vector x_i .
- **x_type** – [in] [rocblas_datatype] specifies the datatype of each vector x_i .
- **incx** – [in] [rocblas_int] specifies the increment for the elements of each x_i .
- **y** – [in] device array of device pointers storing each vector y_i .
- **y_type** – [in] [rocblas_datatype] specifies the datatype of each vector y_i .
- **incy** – [in] [rocblas_int] specifies the increment for the elements of each y_i .
- **batch_count** – [in] [rocblas_int] number of instances in the batch.
- **result** – [inout] device array or host array of batch_count size to store the dot products of each batch. return 0.0 for each element if $n \leq 0$.
- **result_type** – [in] [rocblas_datatype] specifies the datatype of the result.
- **execution_type** – [in] [rocblas_datatype] specifies the datatype of computation.

rocblas_status rocblas_dot_strided_batched_ex(*rocblas_handle* handle, *rocblas_int* n, const void *x, *rocblas_datatype* x_type, *rocblas_int* incx, *rocblas_stride* stride_x, const void *y, *rocblas_datatype* y_type, *rocblas_int* incy, *rocblas_stride* stride_y, *rocblas_int* batch_count, void *result, *rocblas_datatype* result_type, *rocblas_datatype* execution_type)

BLAS EX API

dot_strided_batched_ex performs a batch of dot products of vectors x and y .

```
result_i = x_i * y_i;
```

dotc_strided_batched_ex performs a batch of dot products of the conjugate of complex vector x and complex vector y

```
result_i = conjugate (x_i) * y_i;
```

where (x_i, y_i) is the i -th instance of the batch. x_i and y_i are vectors, for $i = 1, \dots, \text{batch_count}$

Currently supported datatypes are as follows:

x_type	y_type	result_type	execution_type
f16_r	f16_r	f16_r	f16_r
f16_r	f16_r	f16_r	f32_r
bf16_r	bf16_r	bf16_r	f32_r
f32_r	f32_r	f32_r	f32_r
f64_r	f64_r	f64_r	f64_r
f32_c	f32_c	f32_c	f32_c
f64_c	f64_c	f64_c	f64_c

Parameters

- **handle** – [in] [rocbblas_handle] handle to the rocbblas library context queue.
- **n** – [in] [rocbblas_int] the number of elements in each x_i and y_i .
- **x** – [in] device pointer to the first vector (x_1) in the batch.
- **x_type** – [in] [rocbblas_datatype] specifies the datatype of each vector x_i .
- **incx** – [in] [rocbblas_int] specifies the increment for the elements of each x_i .
- **stride_x** – [in] [rocbblas_stride] stride from the start of one vector (x_i) and the next one (x_{i+1})
- **y** – [in] device pointer to the first vector (y_1) in the batch.
- **y_type** – [in] [rocbblas_datatype] specifies the datatype of each vector y_i .
- **incy** – [in] [rocbblas_int] specifies the increment for the elements of each y_i .
- **stride_y** – [in] [rocbblas_stride] stride from the start of one vector (y_i) and the next one (y_{i+1})
- **batch_count** – [in] [rocbblas_int] number of instances in the batch.
- **result** – [inout] device array or host array of batch_count size to store the dot products of each batch. return 0.0 for each element if $n \leq 0$.
- **result_type** – [in] [rocbblas_datatype] specifies the datatype of the result.
- **execution_type** – [in] [rocbblas_datatype] specifies the datatype of computation.

5.7.3 rocbblas_dotc_ex + batched, strided_batched

rocbblas_status **rocbblas_dotc_ex**(*rocbblas_handle* handle, *rocbblas_int* n, const void *x, *rocbblas_datatype* x_type, *rocbblas_int* incx, const void *y, *rocbblas_datatype* y_type, *rocbblas_int* incy, void *result, *rocbblas_datatype* result_type, *rocbblas_datatype* execution_type)

BLAS EX API

dot_ex performs the dot product of vectors x and y.

```
result = x * y;
```

dotc_ex performs the dot product of the conjugate of complex vector x and complex vector y

```
result = conjugate (x) * y;
```


Currently supported datatypes are as follows:

x_type	y_type	result_type	execution_type
f16_r	f16_r	f16_r	f16_r
f16_r	f16_r	f16_r	f32_r
bf16_r	bf16_r	bf16_r	f32_r
f32_r	f32_r	f32_r	f32_r
f64_r	f64_r	f64_r	f64_r
f32_c	f32_c	f32_c	f32_c
f64_c	f64_c	f64_c	f64_c

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **n** – [in] [rocblas_int] the number of elements in x and y.
- **x** – [in] device pointer storing vector x.
- **x_type** – [in] [rocblas_datatype] specifies the datatype of vector x.
- **incx** – [in] [rocblas_int] specifies the increment for the elements of y.
- **y** – [in] device pointer storing vector y.
- **y_type** – [in] [rocblas_datatype] specifies the datatype of vector y.
- **incy** – [in] [rocblas_int] specifies the increment for the elements of y.
- **result** – [inout] device pointer or host pointer to store the dot product. return is 0.0 if n <= 0.
- **result_type** – [in] [rocblas_datatype] specifies the datatype of the result.
- **execution_type** – [in] [rocblas_datatype] specifies the datatype of computation.

rocblas_status rocblas_dotc_batched_ex(*rocblas_handle* handle, *rocblas_int* n, const void *x, *rocblas_datatype* x_type, *rocblas_int* incx, const void *y, *rocblas_datatype* y_type, *rocblas_int* incy, *rocblas_int* batch_count, void *result, *rocblas_datatype* result_type, *rocblas_datatype* execution_type)

BLAS EX API

dot_batched_ex performs a batch of dot products of vectors x and y.

```
result_i = x_i * y_i;
```

dotc_batched_ex performs a batch of dot products of the conjugate of complex vector x and complex vector y

```
result_i = conjugate (x_i) * y_i;
```

where (x_i, y_i) is the i-th instance of the batch. x_i and y_i are vectors, for i = 1, ..., batch_count

Currently supported datatypes are as follows:

x_type	y_type	result_type	execution_type
f16_r	f16_r	f16_r	f16_r
f16_r	f16_r	f16_r	f32_r
bf16_r	bf16_r	bf16_r	f32_r
f32_r	f32_r	f32_r	f32_r
f64_r	f64_r	f64_r	f64_r
f32_c	f32_c	f32_c	f32_c
f64_c	f64_c	f64_c	f64_c

Parameters

- **handle** – [in] [rocbblas_handle] handle to the rocbblas library context queue.
- **n** – [in] [rocbblas_int] the number of elements in each x_i and y_i .
- **x** – [in] device array of device pointers storing each vector x_i .
- **x_type** – [in] [rocbblas_datatype] specifies the datatype of each vector x_i .
- **incx** – [in] [rocbblas_int] specifies the increment for the elements of each x_i .
- **y** – [in] device array of device pointers storing each vector y_i .
- **y_type** – [in] [rocbblas_datatype] specifies the datatype of each vector y_i .
- **incy** – [in] [rocbblas_int] specifies the increment for the elements of each y_i .
- **batch_count** – [in] [rocbblas_int] number of instances in the batch.
- **result** – [inout] device array or host array of batch_count size to store the dot products of each batch. return 0.0 for each element if $n \leq 0$.
- **result_type** – [in] [rocbblas_datatype] specifies the datatype of the result.
- **execution_type** – [in] [rocbblas_datatype] specifies the datatype of computation.

rocbblas_status rocbblas_dotc_strided_batched_ex(*rocbblas_handle* handle, *rocbblas_int* n, const void *x, *rocbblas_datatype* x_type, *rocbblas_int* incx, *rocbblas_stride* stride_x, const void *y, *rocbblas_datatype* y_type, *rocbblas_int* incy, *rocbblas_stride* stride_y, *rocbblas_int* batch_count, void *result, *rocbblas_datatype* result_type, *rocbblas_datatype* execution_type)

BLAS EX API

dot_strided_batched_ex performs a batch of dot products of vectors x and y .

```
result_i = x_i * y_i;
```

dotc_strided_batched_ex performs a batch of dot products of the conjugate of complex vector x and complex vector y

```
result_i = conjugate (x_i) * y_i;
```

where (x_i, y_i) is the i -th instance of the batch. x_i and y_i are vectors, for $i = 1, \dots, \text{batch_count}$

Currently supported datatypes are as follows:

x_type	y_type	result_type	execution_type
f16_r	f16_r	f16_r	f16_r
f16_r	f16_r	f16_r	f32_r
bf16_r	bf16_r	bf16_r	f32_r
f32_r	f32_r	f32_r	f32_r
f64_r	f64_r	f64_r	f64_r
f32_c	f32_c	f32_c	f32_c
f64_c	f64_c	f64_c	f64_c

Parameters

- **handle** – [in] [rocbblas_handle] handle to the rocbblas library context queue.
- **n** – [in] [rocbblas_int] the number of elements in each x_i and y_i .
- **x** – [in] device pointer to the first vector (x_1) in the batch.
- **x_type** – [in] [rocbblas_datatype] specifies the datatype of each vector x_i .
- **incx** – [in] [rocbblas_int] specifies the increment for the elements of each x_i .
- **stride_x** – [in] [rocbblas_stride] stride from the start of one vector (x_i) and the next one (x_{i+1})
- **y** – [in] device pointer to the first vector (y_1) in the batch.
- **y_type** – [in] [rocbblas_datatype] specifies the datatype of each vector y_i .
- **incy** – [in] [rocbblas_int] specifies the increment for the elements of each y_i .
- **stride_y** – [in] [rocbblas_stride] stride from the start of one vector (y_i) and the next one (y_{i+1})
- **batch_count** – [in] [rocbblas_int] number of instances in the batch.
- **result** – [inout] device array or host array of batch_count size to store the dot products of each batch. return 0.0 for each element if $n \leq 0$.
- **result_type** – [in] [rocbblas_datatype] specifies the datatype of the result.
- **execution_type** – [in] [rocbblas_datatype] specifies the datatype of computation.

5.7.4 rocbblas_nrm2_ex + batched, strided_batched

rocbblas_status **rocbblas_nrm2_ex**(*rocbblas_handle* handle, *rocbblas_int* n, const void *x, *rocbblas_datatype* x_type, *rocbblas_int* incx, void *results, *rocbblas_datatype* result_type, *rocbblas_datatype* execution_type)

BLAS_EX API.

nrm2_ex computes the euclidean norm of a real or complex vector.

```
result := sqrt( x'*x ) for real vectors
result := sqrt( x**H*x ) for complex vectors
```

Currently supported datatypes are as follows:

x_type	result	execution_type
bf16_r	bf16_r	f32_r
f16_r	f16_r	f32_r
f32_r	f32_r	f32_r
f64_r	f64_r	f64_r
f32_c	f32_r	f32_r
f64_c	f64_r	f64_r

Parameters

- **handle** – [in] [roclblas_handle] handle to the roclblas library context queue.
- **n** – [in] [roclblas_int] the number of elements in x.
- **x** – [in] device pointer storing vector x.
- **x_type** – [in] [roclblas_datatype] specifies the datatype of the vector x.
- **incx** – [in] [roclblas_int] specifies the increment for the elements of y.
- **results** – [inout] device pointer or host pointer to store the nrm2 product. return is 0.0 if n, incx<=0.
- **result_type** – [in] [roclblas_datatype] specifies the datatype of the result.
- **execution_type** – [in] [roclblas_datatype] specifies the datatype of computation.

roclblas_status **roclblas_nrm2_batched_ex**(*roclblas_handle* handle, *roclblas_int* n, const void *x, *roclblas_datatype* x_type, *roclblas_int* incx, *roclblas_int* batch_count, void *results, *roclblas_datatype* result_type, *roclblas_datatype* execution_type)

BLAS_EX API.

nrm2_batched_ex computes the euclidean norm over a batch of real or complex vectors.

```
result := sqrt( x_i'*x_i ) for real vectors x, for i = 1, ..., batch_count
result := sqrt( x_i**H*x_i ) for complex vectors x, for i = 1, ..., batch_
↪count
```

Currently supported datatypes are as follows:

x_type	result	execution_type
bf16_r	bf16_r	f32_r
f16_r	f16_r	f32_r
f32_r	f32_r	f32_r
f64_r	f64_r	f64_r
f32_c	f32_r	f32_r
f64_c	f64_r	f64_r

Parameters

- **handle** – [in] [roclblas_handle] handle to the roclblas library context queue.
- **n** – [in] [roclblas_int] number of elements in each x_i.
- **x** – [in] device array of device pointers storing each vector x_i.
- **x_type** – [in] [roclblas_datatype] specifies the datatype of each vector x_i.

- **incx** – [in] [rocbblas_int] specifies the increment for the elements of each x_i . incx must be > 0 .
- **batch_count** – [in] [rocbblas_int] number of instances in the batch.
- **results** – [out] device pointer or host pointer to array of batch_count size for nrm2 results. return is 0.0 for each element if $n \leq 0$, $incx \leq 0$.
- **result_type** – [in] [rocbblas_datatype] specifies the datatype of the result.
- **execution_type** – [in] [rocbblas_datatype] specifies the datatype of computation.

rocbblas_status rocbblas_nrm2_strided_batched_ex(*rocbblas_handle* handle, *rocbblas_int* n, const void *x, *rocbblas_datatype* x_type, *rocbblas_int* incx, *rocbblas_stride* stride_x, *rocbblas_int* batch_count, void *results, *rocbblas_datatype* result_type, *rocbblas_datatype* execution_type)

BLAS_EX API.

nrm2_strided_batched_ex computes the euclidean norm over a batch of real or complex vectors.

```
result := sqrt( x_i'*x_i ) for real vectors x, for i = 1, ..., batch_count
result := sqrt( x_i**H*x_i ) for complex vectors, for i = 1, ..., batch_count
```

Currently supported datatypes are as follows:

x_type	result	execution_type
bf16_r	bf16_r	f32_r
f16_r	f16_r	f32_r
f32_r	f32_r	f32_r
f64_r	f64_r	f64_r
f32_c	f32_r	f32_r
f64_c	f64_r	f64_r

Parameters

- **handle** – [in] [rocbblas_handle] handle to the rocbblas library context queue.
- **n** – [in] [rocbblas_int] number of elements in each x_i .
- **x** – [in] device pointer to the first vector x_1 .
- **x_type** – [in] [rocbblas_datatype] specifies the datatype of each vector x_i .
- **incx** – [in] [rocbblas_int] specifies the increment for the elements of each x_i . incx must be > 0 .
- **stride_x** – [in] [rocbblas_stride] stride from the start of one vector (x_i) and the next one (x_{i+1}). There are no restrictions placed on stride_x. However, ensure that stride_x is of appropriate size. For a typical case this means $stride_x \geq n * incx$.
- **batch_count** – [in] [rocbblas_int] number of instances in the batch.
- **results** – [out] device pointer or host pointer to array for storing contiguous batch_count results. return is 0.0 for each element if $n \leq 0$, $incx \leq 0$.
- **result_type** – [in] [rocbblas_datatype] specifies the datatype of the result.
- **execution_type** – [in] [rocbblas_datatype] specifies the datatype of computation.

5.7.5 rocblas_rot_ex + batched, strided_batched

rocblas_status **rocblas_rot_ex**(*rocblas_handle* handle, *rocblas_int* n, void *x, *rocblas_datatype* x_type, *rocblas_int* incx, void *y, *rocblas_datatype* y_type, *rocblas_int* incy, const void *c, const void *s, *rocblas_datatype* cs_type, *rocblas_datatype* execution_type)

BLAS EX API

rot_ex applies the Givens rotation matrix defined by $c=\cos(\alpha)$ and $s=\sin(\alpha)$ to vectors x and y. Scalars c and s may be stored in either host or device memory. Location is specified by calling rocblas_set_pointer_mode.

In the case where cs_type is real:

$$\begin{aligned} x &:= c * x + s * y \\ y &:= c * y - s * x \end{aligned}$$

In the case where cs_type is complex, the imaginary part of c is ignored:

$$\begin{aligned} x &:= \text{real}(c) * x + s * y \\ y &:= \text{real}(c) * y - \text{conj}(s) * x \end{aligned}$$

Currently supported datatypes are as follows:

x_type	y_type	cs_type	execution_type
bf16_r	bf16_r	bf16_r	f32_r
f16_r	f16_r	f16_r	f32_r
f32_r	f32_r	f32_r	f32_r
f64_r	f64_r	f64_r	f64_r
f32_c	f32_c	f32_c	f32_c
f32_c	f32_c	f32_r	f32_c
f64_c	f64_c	f64_c	f64_c
f64_c	f64_c	f64_r	f64_c

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **n** – [in] [rocblas_int] number of elements in the x and y vectors.
- **x** – [inout] device pointer storing vector x.
- **x_type** – [in] [rocblas_datatype] specifies the datatype of vector x.
- **incx** – [in] [rocblas_int] specifies the increment between elements of x.
- **y** – [inout] device pointer storing vector y.
- **y_type** – [in] [rocblas_datatype] specifies the datatype of vector y.
- **incy** – [in] [rocblas_int] specifies the increment between elements of y.
- **c** – [in] device pointer or host pointer storing scalar cosine component of the rotation matrix.
- **s** – [in] device pointer or host pointer storing scalar sine component of the rotation matrix.
- **cs_type** – [in] [rocblas_datatype] specifies the datatype of c and s.
- **execution_type** – [in] [rocblas_datatype] specifies the datatype of computation.

rocblas_status **rocblas_rot_batched_ex**(*rocblas_handle* handle, *rocblas_int* n, void *x, *rocblas_datatype* x_type, *rocblas_int* incx, void *y, *rocblas_datatype* y_type, *rocblas_int* incy, const void *c, const void *s, *rocblas_datatype* cs_type, *rocblas_int* batch_count, *rocblas_datatype* execution_type)

BLAS EX API

rot_batched_ex applies the Givens rotation matrix defined by $c=\cos(\alpha)$ and $s=\sin(\alpha)$ to batched vectors x_i and y_i , for $i = 1, \dots, \text{batch_count}$. Scalars c and s may be stored in either host or device memory. Location is specified by calling rocblas_set_pointer_mode.

In the case where cs_type is real:

$$\begin{aligned} x &:= c * x + s * y \\ y &:= c * y - s * x \end{aligned}$$

In the case where cs_type is complex, the imaginary part of c is ignored:

$$\begin{aligned} x &:= \text{real}(c) * x + s * y \\ y &:= \text{real}(c) * y - \text{conj}(s) * x \end{aligned}$$

Currently supported datatypes are as follows:

x_type	y_type	cs_type	execution_type
bf16_r	bf16_r	bf16_r	f32_r
f16_r	f16_r	f16_r	f32_r
f32_r	f32_r	f32_r	f32_r
f64_r	f64_r	f64_r	f64_r
f32_c	f32_c	f32_c	f32_c
f32_c	f32_c	f32_r	f32_c
f64_c	f64_c	f64_c	f64_c
f64_c	f64_c	f64_r	f64_c

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **n** – [in] [rocblas_int] number of elements in each x_i and y_i vectors.
- **x** – [inout] device array of device pointers storing each vector x_i .
- **x_type** – [in] [rocblas_datatype] specifies the datatype of each vector x_i .
- **incx** – [in] [rocblas_int] specifies the increment between elements of each x_i .
- **y** – [inout] device array of device pointers storing each vector y_i .
- **y_type** – [in] [rocblas_datatype] specifies the datatype of each vector y_i .
- **incy** – [in] [rocblas_int] specifies the increment between elements of each y_i .
- **c** – [in] device pointer or host pointer to scalar cosine component of the rotation matrix.
- **s** – [in] device pointer or host pointer to scalar sine component of the rotation matrix.
- **cs_type** – [in] [rocblas_datatype] specifies the datatype of c and s .
- **batch_count** – [in] [rocblas_int] the number of x and y arrays, the number of batches.
- **execution_type** – [in] [rocblas_datatype] specifies the datatype of computation.

rocblas_status **rocblas_rot_strided_batched_ex**(*rocblas_handle* handle, *rocblas_int* n, void *x, *rocblas_datatype* x_type, *rocblas_int* incx, *rocblas_stride* stride_x, void *y, *rocblas_datatype* y_type, *rocblas_int* incy, *rocblas_stride* stride_y, const void *c, const void *s, *rocblas_datatype* cs_type, *rocblas_int* batch_count, *rocblas_datatype* execution_type)

BLAS Level 1 API

rot_strided_batched_ex applies the Givens rotation matrix defined by $c=\cos(\alpha)$ and $s=\sin(\alpha)$ to strided batched vectors x_i and y_i , for $i = 1, \dots, \text{batch_count}$. Scalars c and s may be stored in either host or device memory. Location is specified by calling rocblas_set_pointer_mode.

In the case where cs_type is real:

```
x := c * x + s * y
y := c * y - s * x
```

In the case where cs_type is complex, the imaginary part of c is ignored:

```
x := real(c) * x + s * y
y := real(c) * y - conj(s) * x
```

Currently supported datatypes are as follows:

x_type	y_type	cs_type	execution_type
bf16_r	bf16_r	bf16_r	f32_r
f16_r	f16_r	f16_r	f32_r
f32_r	f32_r	f32_r	f32_r
f64_r	f64_r	f64_r	f64_r
f32_c	f32_c	f32_c	f32_c
f32_c	f32_c	f32_r	f32_c
f64_c	f64_c	f64_c	f64_c
f64_c	f64_c	f64_r	f64_c

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **n** – [in] [rocblas_int] number of elements in each x_i and y_i vectors.
- **x** – [inout] device pointer to the first vector x_1 .
- **x_type** – [in] [rocblas_datatype] specifies the datatype of each vector x_i .
- **incx** – [in] [rocblas_int] specifies the increment between elements of each x_i .
- **stride_x** – [in] [rocblas_stride] specifies the increment from the beginning of x_i to the beginning of x_{i+1} .
- **y** – [inout] device pointer to the first vector y_1 .
- **y_type** – [in] [rocblas_datatype] specifies the datatype of each vector y_i .
- **incy** – [in] [rocblas_int] specifies the increment between elements of each y_i .
- **stride_y** – [in] [rocblas_stride] specifies the increment from the beginning of y_i to the beginning of y_{i+1} .
- **c** – [in] device pointer or host pointer to scalar cosine component of the rotation matrix.

- **s** – [in] device pointer or host pointer to scalar sine component of the rotation matrix.
- **cs_type** – [in] [rocblas_datatype] specifies the datatype of c and s.
- **batch_count** – [in] [rocblas_int] the number of x and y arrays, the number of batches.
- **execution_type** – [in] [rocblas_datatype] specifies the datatype of computation.

5.7.6 rocblas_scal_ex + batched, strided_batched

rocblas_status **rocblas_scal_ex**(*rocblas_handle* handle, *rocblas_int* n, const void *alpha, *rocblas_datatype* alpha_type, void *x, *rocblas_datatype* x_type, *rocblas_int* incx, *rocblas_datatype* execution_type)

BLAS EX API

scal_ex scales each element of vector x with scalar alpha.

$$x := \alpha * x$$

Currently supported datatypes are as follows:

alpha_type	x_type	execution_type
f32_r	bf16_r	f32_r
bf16_r	bf16_r	f32_r
f16_r	f16_r	f16_r
f16_r	f16_r	f32_r
f32_r	f16_r	f32_r
f32_r	f32_r	f32_r
f64_r	f64_r	f64_r
f32_c	f32_c	f32_c
f64_c	f64_c	f64_c
f32_r	f32_c	f32_c
f64_r	f64_c	f64_c

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **n** – [in] [rocblas_int] the number of elements in x.
- **alpha** – [in] device pointer or host pointer for the scalar alpha.
- **alpha_type** – [in] [rocblas_datatype] specifies the datatype of alpha.
- **x** – [inout] device pointer storing vector x.
- **x_type** – [in] [rocblas_datatype] specifies the datatype of vector x.
- **incx** – [in] [rocblas_int] specifies the increment for the elements of x.
- **execution_type** – [in] [rocblas_datatype] specifies the datatype of computation.

rocblas_status **rocblas_scal_batched_ex**(*rocblas_handle* handle, *rocblas_int* n, const void *alpha, *rocblas_datatype* alpha_type, void *x, *rocblas_datatype* x_type, *rocblas_int* incx, *rocblas_int* batch_count, *rocblas_datatype* execution_type)

BLAS EX API

`scal_batched_ex` scales each element of each vector `x_i` with scalar `alpha`.

$$x_i := \alpha * x_i$$

Currently supported datatypes are as follows:

alpha_type	x_type	execution_type
f32_r	bf16_r	f32_r
bf16_r	bf16_r	f32_r
f16_r	f16_r	f16_r
f16_r	f16_r	f32_r
f32_r	f16_r	f32_r
f32_r	f32_r	f32_r
f64_r	f64_r	f64_r
f32_c	f32_c	f32_c
f64_c	f64_c	f64_c
f32_r	f32_c	f32_c
f64_r	f64_c	f64_c

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **n** – [in] [rocblas_int] the number of elements in `x`.
- **alpha** – [in] device pointer or host pointer for the scalar `alpha`.
- **alpha_type** – [in] [rocblas_datatype] specifies the datatype of `alpha`.
- **x** – [inout] device array of device pointers storing each vector `x_i`.
- **x_type** – [in] [rocblas_datatype] specifies the datatype of each vector `x_i`.
- **incx** – [in] [rocblas_int] specifies the increment for the elements of each `x_i`.
- **batch_count** – [in] [rocblas_int] number of instances in the batch.
- **execution_type** – [in] [rocblas_datatype] specifies the datatype of computation.

rocblas_status **rocblas_scal_strided_batched_ex**(*rocblas_handle* handle, *rocblas_int* n, const void *alpha, *rocblas_datatype* alpha_type, void *x, *rocblas_datatype* x_type, *rocblas_int* incx, *rocblas_stride* stridex, *rocblas_int* batch_count, *rocblas_datatype* execution_type)

BLAS EX API

`scal_strided_batched_ex` scales each element of vector `x` with scalar `alpha` over a set of strided batched vectors.

$$x := \alpha * x$$

Currently supported datatypes are as follows:

alpha_type	x_type	execution_type
f32_r	bf16_r	f32_r
bf16_r	bf16_r	f32_r
f16_r	f16_r	f16_r
f16_r	f16_r	f32_r
f32_r	f16_r	f32_r
f32_r	f32_r	f32_r
f64_r	f64_r	f64_r
f32_c	f32_c	f32_c
f64_c	f64_c	f64_c
f32_r	f32_c	f32_c
f64_r	f64_c	f64_c

Parameters

- **handle** – [in] [rocbblas_handle] handle to the rocbblas library context queue.
- **n** – [in] [rocbblas_int] the number of elements in x.
- **alpha** – [in] device pointer or host pointer for the scalar alpha.
- **alpha_type** – [in] [rocbblas_datatype] specifies the datatype of alpha.
- **x** – [inout] device pointer to the first vector x₁.
- **x_type** – [in] [rocbblas_datatype] specifies the datatype of each vector x_i.
- **incx** – [in] [rocbblas_int] specifies the increment for the elements of each x_i.
- **stridex** – [in] [rocbblas_stride] stride from the start of one vector (x_i) to the next one (x_{i+1}). There are no restrictions placed on stridex. However, ensure that stridex is of appropriate size. For a typical case this means stridex ≥ n * incx.
- **batch_count** – [in] [rocbblas_int] number of instances in the batch.
- **execution_type** – [in] [rocbblas_datatype] specifies the datatype of computation.

5.7.7 rocbblas_gemm_ex + batched, strided_batched

rocbblas_status **rocbblas_gemm_ex**(*rocbblas_handle* handle, *rocbblas_operation* transA, *rocbblas_operation* transB, *rocbblas_int* m, *rocbblas_int* n, *rocbblas_int* k, const void *alpha, const void *a, *rocbblas_datatype* a_type, *rocbblas_int* lda, const void *b, *rocbblas_datatype* b_type, *rocbblas_int* ldb, const void *beta, const void *c, *rocbblas_datatype* c_type, *rocbblas_int* ldc, void *d, *rocbblas_datatype* d_type, *rocbblas_int* ldd, *rocbblas_datatype* compute_type, *rocbblas_gemm_algo* algo, int32_t solution_index, uint32_t flags)

BLAS EX API

gemm_ex performs one of the matrix-matrix operations:

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

where op(X) is one of

$$\begin{aligned} \text{op}(X) &= X && \text{or} \\ \text{op}(X) &= X^{**T} && \text{or} \\ \text{op}(X) &= X^{**H}, \end{aligned}$$

alpha and beta are scalars, and A, B, C, and D are matrices, with $\text{op}(A)$ an m by k matrix, $\text{op}(B)$ a k by n matrix and C and D are m by n matrices. C and D may point to the same matrix if their parameters are identical.

Supported types are as follows:

- `rocblas_datatype_f64_r = a_type = b_type = c_type = d_type = compute_type`
- `rocblas_datatype_f32_r = a_type = b_type = c_type = d_type = compute_type`
- `rocblas_datatype_f16_r = a_type = b_type = c_type = d_type = compute_type`
- `rocblas_datatype_f16_r = a_type = b_type = c_type = d_type; rocblas_datatype_f32_r = compute_type`
- `rocblas_datatype_f16_r = a_type = b_type; rocblas_datatype_f32_r = c_type = d_type = compute_type`
- `rocblas_datatype_bf16_r = a_type = b_type = c_type = d_type; rocblas_datatype_f32_r = compute_type`
- `rocblas_datatype_bf16_r = a_type = b_type; rocblas_datatype_f32_r = c_type = d_type = compute_type`
- `rocblas_datatype_i8_r = a_type = b_type; rocblas_datatype_i32_r = c_type = d_type = compute_type`
- `rocblas_datatype_f32_c = a_type = b_type = c_type = d_type = compute_type`
- `rocblas_datatype_f64_c = a_type = b_type = c_type = d_type = compute_type`

Two int8 datatypes are supported: `int8_t` and `rocblas_int8x4`. `int8_t` is the C99 signed 8 bit integer. The default is `int8_t` and it is recommended `int8_t` be used. `rocblas_int8x4` is a packed datatype. The packed int 8 datatype occurs if the user sets:

```
flags |= rocblas_gemm_flags_pack_int8x4;
```

For this packed int8 datatype matrices A and B are packed into int8x4 in the k dimension. This will impose the following size restrictions on A or B:

- k must be a multiple of 4
- **if** `transA == rocblas_operation_transpose` then lda must be a multiple of 4
- **if** `transB == rocblas_operation_none` then ldb must be a multiple of 4
- **if** `transA == rocblas_operation_none` the matrix A must have each 4 consecutive values **in** the k dimension packed
- **if** `transB == rocblas_operation_transpose` the matrix B must have each 4 consecutive values **in** the k dimension packed.

This packing can be achieved with the following pseudo-code. The code assumes the original matrices are in A and B, and the packed matrices are A_packed and B_packed. The size of the A_packed and B_packed are the same as the size of the A and B respectively.

```
if(transA == rocblas_operation_none)
{
    int nb = 4;
    for(int i_m = 0; i_m < m; i_m++)
    {
        for(int i_k = 0; i_k < k; i_k++)
        {
            A_packed[i_k % nb + (i_m + (i_k / nb) * lda) * nb] = A[i_m + i_k * lda];
        }
    }
}
```

(continues on next page)

(continued from previous page)

```

else
{
    A_packed = A;
}
if(transB == rocblas_operation_transpose)
{
    int nb = 4;
    for(int i_n = 0; i_n < m; i_n++)
    {
        for(int i_k = 0; i_k < k; i_k++)
        {
            B_packed[i_k % nb + (i_n + (i_k / nb) * ldb) * nb] = B[i_n + i_k * ldb];
        }
    }
}
else
{
    B_packed = B;
}

```

Please note that the use of packed int8x4 is deprecated and will be removed in a future release. It is strongly recommended that users do not use the rocblas_gemm_flags_pack_int8x4 flag.

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **transA** – [in] [rocblas_operation] specifies the form of op(A).
- **transB** – [in] [rocblas_operation] specifies the form of op(B).
- **m** – [in] [rocblas_int] matrix dimension m.
- **n** – [in] [rocblas_int] matrix dimension n.
- **k** – [in] [rocblas_int] matrix dimension k.
- **alpha** – [in] [const void *] device pointer or host pointer specifying the scalar alpha. Same datatype as compute_type.
- **a** – [in] [void *] device pointer storing matrix A.
- **a_type** – [in] [rocblas_datatype] specifies the datatype of matrix A.
- **lda** – [in] [rocblas_int] specifies the leading dimension of A.
- **b** – [in] [void *] device pointer storing matrix B.
- **b_type** – [in] [rocblas_datatype] specifies the datatype of matrix B.
- **ldb** – [in] [rocblas_int] specifies the leading dimension of B.
- **beta** – [in] [const void *] device pointer or host pointer specifying the scalar beta. Same datatype as compute_type.
- **c** – [in] [void *] device pointer storing matrix C.
- **c_type** – [in] [rocblas_datatype] specifies the datatype of matrix C.
- **ldc** – [in] [rocblas_int] specifies the leading dimension of C.

- **d** – [out] [void *] device pointer storing matrix D. If d and c pointers are to the same matrix then d_type must equal c_type and ldd must equal ldc or the respective invalid status will be returned.
- **d_type** – [in] [rocblas_datatype] specifies the datatype of matrix D.
- **ldd** – [in] [rocblas_int] specifies the leading dimension of D.
- **compute_type** – [in] [rocblas_datatype] specifies the datatype of computation.
- **algo** – [in] [rocblas_gemm_algo] enumerant specifying the algorithm type.
- **solution_index** – [in] [int32_t] if algo is rocblas_gemm_algo_solution_index, this controls which solution is used. When algo is not rocblas_gemm_algo_solution_index, or if solution_index <= 0, the default solution is used. This parameter was unused in previous releases and instead always used the default solution
- **flags** – [in] [uint32_t] optional gemm flags.

rocblas_status rocblas_gemm_batched_ex(*rocblas_handle* handle, *rocblas_operation* transA, *rocblas_operation* transB, *rocblas_int* m, *rocblas_int* n, *rocblas_int* k, const void *alpha, const void *a, *rocblas_datatype* a_type, *rocblas_int* lda, const void *b, *rocblas_datatype* b_type, *rocblas_int* ldb, const void *beta, const void *c, *rocblas_datatype* c_type, *rocblas_int* ldc, void *d, *rocblas_datatype* d_type, *rocblas_int* ldd, *rocblas_int* batch_count, *rocblas_datatype* compute_type, *rocblas_gemm_algo* algo, int32_t solution_index, uint32_t flags)

BLAS EX API

gemm_batched_ex performs one of the batched matrix-matrix operations: $D_i = \alpha * op(A_i) * op(B_i) + \beta * C_i$, for $i = 1, \dots, \text{batch_count}$. 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, C, and D are batched pointers to matrices, with $op(A)$ an m by k by batch_count batched matrix, $op(B)$ a k by n by batch_count batched matrix and C and D are m by n by batch_count batched matrices. The batched matrices are an array of pointers to matrices. The number of pointers to matrices is batch_count. C and D may point to the same matrices if their parameters are identical.

Supported types are as follows:

- rocblas_datatype_f64_r = a_type = b_type = c_type = d_type = compute_type
- rocblas_datatype_f32_r = a_type = b_type = c_type = d_type = compute_type
- rocblas_datatype_f16_r = a_type = b_type = c_type = d_type = compute_type
- rocblas_datatype_f16_r = a_type = b_type = c_type = d_type; rocblas_datatype_f32_r = compute_type
- rocblas_datatype_bf16_r = a_type = b_type = c_type = d_type; rocblas_datatype_f32_r = compute_type
- rocblas_datatype_i8_r = a_type = b_type; rocblas_datatype_i32_r = c_type = d_type = compute_type
- rocblas_datatype_f32_c = a_type = b_type = c_type = d_type = compute_type
- rocblas_datatype_f64_c = a_type = b_type = c_type = d_type = compute_type

Two int8 datatypes are supported: int8_t and rocblas_int8x4. int8_t is the C99 signed 8 bit integer. The default is int8_t and it is recommended int8_t be used. rocblas_int8x4 is a packed datatype. The packed int 8 datatype occurs if the user sets:

```
flags |= rocblas_gemm_flags_pack_int8x4;
```

For this packed int8 datatype matrices A and B are packed into int8x4 in the k dimension. This will impose the following size restrictions on A or B:

- k must be a multiple of 4
- **if** transA == rocblas_operation_transpose then lda must be a multiple of 4
- **if** transB == rocblas_operation_none then ldb must be a multiple of 4
- **if** transA == rocblas_operation_none the matrix A must have each 4 consecutive values **in** the k dimension packed
- **if** transB == rocblas_operation_transpose the matrix B must have each 4 consecutive values **in** the k dimension packed.

This packing can be achieved with the following pseudo-code. The code assumes the original matrices are in A and B, and the packed matrices are A_packed and B_packed. The size of the A_packed and B_packed are the same as the size of the A and B respectively.

```

if(transA == rocblas_operation_none)
{
    int nb = 4;
    for(int i_m = 0; i_m < m; i_m++)
    {
        for(int i_k = 0; i_k < k; i_k++)
        {
            A_packed[i_k % nb + (i_m + (i_k / nb) * lda) * nb] = A[i_m + i_k * lda];
        }
    }
}
else
{
    A_packed = A;
}
if(transB == rocblas_operation_transpose)
{
    int nb = 4;
    for(int i_n = 0; i_n < m; i_n++)
    {
        for(int i_k = 0; i_k < k; i_k++)
        {
            B_packed[i_k % nb + (i_n + (i_k / nb) * ldb) * nb] = B[i_n + i_k * ldb];
        }
    }
}
else
{
    B_packed = B;
}

```

Please note that the use of packed int8x4 is deprecated and will be removed in a future release. It is strongly recommended that users do not use the rocblas_gemm_flags_pack_int8x4 flag.

Parameters

- **handle** – [**in**] [rocblas_handle] handle to the rocblas library context queue.
- **transA** – [**in**] [rocblas_operation] specifies the form of op(A).

- **transB** – [in] [rocblas_operation] specifies the form of $\text{op}(B)$.
- **m** – [in] [rocblas_int] matrix dimension m.
- **n** – [in] [rocblas_int] matrix dimension n.
- **k** – [in] [rocblas_int] matrix dimension k.
- **alpha** – [in] [const void *] device pointer or host pointer specifying the scalar alpha. Same datatype as compute_type.
- **a** – [in] [void *] device pointer storing array of pointers to each matrix A_i .
- **a_type** – [in] [rocblas_datatype] specifies the datatype of each matrix A_i .
- **lda** – [in] [rocblas_int] specifies the leading dimension of each A_i .
- **b** – [in] [void *] device pointer storing array of pointers to each matrix B_i .
- **b_type** – [in] [rocblas_datatype] specifies the datatype of each matrix B_i .
- **ldb** – [in] [rocblas_int] specifies the leading dimension of each B_i .
- **beta** – [in] [const void *] device pointer or host pointer specifying the scalar beta. Same datatype as compute_type.
- **c** – [in] [void *] device array of device pointers to each matrix C_i .
- **c_type** – [in] [rocblas_datatype] specifies the datatype of each matrix C_i .
- **ldc** – [in] [rocblas_int] specifies the leading dimension of each C_i .
- **d** – [out] [void *] device array of device pointers to each matrix D_i . If d and c are the same array of matrix pointers then d_type must equal c_type and ldd must equal ldc or the respective invalid status will be returned.
- **d_type** – [in] [rocblas_datatype] specifies the datatype of each matrix D_i .
- **ldd** – [in] [rocblas_int] specifies the leading dimension of each D_i .
- **batch_count** – [in] [rocblas_int] number of gemm operations in the batch.
- **compute_type** – [in] [rocblas_datatype] specifies the datatype of computation.
- **algo** – [in] [rocblas_gemm_algo] enumerant specifying the algorithm type.
- **solution_index** – [in] [int32_t] if algo is rocblas_gemm_algo_solution_index, this controls which solution is used. When algo is not rocblas_gemm_algo_solution_index, or if solution_index ≤ 0 , the default solution is used. This parameter was unused in previous releases and instead always used the default solution
- **flags** – [in] [uint32_t] optional gemm flags.

rocblas_status rocblas_gemm_strided_batched_ex(*rocblas_handle* handle, *rocblas_operation* transA, *rocblas_operation* transB, *rocblas_int* m, *rocblas_int* n, *rocblas_int* k, const void *alpha, const void *a, *rocblas_datatype* a_type, *rocblas_int* lda, *rocblas_stride* stride_a, const void *b, *rocblas_datatype* b_type, *rocblas_int* ldb, *rocblas_stride* stride_b, const void *beta, const void *c, *rocblas_datatype* c_type, *rocblas_int* ldc, *rocblas_stride* stride_c, void *d, *rocblas_datatype* d_type, *rocblas_int* ldd, *rocblas_stride* stride_d, *rocblas_int* batch_count, *rocblas_datatype* compute_type, *rocblas_gemm_algo* algo, int32_t solution_index, uint32_t flags)

BLAS EX API

gemm_strided_batched_ex performs one of the strided_batched matrix-matrix operations:

```
D_i = alpha*op(A_i)*op(B_i) + beta*C_i, for i = 1, ..., batch_count
```

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, C, and D are strided_batched matrices, with op(A) an m by k by batch_count strided_batched matrix, op(B) a k by n by batch_count strided_batched matrix and C and D are m by n by batch_count strided_batched matrices. C and D may point to the same matrices if their parameters are identical.

The strided_batched matrices are multiple matrices separated by a constant stride. The number of matrices is batch_count.

Supported types are as follows:

- rocbblas_datatype_f64_r = a_type = b_type = c_type = d_type = compute_type
- rocbblas_datatype_f32_r = a_type = b_type = c_type = d_type = compute_type
- rocbblas_datatype_f16_r = a_type = b_type = c_type = d_type = compute_type
- rocbblas_datatype_f16_r = a_type = b_type = c_type = d_type; rocbblas_datatype_f32_r = compute_type
- rocbblas_datatype_bf16_r = a_type = b_type = c_type = d_type; rocbblas_datatype_f32_r = compute_type
- rocbblas_datatype_i8_r = a_type = b_type; rocbblas_datatype_i32_r = c_type = d_type = compute_type
- rocbblas_datatype_f32_c = a_type = b_type = c_type = d_type = compute_type
- rocbblas_datatype_f64_c = a_type = b_type = c_type = d_type = compute_type

Two int8 datatypes are supported: int8_t and rocbblas_int8x4. int8_t is the C99 signed 8 bit integer. The default is int8_t and it is recommended int8_t be used. rocbblas_int8x4 is a packed datatype. The packed int 8 datatype occurs if the user sets:

```
flags |= rocbblas_gemm_flags_pack_int8x4;
```

For this packed int8 datatype matrices A and B are packed into int8x4 in the k dimension. This will impose the following size restrictions on A or B:

- k must be a multiple of 4
- if transA == rocbblas_operation_transpose then lda must be a multiple of 4
- if transB == rocbblas_operation_none then ldb must be a multiple of 4
- if transA == rocbblas_operation_none the matrix A must have each 4 consecutive values in the k dimension packed
- if transB == rocbblas_operation_transpose the matrix B must have each 4 consecutive values in the k dimension packed.

This packing can be achieved with the following pseudo-code. The code assumes the original matrices are in A and B, and the packed matrices are A_packed and B_packed. The size of the A_packed and B_packed are the same as the size of the A and B respectively.

```

if(transA == rocblas_operation_none)
{
    int nb = 4;
    for(int i_m = 0; i_m < m; i_m++)
    {
        for(int i_k = 0; i_k < k; i_k++)
        {
            A_packed[i_k % nb + (i_m + (i_k / nb) * lda) * nb] = A[i_m + i_k * lda];
        }
    }
}
else
{
    A_packed = A;
}
if(transB == rocblas_operation_transpose)
{
    int nb = 4;
    for(int i_n = 0; i_n < m; i_n++)
    {
        for(int i_k = 0; i_k < k; i_k++)
        {
            B_packed[i_k % nb + (i_n + (i_k / nb) * ldb) * nb] = B[i_n + i_k * ldb];
        }
    }
}
else
{
    B_packed = B;
}

```

Please note that the use of packed int8x4 is deprecated and will be removed in a future release. It is strongly recommended that users do not use the rocblas_gemm_flags_pack_int8x4 flag.

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **transA** – [in] [rocblas_operation] specifies the form of op(A).
- **transB** – [in] [rocblas_operation] specifies the form of op(B).
- **m** – [in] [rocblas_int] matrix dimension m.
- **n** – [in] [rocblas_int] matrix dimension n.
- **k** – [in] [rocblas_int] matrix dimension k.
- **alpha** – [in] [const void *] device pointer or host pointer specifying the scalar alpha. Same datatype as compute_type.
- **a** – [in] [void *] device pointer pointing to first matrix A_1.
- **a_type** – [in] [rocblas_datatype] specifies the datatype of each matrix A_i.
- **lda** – [in] [rocblas_int] specifies the leading dimension of each A_i.
- **stride_a** – [in] [rocblas_stride] specifies stride from start of one A_i matrix to the next A_(i + 1).

- **b** – [in] [void *] device pointer pointing to first matrix B_1.
- **b_type** – [in] [rocblas_datatype] specifies the datatype of each matrix B_i.
- **ldb** – [in] [rocblas_int] specifies the leading dimension of each B_i.
- **stride_b** – [in] [rocblas_stride] specifies stride from start of one B_i matrix to the next B_(i + 1).
- **beta** – [in] [const void *] device pointer or host pointer specifying the scalar beta. Same datatype as compute_type.
- **c** – [in] [void *] device pointer pointing to first matrix C_1.
- **c_type** – [in] [rocblas_datatype] specifies the datatype of each matrix C_i.
- **ldc** – [in] [rocblas_int] specifies the leading dimension of each C_i.
- **stride_c** – [in] [rocblas_stride] specifies stride from start of one C_i matrix to the next C_(i + 1).
- **d** – [out] [void *] device pointer storing each matrix D_i. If d and c pointers are to the same matrix then d_type must equal c_type and ldd must equal ldc and stride_d must equal stride_c or the respective invalid status will be returned.
- **d_type** – [in] [rocblas_datatype] specifies the datatype of each matrix D_i.
- **ldd** – [in] [rocblas_int] specifies the leading dimension of each D_i.
- **stride_d** – [in] [rocblas_stride] specifies stride from start of one D_i matrix to the next D_(i + 1).
- **batch_count** – [in] [rocblas_int] number of gemm operations in the batch.
- **compute_type** – [in] [rocblas_datatype] specifies the datatype of computation.
- **algo** – [in] [rocblas_gemm_algo] enumerant specifying the algorithm type.
- **solution_index** – [in] [int32_t] if algo is rocblas_gemm_algo_solution_index, this controls which solution is used. When algo is not rocblas_gemm_algo_solution_index, or if solution_index <= 0, the default solution is used. This parameter was unused in previous releases and instead always used the default solution
- **flags** – [in] [uint32_t] optional gemm flags.

5.7.8 rocblas_trsm_ex + batched, strided_batched

rocblas_status **rocblas_trsm_ex**(*rocblas_handle* handle, *rocblas_side* side, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_diagonal* diag, *rocblas_int* m, *rocblas_int* n, const void *alpha, const void *A, *rocblas_int* lda, void *B, *rocblas_int* ldb, const void *invA, *rocblas_int* invA_size, *rocblas_datatype* compute_type)

BLAS EX API

trsm_ex solves:

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

where alpha is a scalar, X and B are m by n matrices, A is triangular matrix and op(A) is one of

$$\text{op}(A) = A \quad \text{or} \quad \text{op}(A) = A^T \quad \text{or} \quad \text{op}(A) = A^H.$$

The matrix X is overwritten on B.

This function gives the user the ability to reuse the invA matrix between runs. If `invA == NULL`, `rocblas_trsm_ex` will automatically calculate invA on every run.

Setting up invA: The accepted invA matrix consists of the packed 128x128 inverses of the diagonal blocks of matrix A, followed by any smaller diagonal block that remains. To set up invA it is recommended that `rocblas_trtri_batched` be used with matrix A as the input.

Device memory of size 128 x k should be allocated for invA ahead of time, where k is m when `rocblas_side_left` and is n when `rocblas_side_right`. The actual number of elements in invA should be passed as `invA_size`.

To begin, `rocblas_trtri_batched` must be called on the full 128x128-sized diagonal blocks of matrix A. Below are the restricted parameters:

- `n = 128`
- `ldinvA = 128`
- `stride_invA = 128x128`
- `batch_count = k / 128`,

Then any remaining block may be added:

- `n = k % 128`
- `invA = invA + stride_invA * previous_batch_count`
- `ldinvA = 128`
- `batch_count = 1`

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **side** – [in] [rocblas_side]
 - `rocblas_side_left`: $\text{op}(A) * X = \alpha * B$
 - `rocblas_side_right`: $X * \text{op}(A) = \alpha * B$
- **uplo** – [in] [rocblas_fill]
 - `rocblas_fill_upper`: A is an upper triangular matrix.
 - `rocblas_fill_lower`: A is a lower triangular matrix.
- **transA** – [in] [rocblas_operation]
 - `transB`: $\text{op}(A) = A$.
 - `rocblas_operation_transpose`: $\text{op}(A) = A^T$
 - `rocblas_operation_conjugate_transpose`: $\text{op}(A) = A^H$
- **diag** – [in] [rocblas_diagonal]
 - `rocblas_diagonal_unit`: A is assumed to be unit triangular.
 - `rocblas_diagonal_non_unit`: A is not assumed to be unit triangular.
- **m** – [in] [rocblas_int] m specifies the number of rows of B. $m \geq 0$.
- **n** – [in] [rocblas_int] n specifies the number of columns of B. $n \geq 0$.
- **alpha** – [in] [void *] device pointer or host pointer specifying the scalar alpha. When alpha is `&zero` then A is not referenced, and B need not be set before entry.

- **A** – [in] [void *] device pointer storing matrix A. of dimension (lda, k), where k is m when rocblas_side_left and is n when rocblas_side_right only the upper/lower triangular part is accessed.
- **lda** – [in] [rocblas_int] lda specifies the first dimension of A.

```
if side == rocblas_side_left, lda >= max( 1, m ),
if side == rocblas_side_right, lda >= max( 1, n ).
```

- **B** – [inout] [void *] device pointer storing matrix B. B is of dimension (ldb, n). Before entry, the leading m by n part of the array B must contain the right-hand side matrix B, and on exit is overwritten by the solution matrix X.
- **ldb** – [in] [rocblas_int] ldb specifies the first dimension of B. ldb >= max(1, m).
- **invA** – [in] [void *] device pointer storing the inverse diagonal blocks of A. invA is of dimension (ld_invA, k), where k is m when rocblas_side_left and is n when rocblas_side_right. ld_invA must be equal to 128.
- **invA_size** – [in] [rocblas_int] invA_size specifies the number of elements of device memory in invA.
- **compute_type** – [in] [rocblas_datatype] specifies the datatype of computation.

rocblas_status rocblas_trsm_batched_ex(*rocblas_handle* handle, *rocblas_side* side, *rocblas_fill* uplo, *rocblas_operation* transA, *rocblas_diagonal* diag, *rocblas_int* m, *rocblas_int* n, const void *alpha, const void *A, *rocblas_int* lda, void *B, *rocblas_int* ldb, *rocblas_int* batch_count, const void *invA, *rocblas_int* invA_size, *rocblas_datatype* compute_type)

BLAS EX API

trsm_batched_ex solves:

```
op(A_i)*X_i = alpha*B_i or X_i*op(A_i) = alpha*B_i,
```

for $i = 1, \dots, \text{batch_count}$; and where alpha is a scalar, X and B are arrays of m by n matrices, A is an array of triangular matrix and each op(A_i) is one of

```
op( A_i ) = A_i or op( A_i ) = A_i^T or op( A_i ) = A_i^H.
```

Each matrix X_i is overwritten on B_i.

This function gives the user the ability to reuse the invA matrix between runs. If invA == NULL, rocblas_trsm_batched_ex will automatically calculate each invA_i on every run.

Setting up invA: Each accepted invA_i matrix consists of the packed 128x128 inverses of the diagonal blocks of matrix A_i, followed by any smaller diagonal block that remains. To set up each invA_i it is recommended that rocblas_trtri_batched be used with matrix A_i as the input. invA is an array of pointers of batch_count length holding each invA_i.

Device memory of size 128 x k should be allocated for each invA_i ahead of time, where k is m when rocblas_side_left and is n when rocblas_side_right. The actual number of elements in each invA_i should be passed as invA_size.

To begin, rocblas_trtri_batched must be called on the full 128x128-sized diagonal blocks of each matrix A_i. Below are the restricted parameters:

- n = 128
- ldinvA = 128

- `stride_invA = 128x128`
- `batch_count = k / 128,`

Then any remaining block may be added:

- `n = k % 128`
- `invA = invA + stride_invA * previous_batch_count`
- `ldinvA = 128`
- `batch_count = 1`

Parameters

- **handle** – [in] [rocbblas_handle] handle to the rocbblas library context queue.
- **side** – [in] [rocbblas_side]
 - `rocbblas_side_left`: $\text{op}(A) * X = \alpha * B$
 - `rocbblas_side_right`: $X * \text{op}(A) = \alpha * B$
- **uplo** – [in] [rocbblas_fill]
 - `rocbblas_fill_upper`: each A_i is an upper triangular matrix.
 - `rocbblas_fill_lower`: each A_i is a lower triangular matrix.
- **transA** – [in] [rocbblas_operation]
 - `transB`: $\text{op}(A) = A$.
 - `rocbblas_operation_transpose`: $\text{op}(A) = A^T$
 - `rocbblas_operation_conjugate_transpose`: $\text{op}(A) = A^H$
- **diag** – [in] [rocbblas_diagonal]
 - `rocbblas_diagonal_unit`: each A_i is assumed to be unit triangular.
 - `rocbblas_diagonal_non_unit`: each A_i is not assumed to be unit triangular.
- **m** – [in] [rocbblas_int] m specifies the number of rows of each B_i . $m \geq 0$.
- **n** – [in] [rocbblas_int] n specifies the number of columns of each B_i . $n \geq 0$.
- **alpha** – [in] [void *] device pointer or host pointer α specifying the scalar α . When α is `&zero` then A is not referenced, and B need not be set before entry.
- **A** – [in] [void *] device array of device pointers storing each matrix A_i . each A_i is of dimension (lda, k) , where k is m when `rocbblas_side_left` and is n when `rocbblas_side_right` only the upper/lower triangular part is accessed.
- **lda** – [in] [rocbblas_int] lda specifies the first dimension of each A_i .

```
if side == rocbblas_side_left, lda >= max( 1, m ),
if side == rocbblas_side_right, lda >= max( 1, n ).
```

- **B** – [inout] [void *] device array of device pointers storing each matrix B_i . each B_i is of dimension (ldb, n) . Before entry, the leading m by n part of the array B_i must contain the right-hand side matrix B_i , and on exit is overwritten by the solution matrix X_i
- **ldb** – [in] [rocbblas_int] ldb specifies the first dimension of each B_i . $ldb \geq \max(1, m)$.
- **batch_count** – [in] [rocbblas_int] specifies how many batches.

- **invA** – [in] [void *] device array of device pointers storing the inverse diagonal blocks of each A_i . each $invA_i$ is of dimension (ld_invA , k), where k is m when `rocbblas_side_left` and is n when `rocbblas_side_right`. ld_invA must be equal to 128.
- **invA_size** – [in] [rocbblas_int] $invA_size$ specifies the number of elements of device memory in each $invA_i$.
- **compute_type** – [in] [rocbblas_datatype] specifies the datatype of computation.

rocbblas_status **rocbblas_trsm_strided_batched_ex**(*rocbblas_handle* handle, *rocbblas_side* side, *rocbblas_fill* uplo, *rocbblas_operation* transA, *rocbblas_diagonal* diag, *rocbblas_int* m, *rocbblas_int* n, const void *alpha, const void *A, *rocbblas_int* lda, *rocbblas_stride* stride_A, void *B, *rocbblas_int* ldb, *rocbblas_stride* stride_B, *rocbblas_int* batch_count, const void *invA, *rocbblas_int* invA_size, *rocbblas_stride* stride_invA, *rocbblas_datatype* compute_type)

BLAS EX API

`trsm_strided_batched_ex` solves:

$$op(A_i) * X_i = \alpha * B_i \quad \text{or} \quad X_i * op(A_i) = \alpha * B_i,$$

for $i = 1, \dots, \text{batch_count}$; and where α is a scalar, X and B are strided batched m by n matrices, A is a strided batched triangular matrix and $op(A_i)$ is one of

$$op(A_i) = A_i \quad \text{or} \quad op(A_i) = A_i^T \quad \text{or} \quad op(A_i) = A_i^H.$$

Each matrix X_i is overwritten on B_i .

This function gives the user the ability to reuse each $invA_i$ matrix between runs. If $invA == \text{NULL}$, `rocbblas_trsm_batched_ex` will automatically calculate each $invA_i$ on every run.

Setting up $invA$: Each accepted $invA_i$ matrix consists of the packed 128×128 inverses of the diagonal blocks of matrix A_i , followed by any smaller diagonal block that remains. To set up $invA_i$ it is recommended that `rocbblas_trtri_batched` be used with matrix A_i as the input. $invA$ is a contiguous piece of memory holding each $invA_i$.

Device memory of size $128 \times k$ should be allocated for each $invA_i$ ahead of time, where k is m when `rocbblas_side_left` and is n when `rocbblas_side_right`. The actual number of elements in each $invA_i$ should be passed as $invA_size$.

To begin, `rocbblas_trtri_batched` must be called on the full 128×128 -sized diagonal blocks of each matrix A_i . Below are the restricted parameters:

- $n = 128$
- $ldinvA = 128$
- $stride_invA = 128 \times 128$
- $batch_count = k / 128$

Then any remaining block may be added:

- $n = k \% 128$
- $invA = invA + stride_invA * \text{previous_batch_count}$
- $ldinvA = 128$
- $batch_count = 1$

Parameters

- **handle** – [in] [rocbblas_handle] handle to the rocbblas library context queue.
- **side** – [in] [rocbblas_side]
 - rocbblas_side_left: $\text{op}(A)*X = \alpha*B$
 - rocbblas_side_right: $X*\text{op}(A) = \alpha*B$
- **uplo** – [in] [rocbblas_fill]
 - rocbblas_fill_upper: each A_i is an upper triangular matrix.
 - rocbblas_fill_lower: each A_i is a lower triangular matrix.
- **transA** – [in] [rocbblas_operation]
 - transB: $\text{op}(A) = A$.
 - rocbblas_operation_transpose: $\text{op}(A) = A^T$
 - rocbblas_operation_conjugate_transpose: $\text{op}(A) = A^H$
- **diag** – [in] [rocbblas_diagonal]
 - rocbblas_diagonal_unit: each A_i is assumed to be unit triangular.
 - rocbblas_diagonal_non_unit: each A_i is not assumed to be unit triangular.
- **m** – [in] [rocbblas_int] m specifies the number of rows of each B_i . $m \geq 0$.
- **n** – [in] [rocbblas_int] n specifies the number of columns of each B_i . $n \geq 0$.
- **alpha** – [in] [void *] device pointer or host pointer specifying the scalar α . When α is $\&0$ then A is not referenced, and B need not be set before entry.
- **A** – [in] [void *] device pointer storing matrix A . of dimension (lda, k), where k is m when rocbblas_side_left and is n when rocbblas_side_right only the upper/lower triangular part is accessed.
- **lda** – [in] [rocbblas_int] lda specifies the first dimension of A .

```
if side == rocbblas_side_left, lda >= max( 1, m ),
if side == rocbblas_side_right, lda >= max( 1, n ).
```

- **stride_A** – [in] [rocbblas_stride] The stride between each A matrix.
- **B** – [inout] [void *] device pointer pointing to first matrix B_i . each B_i is of dimension (ldb, n). Before entry, the leading m by n part of each array B_i must contain the right-hand side of matrix B_i , and on exit is overwritten by the solution matrix X_i .
- **ldb** – [in] [rocbblas_int] ldb specifies the first dimension of each B_i . $ldb \geq \max(1, m)$.
- **stride_B** – [in] [rocbblas_stride] The stride between each B_i matrix.
- **batch_count** – [in] [rocbblas_int] specifies how many batches.
- **invA** – [in] [void *] device pointer storing the inverse diagonal blocks of each A_i . invA points to the first invA_1 . each invA_i is of dimension (ld_{invA}, k), where k is m when rocbblas_side_left and is n when rocbblas_side_right. ld_{invA} must be equal to 128.
- **invA_size** – [in] [rocbblas_int] invA_size specifies the number of elements of device memory in each invA_i .
- **stride_invA** – [in] [rocbblas_stride] The stride between each invA matrix.
- **compute_type** – [in] [rocbblas_datatype] specifies the datatype of computation.

5.7.9 rocblas_Xgeam + batched, strided_batched

rocblas_status **rocblas_sgeam**(*rocblas_handle* handle, *rocblas_operation* transA, *rocblas_operation* transB, *rocblas_int* m, *rocblas_int* n, const float *alpha, const float *A, *rocblas_int* lda, const float *beta, const float *B, *rocblas_int* ldb, float *C, *rocblas_int* ldc)

rocblas_status **rocblas_dgeam**(*rocblas_handle* handle, *rocblas_operation* transA, *rocblas_operation* transB, *rocblas_int* m, *rocblas_int* n, const double *alpha, const double *A, *rocblas_int* lda, const double *beta, const double *B, *rocblas_int* ldb, double *C, *rocblas_int* ldc)

rocblas_status **rocblas_cgeam**(*rocblas_handle* handle, *rocblas_operation* transA, *rocblas_operation* transB, *rocblas_int* m, *rocblas_int* n, const *rocblas_float_complex* *alpha, const *rocblas_float_complex* *A, *rocblas_int* lda, const *rocblas_float_complex* *beta, const *rocblas_float_complex* *B, *rocblas_int* ldb, *rocblas_float_complex* *C, *rocblas_int* ldc)

rocblas_status **rocblas_zgeam**(*rocblas_handle* handle, *rocblas_operation* transA, *rocblas_operation* transB, *rocblas_int* m, *rocblas_int* n, const *rocblas_double_complex* *alpha, const *rocblas_double_complex* *A, *rocblas_int* lda, const *rocblas_double_complex* *beta, const *rocblas_double_complex* *B, *rocblas_int* ldb, *rocblas_double_complex* *C, *rocblas_int* ldc)

BLAS Level 3 API

geam performs one of the matrix-matrix operations:

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

where $\text{op}(X)$ is one of

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

alpha and beta are scalars, and A, B and C are matrices, with $\text{op}(A)$ an m by n matrix, $\text{op}(B)$ an m by n matrix, and C an m by n matrix.

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **transA** – [in] [rocblas_operation] specifies the form of $\text{op}(A)$.
- **transB** – [in] [rocblas_operation] specifies the form of $\text{op}(B)$.
- **m** – [in] [rocblas_int] matrix dimension m.
- **n** – [in] [rocblas_int] matrix dimension n.
- **alpha** – [in] device pointer or host pointer specifying the scalar alpha.
- **A** – [in] device pointer storing matrix A.
- **lda** – [in] [rocblas_int] specifies the leading dimension of A.
- **beta** – [in] device pointer or host pointer specifying the scalar beta.
- **B** – [in] device pointer storing matrix B.
- **ldb** – [in] [rocblas_int] specifies the leading dimension of B.

- **C** – [inout] device pointer storing matrix C.
- **ldc** – [in] [rocblas_int] specifies the leading dimension of C.

rocblas_status **rocblas_sgeam_batched**(*rocblas_handle* handle, *rocblas_operation* transA, *rocblas_operation* transB, *rocblas_int* m, *rocblas_int* n, const float *alpha, const float *const A[], *rocblas_int* lda, const float *beta, const float *const B[], *rocblas_int* ldb, float *const C[], *rocblas_int* ldc, *rocblas_int* batch_count)

rocblas_status **rocblas_dgeam_batched**(*rocblas_handle* handle, *rocblas_operation* transA, *rocblas_operation* transB, *rocblas_int* m, *rocblas_int* n, const double *alpha, const double *const A[], *rocblas_int* lda, const double *beta, const double *const B[], *rocblas_int* ldb, double *const C[], *rocblas_int* ldc, *rocblas_int* batch_count)

rocblas_status **rocblas_cgeam_batched**(*rocblas_handle* handle, *rocblas_operation* transA, *rocblas_operation* transB, *rocblas_int* m, *rocblas_int* n, const *rocblas_float_complex* *alpha, const *rocblas_float_complex* *const A[], *rocblas_int* lda, const *rocblas_float_complex* *beta, const *rocblas_float_complex* *const B[], *rocblas_int* ldb, *rocblas_float_complex* *const C[], *rocblas_int* ldc, *rocblas_int* batch_count)

rocblas_status **rocblas_zgeam_batched**(*rocblas_handle* handle, *rocblas_operation* transA, *rocblas_operation* transB, *rocblas_int* m, *rocblas_int* n, const *rocblas_double_complex* *alpha, const *rocblas_double_complex* *const A[], *rocblas_int* lda, const *rocblas_double_complex* *beta, const *rocblas_double_complex* *const B[], *rocblas_int* ldb, *rocblas_double_complex* *const C[], *rocblas_int* ldc, *rocblas_int* batch_count)

BLAS Level 3 API

geam_batched performs one of the batched matrix-matrix operations:

```
C_i = alpha*op( A_i ) + beta*op( B_i )  for i = 0, 1, ... batch_count - 1,
```

where alpha **and** beta are scalars, **and** op(A_i), op(B_i) **and** C_i are m by n matrices **and** op(X) **is** one of

```
op( X ) = X           or
op( X ) = X**T
```

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **transA** – [in] [rocblas_operation] specifies the form of op(A).
- **transB** – [in] [rocblas_operation] specifies the form of op(B).
- **m** – [in] [rocblas_int] matrix dimension m.
- **n** – [in] [rocblas_int] matrix dimension n.
- **alpha** – [in] device pointer or host pointer specifying the scalar alpha.
- **A** – [in] device array of device pointers storing each matrix A_i on the GPU. Each A_i is of dimension (lda, k), where k is m when transA == rocblas_operation_none and is n when transA == rocblas_operation_transpose.
- **lda** – [in] [rocblas_int] specifies the leading dimension of A.

- **beta** – [in] device pointer or host pointer specifying the scalar beta.
- **B** – [in] device array of device pointers storing each matrix B_i on the GPU. Each B_i is of dimension (ldb, k), where k is m when `transB == rocblas_operation_none` and is n when `transB == rocblas_operation_transpose`.
- **ldb** – [in] [rocblas_int] specifies the leading dimension of B.
- **C** – [inout] device array of device pointers storing each matrix C_i on the GPU. Each C_i is of dimension (ldc, n).
- **ldc** – [in] [rocblas_int] specifies the leading dimension of C.
- **batch_count** – [in] [rocblas_int] number of instances i in the batch.

```
rocblas_status rocblas_sgeam_strided_batched(rocblas_handle handle, rocblas_operation transA,
                                             rocblas_operation transB, rocblas_int m, rocblas_int n, const
                                             float *alpha, const float *A, rocblas_int lda, rocblas_stride
                                             stride_A, const float *beta, const float *B, rocblas_int ldb,
                                             rocblas_stride stride_B, float *C, rocblas_int ldc,
                                             rocblas_stride stride_C, rocblas_int batch_count)
```

```
rocblas_status rocblas_dgeam_strided_batched(rocblas_handle handle, rocblas_operation transA,
                                             rocblas_operation transB, rocblas_int m, rocblas_int n, const
                                             double *alpha, const double *A, rocblas_int lda,
                                             rocblas_stride stride_A, const double *beta, const double *B,
                                             rocblas_int ldb, rocblas_stride stride_B, double *C,
                                             rocblas_int ldc, rocblas_stride stride_C, rocblas_int
                                             batch_count)
```

```
rocblas_status rocblas_cgeam_strided_batched(rocblas_handle handle, rocblas_operation transA,
                                             rocblas_operation transB, rocblas_int m, rocblas_int n, const
                                             rocblas_float_complex *alpha, const rocblas_float_complex
                                             *A, rocblas_int lda, rocblas_stride stride_A, const
                                             rocblas_float_complex *beta, const rocblas_float_complex *B,
                                             rocblas_int ldb, rocblas_stride stride_B,
                                             rocblas_float_complex *C, rocblas_int ldc, rocblas_stride
                                             stride_C, rocblas_int batch_count)
```

```
rocblas_status rocblas_zgeam_strided_batched(rocblas_handle handle, rocblas_operation transA,
                                             rocblas_operation transB, rocblas_int m, rocblas_int n, const
                                             rocblas_double_complex *alpha, const
                                             rocblas_double_complex *A, rocblas_int lda, rocblas_stride
                                             stride_A, const rocblas_double_complex *beta, const
                                             rocblas_double_complex *B, rocblas_int ldb, rocblas_stride
                                             stride_B, rocblas_double_complex *C, rocblas_int ldc,
                                             rocblas_stride stride_C, rocblas_int batch_count)
```

BLAS Level 3 API

`geam_strided_batched` performs one of the batched matrix-matrix operations:

```
C_i = alpha*op( A_i ) + beta*op( B_i )  for i = 0, 1, ... batch_count - 1,
```

where alpha and beta are scalars, and op(A_i), op(B_i) and C_i are m by n matrices and op(X) is one of

(continues on next page)

(continued from previous page)

```

op( X ) = X      or
op( X ) = X**T

```

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **transA** – [in] [rocblas_operation] specifies the form of op(A).
- **transB** – [in] [rocblas_operation] specifies the form of op(B).
- **m** – [in] [rocblas_int] matrix dimension m.
- **n** – [in] [rocblas_int] matrix dimension n.
- **alpha** – [in] device pointer or host pointer specifying the scalar alpha.
- **A** – [in] device pointer to the first matrix A_0 on the GPU. Each A_i is of dimension (lda, k), where k is m when transA == rocblas_operation_none and is n when transA == rocblas_operation_transpose.
- **lda** – [in] [rocblas_int] specifies the leading dimension of A.
- **stride_A** – [in] [rocblas_stride] stride from the start of one matrix (A_i) and the next one (A_{i+1}).
- **beta** – [in] device pointer or host pointer specifying the scalar beta.
- **B** – [in] pointer to the first matrix B_0 on the GPU. Each B_i is of dimension (ldb, k), where k is m when transB == rocblas_operation_none and is n when transB == rocblas_operation_transpose.
- **ldb** – [in] [rocblas_int] specifies the leading dimension of B.
- **stride_B** – [in] [rocblas_stride] stride from the start of one matrix (B_i) and the next one (B_{i+1}).
- **C** – [inout] pointer to the first matrix C_0 on the GPU. Each C_i is of dimension (ldc, n).
- **ldc** – [in] [rocblas_int] specifies the leading dimension of C.
- **stride_C** – [in] [rocblas_stride] stride from the start of one matrix (C_i) and the next one (C_{i+1}).
- **batch_count** – [in] [rocblas_int] number of instances i in the batch.

5.7.10 rocblas_Xdggmm + batched, strided_batched

```

rocblas_status rocblas_sdggmm(rocblas_handle handle, rocblas_side side, rocblas_int m, rocblas_int n, const float
    *A, rocblas_int lda, const float *x, rocblas_int incx, float *C, rocblas_int ldc)

```

```

rocblas_status rocblas_ddggmm(rocblas_handle handle, rocblas_side side, rocblas_int m, rocblas_int n, const
    double *A, rocblas_int lda, const double *x, rocblas_int incx, double *C,
    rocblas_int ldc)

```

```

rocblas_status rocblas_cdggmm(rocblas_handle handle, rocblas_side side, rocblas_int m, rocblas_int n, const
    rocblas_float_complex *A, rocblas_int lda, const rocblas_float_complex *x,
    rocblas_int incx, rocblas_float_complex *C, rocblas_int ldc)

```

```
roblas_status roblas_zdggmm(roblas_handle handle, roblas_side side, roblas_int m, roblas_int n, const
    roblas_double_complex *A, roblas_int lda, const roblas_double_complex *x,
    roblas_int incx, roblas_double_complex *C, roblas_int ldc)
```

BLAS Level 3 API

dgmm performs one of the matrix-matrix operations:

```
C = A * diag(x) if side == roblas_side_right
C = diag(x) * A if side == roblas_side_left
```

where C **and** A are m by n dimensional matrices. diag(x) **is** a diagonal matrix **and** x **is** vector of dimension n **if** side == *roblas_side_right* **and** dimension m **if** side == *roblas_side_left*.

Parameters

- **handle** – [in] [*roblas_handle*] handle to the roblas library context queue.
- **side** – [in] [*roblas_side*] specifies the side of diag(x).
- **m** – [in] [*roblas_int*] matrix dimension m.
- **n** – [in] [*roblas_int*] matrix dimension n.
- **A** – [in] device pointer storing matrix A.
- **lda** – [in] [*roblas_int*] specifies the leading dimension of A.
- **x** – [in] device pointer storing vector x.
- **incx** – [in] [*roblas_int*] specifies the increment between values of x
- **C** – [inout] device pointer storing matrix C.
- **ldc** – [in] [*roblas_int*] specifies the leading dimension of C.

```
roblas_status roblas_sdggmm_batched(roblas_handle handle, roblas_side side, roblas_int m, roblas_int n,
    const float *const A[], roblas_int lda, const float *const x[], roblas_int
    incx, float *const C[], roblas_int ldc, roblas_int batch_count)
```

```
roblas_status roblas_ddggmm_batched(roblas_handle handle, roblas_side side, roblas_int m, roblas_int n,
    const double *const A[], roblas_int lda, const double *const x[],
    roblas_int incx, double *const C[], roblas_int ldc, roblas_int
    batch_count)
```

```
roblas_status roblas_cdggmm_batched(roblas_handle handle, roblas_side side, roblas_int m, roblas_int n,
    const roblas_float_complex *const A[], roblas_int lda, const
    roblas_float_complex *const x[], roblas_int incx,
    roblas_float_complex *const C[], roblas_int ldc, roblas_int
    batch_count)
```

```
roblas_status roblas_zdggmm_batched(roblas_handle handle, roblas_side side, roblas_int m, roblas_int n,
    const roblas_double_complex *const A[], roblas_int lda, const
    roblas_double_complex *const x[], roblas_int incx,
    roblas_double_complex *const C[], roblas_int ldc, roblas_int
    batch_count)
```

BLAS Level 3 API

dgmm_batched performs one of the batched matrix-matrix operations:

```

C_i = A_i * diag(x_i) for i = 0, 1, ... batch_count-1 if side == rocblas_side_right
C_i = diag(x_i) * A_i for i = 0, 1, ... batch_count-1 if side == rocblas_side_left,

where C_i and A_i are m by n dimensional matrices. diag(x_i) is a diagonal matrix
and x_i is vector of dimension n if side == rocblas_side_right and dimension m
if side == rocblas_side_left.

```

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **side** – [in] [rocblas_side] specifies the side of diag(x).
- **m** – [in] [rocblas_int] matrix dimension m.
- **n** – [in] [rocblas_int] matrix dimension n.
- **A** – [in] device array of device pointers storing each matrix A_i on the GPU. Each A_i is of dimension (lda, n).
- **lda** – [in] [rocblas_int] specifies the leading dimension of A_i.
- **x** – [in] device array of device pointers storing each vector x_i on the GPU. Each x_i is of dimension n if side == rocblas_side_right and dimension m if side == rocblas_side_left.
- **incx** – [in] [rocblas_int] specifies the increment between values of x_i.
- **C** – [inout] device array of device pointers storing each matrix C_i on the GPU. Each C_i is of dimension (ldc, n).
- **ldc** – [in] [rocblas_int] specifies the leading dimension of C_i.
- **batch_count** – [in] [rocblas_int] number of instances in the batch.

```

rocblas_status rocblas_sdggmm_strided_batched(rocblas_handle handle, rocblas_side side, rocblas_int m,
rocblas_int n, const float *A, rocblas_int lda, rocblas_stride
stride_A, const float *x, rocblas_int incx, rocblas_stride
stride_x, float *C, rocblas_int ldc, rocblas_stride stride_C,
rocblas_int batch_count)

```

```

rocblas_status rocblas_ddggmm_strided_batched(rocblas_handle handle, rocblas_side side, rocblas_int m,
rocblas_int n, const double *A, rocblas_int lda, rocblas_stride
stride_A, const double *x, rocblas_int incx, rocblas_stride
stride_x, double *C, rocblas_int ldc, rocblas_stride stride_C,
rocblas_int batch_count)

```

```

rocblas_status rocblas_cdggmm_strided_batched(rocblas_handle handle, rocblas_side side, rocblas_int m,
rocblas_int n, const rocblas_float_complex *A, rocblas_int
lda, rocblas_stride stride_A, const rocblas_float_complex *x,
rocblas_int incx, rocblas_stride stride_x,
rocblas_float_complex *C, rocblas_int ldc, rocblas_stride
stride_C, rocblas_int batch_count)

```

```

rocblas_status rocblas_zdggmm_strided_batched(rocblas_handle handle, rocblas_side side, rocblas_int m,
rocblas_int n, const rocblas_double_complex *A, rocblas_int
lda, rocblas_stride stride_A, const rocblas_double_complex
*x, rocblas_int incx, rocblas_stride stride_x,
rocblas_double_complex *C, rocblas_int ldc, rocblas_stride
stride_C, rocblas_int batch_count)

```

BLAS Level 3 API

dgmm_strided_batched performs one of the batched matrix-matrix operations:

```
C_i = A_i * diag(x_i)  if side == rocblas_side_right  for i = 0, 1, ... batch_
↪count-1
C_i = diag(x_i) * A_i  if side == rocblas_side_left   for i = 0, 1, ... batch_
↪count-1,
```

where C_i and A_i are m by n dimensional matrices. $\text{diag}(x_i)$ is a diagonal matrix and x_i is vector of dimension n if `side == rocblas_side_right` and dimension m if `side == rocblas_side_left`.

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **side** – [in] [rocblas_side] specifies the side of $\text{diag}(x)$.
- **m** – [in] [rocblas_int] matrix dimension m .
- **n** – [in] [rocblas_int] matrix dimension n .
- **A** – [in] device pointer to the first matrix A_0 on the GPU. Each A_i is of dimension (lda, n).
- **lda** – [in] [rocblas_int] specifies the leading dimension of A .
- **stride_A** – [in] [rocblas_stride] stride from the start of one matrix (A_i) and the next one (A_{i+1}).
- **x** – [in] pointer to the first vector x_0 on the GPU. Each x_i is of dimension n if `side == rocblas_side_right` and dimension m if `side == rocblas_side_left`.
- **incx** – [in] [rocblas_int] specifies the increment between values of x .
- **stride_x** – [in] [rocblas_stride] stride from the start of one vector (x_i) and the next one (x_{i+1}).
- **C** – [inout] device pointer to the first matrix C_0 on the GPU. Each C_i is of dimension (ldc, n).
- **ldc** – [in] [rocblas_int] specifies the leading dimension of C .
- **stride_C** – [in] [rocblas_stride] stride from the start of one matrix (C_i) and the next one (C_{i+1}).
- **batch_count** – [in] [rocblas_int] number of instances i in the batch.

5.8 rocBLAS Beta Features

To allow for future growth and changes, the features in this section are not subject to the same level of backwards compatibility and support as the normal rocBLAS API. These features are subject to change and/or removal in future release of rocBLAS.

To use the following beta API features, `ROCBLAS_BETA_FEATURES_API` must be defined before including `rocblas.h`.

5.8.1 rocblas_gemm_ex_get_solutions + batched, strided_batched

```
rocblas_status rocblas_gemm_ex_get_solutions(rocblas_handle handle, rocblas_operation transA,
                                             rocblas_operation transB, rocblas_int m, rocblas_int n,
                                             rocblas_int k, const void *alpha, const void *a,
                                             rocblas_datatype a_type, rocblas_int lda, const void *b,
                                             rocblas_datatype b_type, rocblas_int ldb, const void *beta,
                                             const void *c, rocblas_datatype c_type, rocblas_int ldc, void
                                             *d, rocblas_datatype d_type, rocblas_int ldd,
                                             rocblas_datatype compute_type, rocblas_gemm_algo algo,
                                             uint32_t flags, rocblas_int *list_array, rocblas_int *list_size)
```

BLAS BETA API

gemm_ex_get_solutions gets the indices for all the solutions that can solve a corresponding call to gemm_ex. Which solution is used by gemm_ex is controlled by the solution_index parameter.

All parameters correspond to gemm_ex except for list_array and list_size, which are used as input and output for getting the solution indices. If list_array is NULL, list_size is an output and will be filled with the number of solutions that can solve the GEMM. If list_array is not NULL, then it must be pointing to an array with at least list_size elements and will be filled with the solution indices that can solve the GEMM: the number of elements filled is min(list_size, # of solutions).

Parameters

- **handle** – [in] [rocblas_handle] handle to the rocblas library context queue.
- **transA** – [in] [rocblas_operation] specifies the form of op(A).
- **transB** – [in] [rocblas_operation] specifies the form of op(B).
- **m** – [in] [rocblas_int] matrix dimension m.
- **n** – [in] [rocblas_int] matrix dimension n.
- **k** – [in] [rocblas_int] matrix dimension k.
- **alpha** – [in] [const void *] device pointer or host pointer specifying the scalar alpha. Same datatype as compute_type.
- **a** – [in] [void *] device pointer storing matrix A.
- **a_type** – [in] [rocblas_datatype] specifies the datatype of matrix A.
- **lda** – [in] [rocblas_int] specifies the leading dimension of A.
- **b** – [in] [void *] device pointer storing matrix B.
- **b_type** – [in] [rocblas_datatype] specifies the datatype of matrix B.
- **ldb** – [in] [rocblas_int] specifies the leading dimension of B.
- **beta** – [in] [const void *] device pointer or host pointer specifying the scalar beta. Same datatype as compute_type.
- **c** – [in] [void *] device pointer storing matrix C.
- **c_type** – [in] [rocblas_datatype] specifies the datatype of matrix C.
- **ldc** – [in] [rocblas_int] specifies the leading dimension of C.
- **d** – [out] [void *] device pointer storing matrix D. If d and c pointers are to the same matrix then d_type must equal c_type and ldd must equal ldc or the respective invalid status will be returned.

- **d_type** – [in] [rocbblas_datatype] specifies the datatype of matrix D.
- **ldd** – [in] [rocbblas_int] specifies the leading dimension of D.
- **compute_type** – [in] [rocbblas_datatype] specifies the datatype of computation.
- **algo** – [in] [rocbblas_gemm_algo] enumerator specifying the algorithm type.
- **flags** – [in] [uint32_t] optional gemm flags.
- **list_array** – [out] [rocbblas_int *] output array for solution indices or NULL if getting number of solutions
- **list_size** – [inout] [rocbblas_int *] size of list_array if getting solution indices or output with number of solutions if list_array is NULL

```
rocbblas_status rocbblas_gemm_batched_ex_get_solutions(rocbblas_handle handle, rocbblas_operation transA,
rocbblas_operation transB, rocbblas_int m,
rocbblas_int n, rocbblas_int k, const void *alpha,
const void *a, rocbblas_datatype a_type, rocbblas_int
lda, const void *b, rocbblas_datatype b_type,
rocbblas_int ldb, const void *beta, const void *c,
rocbblas_datatype c_type, rocbblas_int ldc, void *d,
rocbblas_datatype d_type, rocbblas_int ldd,
rocbblas_int batch_count, rocbblas_datatype
compute_type, rocbblas_gemm_algo algo, uint32_t
flags, rocbblas_int *list_array, rocbblas_int *list_size)
```

BLAS BETA API

rocbblas_gemm_batched_ex_get_solutions gets the indices for all the solutions that can solve a corresponding call to gemm_batched_ex. Which solution is used by gemm_batched_ex is controlled by the solution_index parameter.

All parameters correspond to gemm_batched_ex except for list_array and list_size, which are used as input and output for getting the solution indices. If list_array is NULL, list_size is an output and will be filled with the number of solutions that can solve the GEMM. If list_array is not NULL, then it must be pointing to an array with at least list_size elements and will be filled with the solution indices that can solve the GEMM: the number of elements filled is min(list_size, # of solutions).

Parameters

- **handle** – [in] [rocbblas_handle] handle to the rocbblas library context queue.
- **transA** – [in] [rocbblas_operation] specifies the form of op(A).
- **transB** – [in] [rocbblas_operation] specifies the form of op(B).
- **m** – [in] [rocbblas_int] matrix dimension m.
- **n** – [in] [rocbblas_int] matrix dimension n.
- **k** – [in] [rocbblas_int] matrix dimension k.
- **alpha** – [in] [const void *] device pointer or host pointer specifying the scalar alpha. Same datatype as compute_type.
- **a** – [in] [void *] device pointer storing array of pointers to each matrix A_i.
- **a_type** – [in] [rocbblas_datatype] specifies the datatype of each matrix A_i.
- **lda** – [in] [rocbblas_int] specifies the leading dimension of each A_i.
- **b** – [in] [void *] device pointer storing array of pointers to each matrix B_i.
- **b_type** – [in] [rocbblas_datatype] specifies the datatype of each matrix B_i.

- **ldb** – [in] [rocblas_int] specifies the leading dimension of each B_i.
- **beta** – [in] [const void *] device pointer or host pointer specifying the scalar beta. Same datatype as compute_type.
- **c** – [in] [void *] device array of device pointers to each matrix C_i.
- **c_type** – [in] [rocblas_datatype] specifies the datatype of each matrix C_i.
- **ldc** – [in] [rocblas_int] specifies the leading dimension of each C_i.
- **d** – [out] [void *] device array of device pointers to each matrix D_i. If d and c are the same array of matrix pointers then d_type must equal c_type and ldd must equal ldc or the respective invalid status will be returned.
- **d_type** – [in] [rocblas_datatype] specifies the datatype of each matrix D_i.
- **ldd** – [in] [rocblas_int] specifies the leading dimension of each D_i.
- **batch_count** – [in] [rocblas_int] number of gemm operations in the batch.
- **compute_type** – [in] [rocblas_datatype] specifies the datatype of computation.
- **algo** – [in] [rocblas_gemm_algo] enumerant specifying the algorithm type.
- **flags** – [in] [uint32_t] optional gemm flags.
- **list_array** – [out] [rocblas_int *] output array for solution indices or NULL if getting number of solutions
- **list_size** – [inout] [rocblas_int *] size of list_array if getting solution indices or output with number of solutions if list_array is NULL

rocblas_status rocblas_gemm_strided_batched_ex_get_solutions(*rocblas_handle* handle, *rocblas_operation* transA, *rocblas_operation* transB, *rocblas_int* m, *rocblas_int* n, *rocblas_int* k, const void *alpha, const void *a, *rocblas_datatype* a_type, *rocblas_int* lda, *rocblas_stride* stride_a, const void *b, *rocblas_datatype* b_type, *rocblas_int* ldb, *rocblas_stride* stride_b, const void *beta, const void *c, *rocblas_datatype* c_type, *rocblas_int* ldc, *rocblas_stride* stride_c, void *d, *rocblas_datatype* d_type, *rocblas_int* ldd, *rocblas_stride* stride_d, *rocblas_int* batch_count, *rocblas_datatype* compute_type, *rocblas_gemm_algo* algo, uint32_t flags, *rocblas_int* *list_array, *rocblas_int* *list_size)

BLAS BETA API

gemm_strided_batched_ex_get_solutions gets the indices for all the solutions that can solve a corresponding call to gemm_strided_batched_ex. Which solution is used by gemm_strided_batched_ex is controlled by the solution_index parameter.

All parameters correspond to gemm_strided_batched_ex except for list_array and list_size, which are used as input and output for getting the solution indices. If list_array is NULL, list_size is an output and will be filled with the number of solutions that can solve the GEMM. If list_array is not NULL, then it must be pointing to an array with at least list_size elements and will be filled with the solution indices that can solve the GEMM: the number of elements filled is min(list_size, # of solutions).

Parameters

- **handle** – [in] [rocbblas_handle] handle to the rocbblas library context queue.
- **transA** – [in] [rocbblas_operation] specifies the form of $op(A)$.
- **transB** – [in] [rocbblas_operation] specifies the form of $op(B)$.
- **m** – [in] [rocbblas_int] matrix dimension m.
- **n** – [in] [rocbblas_int] matrix dimension n.
- **k** – [in] [rocbblas_int] matrix dimension k.
- **alpha** – [in] [const void *] device pointer or host pointer specifying the scalar alpha. Same datatype as compute_type.
- **a** – [in] [void *] device pointer pointing to first matrix A_1 .
- **a_type** – [in] [rocbblas_datatype] specifies the datatype of each matrix A_i .
- **lda** – [in] [rocbblas_int] specifies the leading dimension of each A_i .
- **stride_a** – [in] [rocbblas_stride] specifies stride from start of one A_i matrix to the next $A_{(i+1)}$.
- **b** – [in] [void *] device pointer pointing to first matrix B_1 .
- **b_type** – [in] [rocbblas_datatype] specifies the datatype of each matrix B_i .
- **ldb** – [in] [rocbblas_int] specifies the leading dimension of each B_i .
- **stride_b** – [in] [rocbblas_stride] specifies stride from start of one B_i matrix to the next $B_{(i+1)}$.
- **beta** – [in] [const void *] device pointer or host pointer specifying the scalar beta. Same datatype as compute_type.
- **c** – [in] [void *] device pointer pointing to first matrix C_1 .
- **c_type** – [in] [rocbblas_datatype] specifies the datatype of each matrix C_i .
- **ldc** – [in] [rocbblas_int] specifies the leading dimension of each C_i .
- **stride_c** – [in] [rocbblas_stride] specifies stride from start of one C_i matrix to the next $C_{(i+1)}$.
- **d** – [out] [void *] device pointer storing each matrix D_i . If d and c pointers are to the same matrix then d_type must equal c_type and ldd must equal ldc and stride_d must equal stride_c or the respective invalid status will be returned.
- **d_type** – [in] [rocbblas_datatype] specifies the datatype of each matrix D_i .
- **ldd** – [in] [rocbblas_int] specifies the leading dimension of each D_i .
- **stride_d** – [in] [rocbblas_stride] specifies stride from start of one D_i matrix to the next $D_{(i+1)}$.
- **batch_count** – [in] [rocbblas_int] number of gemm operations in the batch.
- **compute_type** – [in] [rocbblas_datatype] specifies the datatype of computation.
- **algo** – [in] [rocbblas_gemm_algo] enumerant specifying the algorithm type.
- **flags** – [in] [uint32_t] optional gemm flags.
- **list_array** – [out] [rocbblas_int *] output array for solution indices or NULL if getting number of solutions

- **list_size** – [inout] [rocblas_int *] size of list_array if getting solution indices or output with number of solutions if list_array is NULL

5.9 Graph Support for rocBLAS

Graph support is added as a beta feature in rocBLAS. Most of the rocBLAS functions can be captured into a graph node via Graph ManagementHIP APIs, except those listed in *Functions Unsupported with Graph Capture*. For complete list of support graph APIs, refer to [Graph ManagementHIP API](#).

```
CHECK_HIP_ERROR((hipStreamBeginCapture(stream, hipStreamCaptureModeGlobal));
rocblas_<function>(<arguments>);
CHECK_HIP_ERROR(hipStreamEndCapture(stream, &graph));
```

The above code will create a graph with *rocblas_function()* as graph node. The captured graph can be launched as shown below:

```
CHECK_HIP_ERROR(hipGraphInstantiate(&instance, graph, NULL, NULL, 0));
CHECK_HIP_ERROR(hipGraphLaunch(instance, stream));
```

Graph support requires Asynchronous HIP APIs, hence, users must enable stream-order memory allocation. For more details refer to section *Stream-Ordered Memory Allocation*.

During stream capture, rocBLAS stores the allocated host and device memory in the handle and the allocated memory will be freed when the handle is destroyed.

5.9.1 Functions Unsupported with Graph Capture

- The following Level-1 functions place results into host buffers (in pointer mode host) which enforces synchronization.
 - *dot*
 - *asum*
 - *nrm2*
 - *imax*
 - *imin*
- BLAS Level-3 and BLAS-EX functions in pointer mode device do not support HIP Graph. Support will be added in future releases.

5.9.2 HIP Graph Known Issues in rocBLAS

- On Windows platform, batched functions (Level-1, Level-2 and Level-3) produce incorrect results.

5.10 Device Memory Allocation in rocBLAS

The following computational functions use temporary device memory.

Function	use of temporary device memory
L1 reduction functions <ul style="list-style-type: none"> • rocblas_Xdot • rocblas_Xmax • rocblas_Xmin • rocblas_Xnrm2 • rocblas_dot_ex • rocblas_nrm2_ex 	reduction array
L2 functions <ul style="list-style-type: none"> • rocblas_Xtbmv • rocblas_Xtpmv • rocblas_Xtrmv • rocblas_Xtrsv • rocblas_Xgemv (optional) 	result array before overwriting input column reductions of skinny transposed matrices
L3 gemm based functions <ul style="list-style-type: none"> • rocblas_Xtrsm • rocblas_Xgemm • rocblas_Xtrtri 	block of matrix
auxiliary <ul style="list-style-type: none"> • rocblas_set_vector • rocblas_get_vector • rocblas_set_matrix • rocblas_get_matrix 	buffer to compress noncontiguous arrays

For temporary device memory, rocBLAS uses a per-handle memory allocation with out-of-band management. The temporary device memory is stored in the handle. This allows for recycling temporary device memory across multiple computational kernels that use the same handle. Each handle has a single stream, and kernels execute in order in the stream, with each kernel completing before the next kernel in the stream starts. There are 4 schemes for temporary device memory:

1. **rocBLAS_managed**: This is the default scheme. If there is not enough memory in the handle, computational functions allocate the memory they require. Note that any memory allocated persists in the handle, so it is available for later computational functions that use the handle.
2. **user_managed, preallocate**: An environment variable is set before the rocBLAS handle is created, and thereafter there are no more allocations or deallocations.
3. **user_managed, manual**: The user calls helper functions to get or set memory size throughout the program, thereby controlling when allocation and deallocation occur.
4. **user_owned**: The user allocates workspace and calls a helper function to allow rocBLAS to access the workspace.

The default scheme has the disadvantage that allocation is synchronizing, so if there is not enough memory in the

handle, a synchronizing deallocation and allocation occur.

5.10.1 Environment Variable for Preallocating

The environment variable `ROCBLAS_DEVICE_MEMORY_SIZE` is used to set how much memory to preallocate:

- If `> 0`, sets the default handle device memory size to the specified size (in bytes)
- If `== 0` or unset, lets rocBLAS manage device memory, using a default size (like 32MB), and expanding it when necessary

5.10.2 Functions for Manually Setting Memory Size

- `rocbblas_set_device_memory_size`
- `rocbblas_get_device_memory_size`
- `rocbblas_is_user_managing_device_memory`

5.10.3 Function for Setting User Owned Workspace

- `rocbblas_set_workspace`

5.10.4 Functions for Finding How Much Memory Is Required

- `rocbblas_start_device_memory_size_query`
- `rocbblas_stop_device_memory_size_query`
- `rocbblas_is_managing_device_memory`

See the API section for information on the above functions.

5.10.5 rocBLAS Function Return Values for Insufficient Device Memory

If the user preallocates or manually allocates, then that size is used as the limit, and no resizing or synchronizing ever occurs. The following two function return values indicate insufficient memory:

- `rocbblas_status == rocbblas_status_memory_error`: indicates there is not sufficient device memory for a rocBLAS function
- `rocbblas_status == rocbblas_status_perf_degraded`: indicates that a slower algorithm was used because of insufficient device memory for the optimal algorithm

5.10.6 Stream-Ordered Memory Allocation

Stream-ordered device memory allocation is added to rocBLAS. Asynchronous allocators (`hipMallocAsync()` and `hipFreeAsync()`) are used to allow allocation and free to be stream order.

This is a non-default beta option enabled by setting the environment variable `ROCBLAS_STREAM_ORDER_ALLOC`.

A user may check if the device supports stream-order allocation by calling `hipDeviceGetAttribute()` with device attribute `hipDeviceAttributeMemoryPoolsSupported`.

Environment Variable to Enable Stream-Ordered Memory Allocation

On supported platforms, environment variable `ROCBLAS_STREAM_ORDER_ALLOC` is used to enable stream-ordered memory allocation.

- if `> 0`, sets the allocation to be stream-ordered, uses `hipMallocAsync/hipFreeAsync` to manage device memory.
- if `== 0` or unset, uses `hipMalloc/hipFree` to manage device memory.

Supports Switching Streams Without Any Synchronization

Stream-order memory allocation allows switching of streams without the need to call `hipStreamSynchronize()`.

5.11 Logging in rocBLAS

Note that performance will degrade when logging is enabled.

User can set four environment variables to control logging:

- `ROCBLAS_LAYER`
- `ROCBLAS_LOG_TRACE_PATH`
- `ROCBLAS_LOG_BENCH_PATH`
- `ROCBLAS_LOG_PROFILE_PATH`

`ROCBLAS_LAYER` is a bitwise OR of zero or more bit masks as follows:

- If `ROCBLAS_LAYER` is not set, then there is no logging.
- If `(ROCBLAS_LAYER & 1) != 0`, then there is trace logging.
- If `(ROCBLAS_LAYER & 2) != 0`, then there is bench logging.
- If `(ROCBLAS_LAYER & 4) != 0`, then there is profile logging.

Trace logging outputs a line each time a rocBLAS function is called. The line contains the function name and the values of arguments.

Bench logging outputs a line each time a rocBLAS function is called. The line can be used with the executable `rocblas-bench` to call the function with the same arguments.

Profile logging, at the end of program execution, outputs a YAML description of each rocBLAS function called, the values of its performance-critical arguments, and the number of times it was called with those arguments (the `call_count`). Some arguments, such as `alpha` and `beta` in GEMM, are recorded with a value representing the category that the argument falls in, such as `-1`, `0`, `1`, or `2`. The number of categories, and the values representing them,

may change over time, depending on how many categories are needed to adequately represent all the values that can affect the performance of the function.

The default stream for logging output is standard error. Three environment variables can set the full path name for a log file:

- `ROCBLAS_LOG_TRACE_PATH` sets the full path name for trace logging.
- `ROCBLAS_LOG_BENCH_PATH` sets the full path name for bench logging.
- `ROCBLAS_LOG_PROFILE_PATH` sets the full path name for profile logging.

For example, in Bash shell, to output bench logging to the file `bench_logging.txt` in your present working directory:

- `export ROCBLAS_LOG_BENCH_PATH=$PWD/bench_logging.txt`

Note that a full path is required, not a relative path. In the above command `$PWD` expands to the full path of your present working directory. If paths are not set, then the logging output is streamed to standard error.

When profile logging is enabled, memory usage increases. If the program exits abnormally, then it is possible that profile logging will not be outputted before the program exits.

References:

PROGRAMMER'S GUIDE

6.1 Library Source Code Organization

The rocBLAS code is split into three major parts:

- The *library* directory contains all source code for the library.
- The *clients* directory contains all test code and code to build clients.
- Infrastructure.

6.1.1 The *library* Directory

The *library* directory contains all source code for the library.

library/include

Contains C98 include files for the external API. These files also contain Doxygen comments that document the API.

library/src/blas[1,2,3]

Source code for Level 1, 2, and 3 BLAS functions in *.cpp* and *.hpp* files.

- The *.cpp* files contain
 - External C functions that call templated functions with an *_impl* extension
 - The *_impl* functions have argument checking and logging, and they in turn call functions with a *_template* extension
- The *_kernels.cpp* files contain
 - *_template* functions that set up the workgroup and call HIP launch to run *_kernel* functions
 - *_kernel* functions that run on the device

library/src/blas3/Tensile

Code for calling Tensile from rocBLAS, and YAML files with Tensile tuning configurations

library/src/blas_ex

Source code for mixed precision BLAS

library/src/include

Internal include files for:

- Handle code
- Device memory allocation
- Logging
- Numerical checking
- Utility code

6.1.2 The *clients* Directory

The *clients* directory contains all test code and code to build clients.

clients/gtest

Code for client rocbblas-test. This client is used to test rocBLAS.

clients/benchmarks

Code for client rocbblas-benchmark. This client is used to benchmark rocBLAS functions.

clients/include

Code for testing and benchmarking individual rocBLAS functions, and utility code for testing

clients/common

Common code used by both rocbblas-benchmark and rocbblas-test

clients/samples

Sample code for calling rocBLAS functions

6.1.3 Infrastructure

- CMake is used to build and package rocBLAS. There are CMakeLists.txt files throughout the code.
- Doxygen/Breathe/Sphinx/ReadTheDocs are used to produce documentation. Content for the documentation is from:
 - Doxygen comments in include files in the directory library/include
 - Files in the directory docs/source.
- Jenkins is used to automate Continuous Integration testing.
- clang-format is used to format C++ code.

6.2 Handle, Stream, and Device Management

6.2.1 Handle

A rocBLAS handle must be created before calling other rocBLAS functions. This can be done with:

```
rocblas_handle handle;
if(rocblas_create_handle(&handle) != rocblas_status_success) return EXIT_FAILURE;
```

The created handle should be destroyed when the users have completed calling rocBLAS functions. This can be done with:

```
if(rocblas_destroy_handle(handle) != rocblas_status_success) return EXIT_FAILURE;
```

The above-created handle will use the default stream and the default device. If the user wants the non-default stream and the non-default device, then call:

```
int deviceId = non_default_device_id;
if(hipSetDevice(deviceId) != hipSuccess) return EXIT_FAILURE;

//optional call to rocblas_initialize
rocblas_initialize();

// note the order, call hipSetDevice before hipStreamCreate
hipStream_t stream;
if(hipStreamCreate(&stream) != hipSuccess) return EXIT_FAILURE;

rocblas_handle handle;
if(rocblas_create_handle(&handle) != rocblas_status_success) return EXIT_FAILURE;

if(rocblas_set_stream(handle, stream) != rocblas_status_success) return EXIT_FAILURE;
```

For the library to use a non-default device within a host thread, the device must be set using hipSetDevice() before creating the handle.

The device in the host thread should not be changed between `hipStreamCreate` and `hipStreamDestroy`. If the device in the host thread is changed between creating and destroying, then the stream the behavior is undefined.

If the user created a non-default stream, it is the user's responsibility to synchronize the non-default stream before destroying it:

```
// Synchronize the non-default stream before destroying it
if(hipStreamSynchronize(stream) != hipSuccess) return EXIT_FAILURE;

if(hipStreamDestroy(stream) != hipSuccess) return EXIT_FAILURE;
```

When a user changes the stream from one non-default stream to another non-default stream, it is the user's responsibility to synchronize the old stream before setting the new stream. Then, the user can optionally destroy the old stream:

```
// Synchronize the old stream
if(hipStreamSynchronize(old_stream) != hipSuccess) return EXIT_FAILURE;

// Destroy the old stream (this step is optional but must come after synchronization)
if(hipStreamDestroy(old_stream) != hipSuccess) return EXIT_FAILURE;

// Create a new stream (this step can be done before the steps above)
if(hipStreamCreate(&new_stream) != hipSuccess) return EXIT_FAILURE;

// Set the handle to use the new stream (must come after synchronization)
if(rocblas_set_stream(handle, new_stream) != rocblas_status_success) return EXIT_FAILURE;
```

The above `hipStreamSynchronize` is necessary because the rocBLAS handle contains allocated device memory that must not be shared by multiple asynchronous streams at the same time.

If either the old or new stream is the default (NULL) stream, it is not necessary to synchronize the old stream before destroying it, or before setting the new stream, because the synchronization is implicit.

Note: A user can switch from one non-default stream to another without calling `hipStreamSynchronize()` by enabling stream-order memory allocation. Refer to section [Stream-Ordered Memory Allocation](#).

Creating the handle will incur a startup cost. There is an additional startup cost for gemm functions to load gemm kernels for a specific device. Users can shift the gemm startup cost to occur after setting the device by calling `rocblas_initialize()` after calling `hipSetDevice()`. This action needs to be done once for each device. If the user has two rocBLAS handles which use the same device, then the user only needs to call `rocblas_initialize()` once. If `rocblas_initialize()` is not called, then the first gemm call will have the startup cost.

The rocBLAS handle stores the following:

- Stream
- Logging mode
- Pointer mode
- Atomics mode

6.2.2 Stream and Device Management

HIP kernels are launched in a queue. This queue is otherwise known as a stream. A stream is a queue of work on a particular device.

A rocBLAS handle always has one stream, and a stream is always associated with one device. Furthermore, the rocBLAS handle is passed as an argument to all rocBLAS functions that launch kernels, and these kernels are launched in that handle's stream to run on that stream's device.

If the user does not create a stream, then the rocBLAS handle uses the default (NULL) stream, maintained by the system. Users cannot create or destroy the default stream. However, users can create a new non-default stream and bind it to the rocBLAS handle with the two commands: `hipStreamCreate()` and `rocblas_set_stream()`.

If the user creates a stream, they are responsible for destroying it with `hipStreamDestroy()`. If the handle is switching from one non-default stream to another, then the old stream needs to be synchronized. Next, the user needs to create and set the new non-default stream using `hipStreamCreate()` and `rocblas_set_stream()`, respectively. Then the user can optionally destroy the old stream.

HIP has two important device management functions, `hipSetDevice()`, and `hipGetDevice()`.

- `hipSetDevice()`: Set default device to be used for subsequent hip API calls from this thread.
- `hipGetDevice()`: Return the default device id for the calling host thread.

The device which was set using `hipSetDevice()` at the time of calling `hipStreamCreate()` is the one that is associated with a stream. But, if the device was not set using `hipSetDevice()`, then, the default device will be used.

Users cannot switch the device in a stream between `hipStreamCreate()` and `hipStreamDestroy()`. If users want to use another device, they should create another stream.

rocBLAS never sets a device, it only queries using `hipGetDevice()`. If rocBLAS does not see a valid device, it returns an error message to users.

6.2.3 Multiple Streams and Multiple Devices

If a machine has `num` GPU devices, they will have `deviceId` numbers 0, 1, 2, ... (`num - 1`). The default device has `deviceId == 0`. Each rocBLAS handle can only be used with a single device, but users can run `num` rocBLAS handles on `num` devices concurrently.

6.3 Device Memory Allocation

6.3.1 Requirements

- Some rocBLAS functions need temporary device memory.
- Allocating and deallocating device memory is expensive and synchronizing.
- Temporary device memory should be recycled across multiple rocBLAS function calls using the same `rocblas_handle`.
- The following schemes need to be supported:
 - **Default** Functions allocate required device memory automatically. This has the disadvantage that allocation is a synchronizing event.
 - **Preallocate** Query all the functions called using a `rocblas_handle` to find out how much device memory is needed. Preallocate the required device memory when the `rocblas_handle` is created, and there are no more synchronizing allocations or deallocations.

- **Manual** Query a function to find out how much device memory is required. Allocate and deallocate the device memory before and after function calls. This allows the user to control where the synchronizing allocation and deallocation occur.

In all above schemes, temporary device memory needs to be held by the `rocbblas_handle` and recycled if a subsequent function using the handle needs it.

6.3.2 Design

- rocBLAS uses per-handle device memory allocation with out-of-band management.
- The state of the device memory is stored in the `rocbblas_handle`.
- For the user of rocBLAS:
 - Functions are provided to query how much device memory a function needs.
 - An environment variable is provided to preallocate when the `rocbblas_handle` is created.
 - Functions are provided to manually allocate and deallocate after the `rocbblas_handle` is created.
 - The following two values are added to the `rocbblas_status` enum to indicate how a rocBLAS function is changing the state of the temporary device memory in the rocBLAS handle :
 - * `rocbblas_status_size_unchanged`
 - * `rocbblas_status_size_increased`
- For the rocBLAS developer:
 - Functions are provided to answer device memory size queries.
 - Functions are provided to allocate temporary device memory.
 - opaque RAI objects are used to hold the temporary device memory, and allocated memory is returned to the handle automatically when it is no longer needed.

The functions for the rocBLAS user are described in the User Guide. The functions for the rocBLAS developer are described below.

6.3.3 Answering Device Memory Size Queries in Function That Needs Memory

Example

Functions should contain code like below to answer a query on how much temporary device memory is required. In this case, $m * n * \text{sizeof}(T)$ bytes of memory is required:

```
rocbblas_status rocbblas_function(rocbblas_handle handle, ...)
{
    if(!handle) return rocbblas_status_invalid_handle;

    if (handle->is_device_memory_size_query())
    {
        size_t size = m * n * sizeof(T);
        return handle->set_optimal_device_memory_size(size);
    }

    // rest of function
}
```

Function

```
bool _roclblas_handle::is_device_memory_size_query() const
```

Indicates if the current function call is collecting information about the optimal device memory allocation size

return value:

- **true** if information is being collected
- **false** if information is not being collected

Function

```
roclblas_status _roclblas_handle::set_optimal_device_memory_size(size...)
```

Sets the optimal size(s) of device memory buffer(s) in bytes for this function. The sizes are rounded up to the next multiple of 64 (or some other chunk size), and the running maximum is updated.

return value:

- **roclblas_status_size_unchanged** If the maximum optimal device memory size did not change, this is the case where the function does not use device memory.
- **roclblas_status_size_increased** If the maximum optimal device memory size increased.
- **roclblas_status_internal_error** If this function is not supposed to be collecting size information.

Function

```
size_t roclblas_sizeof_datatype(roclblas_datatype type)
```

Returns size of a rocBLAS runtime data type

6.3.4 Answering Device Memory Size Queries in Function That Does Not Need Memory

Example

```
roclblas_status roclblas_function(roclblas_handle handle, ...)
{
    if(!handle) return roclblas_status_invalid_handle;

    RETURN_ZERO_DEVICE_MEMORY_SIZE_IF_QUERIED(handle);

    // rest of function
}
```

Macro

```
RETURN_ZERO_DEVICE_MEMORY_SIZE_IF_QUERIED(handle)
```

A convenience macro that returns `rocblas_status_size_unchanged` if the function call is a memory size query

6.3.5 rocBLAS Kernel Device Memory Allocation

Example

Device memory can be allocated for `n` floats using `device_malloc` as follows:

```
auto workspace = handle->device_malloc(n * sizeof(float));  
if (!workspace) return rocblas_status_memory_error;  
float* ptr = static_cast<float*>(workspace);
```

Example

To allocate multiple buffers:

```
size_t size1 = m * n;  
size_t size2 = m * k;  
  
auto workspace = handle->device_malloc(size1, size2);  
if (!workspace) return rocblas_status_memory_error;  
  
void * w_buf1, * w_buf2;  
w_buf1 = workspace[0];  
w_buf2 = workspace[1];
```

Function

```
auto workspace = handle->device_malloc(size...)
```

- Returns an opaque RAII object lending allocated device memory to a particular rocBLAS function.
- The object returned is convertible to `void *` or other pointer types if only one size is specified.
- The individual pointers can be accessed with the subscript operator `[]`.
- The lifetime of the returned object is the lifetime of the borrowed device memory (RAII).
- To simplify and optimize the code, only one successful allocation object can be alive at a time.
- If the handle's device memory is currently being managed by rocBLAS, as in the default scheme, it is expanded in size as necessary.
- If the user allocated (or pre-allocated) an explicit size of device memory, then that size is used as the limit, and no resizing or synchronization ever occurs.

Parameters:

- **size** size in bytes of memory to be allocated

return value:

- **On success**, returns an opaque RAII object that evaluates to `true` when converted to `bool`
- **On failure**, returns an opaque RAII object that evaluates to `false` when converted to `bool`

6.3.6 Performance Degrade

The `rocblas_status` enum value `rocblas_status_perf_degraded` is used to indicate that a slower algorithm was used because of insufficient device memory for the optimal algorithm.

Example

```
rocblas_status ret = rocblas_status_success;
size_t size_for_optimal_algorithm = m + n + k;
size_t size_for_degraded_algorithm = m;
auto workspace_optimal = handle->device_malloc(size_for_optimal_algorithm);
if (workspace_optimal)
{
    // Algorithm using larger optimal memory
}
else
{
    auto workspace_degraded = handle->device_malloc(size_for_degraded_algorithm);
    if (workspace_degraded)
    {
        // Algorithm using smaller degraded memory
        ret = rocblas_status_perf_degraded;
    }
    else
    {
        // Not enough device memory for either optimal or degraded algorithm
        ret = rocblas_status_memory_error;
    }
}
return ret;
```

6.4 Thread Safe Logging

rocBLAS has thread safe logging. This prevents garbled output when multiple threads are writing to the same file.

Thread safe logging is obtained from using `rocblas_internal_ostream`, a class that can be used similarly to `std::ostream`. It provides standardized methods for formatted output to either strings or files. The default constructor of `rocblas_internal_ostream` writes to strings, which are thread-safe because they are owned by the calling thread. There are also `rocblas_internal_ostream` constructors for writing to files. The `rocblas_internal_ostream::yaml_on` and `rocblas_internal_ostream::yaml_off` IO modifiers turn YAML formatting mode on and off.

`rocblas_cout` and `rocblas_cerr` are the thread-safe versions of `std::cout` and `std::cerr`.

Many output identifiers have been marked “poisoned” in `rocblas-test` and `rocblas-bench`, to catch the use of non-thread-safe IO. These include `std::cout`, `std::cerr`, `printf`, `fprintf`, `fputs`, `puts`, and others. The poisoning is not turned on in the library itself or in the samples, because we cannot impose restrictions on the use of these symbols on outside users.

`rocblas_handle` contains three `rocblas_internal_ostream` pointers for logging output:

- `static rocblas_internal_ostream* log_trace_os`
- `static rocblas_internal_ostream* log_bench_os`
- `static rocblas_internal_ostream* log_profile_os`

The user can also create `rocblas_internal_ostream` pointers/objects outside of the handle.

Each `rocblas_internal_ostream` associated with a file points to a single `rocblas_internal_ostream::worker` with a `std::shared_ptr`, for writing to the file. The worker is mapped from the device id and inode corresponding to the file. More than one `rocblas_internal_ostream` can point to the same worker.

This means if more than one `rocblas_internal_ostream` is writing to a single output file, they will share the same `rocblas_internal_ostream::worker`.

The `<<` operator for `rocblas_internal_ostream` is overloaded. Output is first accumulated in `rocblas_internal_ostream::os`, a `std::ostringstream` buffer. Each `rocblas_internal_ostream` has its own `os` `std::ostringstream` buffer, so strings in `os` will not be garbled.

When `rocblas_internal_ostream.os` is flushed with either a `std::endl` or an explicit flush of `rocblas_internal_ostream`, then `rocblas_internal_ostream::worker::send` pushes the string contents of `rocblas_internal_ostream.os` and a promise, the pair being called a task, onto `rocblas_internal_ostream.worker.queue`.

The `send` function uses promise/future to asynchronously transfer data from `rocblas_internal_ostream.os` to `rocblas_internal_ostream.worker.queue`, and to wait for the worker to finish writing the string to the file. It also locks a mutex to make sure the push of the task onto the queue is atomic.

The `ostream.worker.queue` will contain a number of tasks. When `rocblas_internal_ostream` is destroyed, all the `tasks.string` in `rocblas_internal_ostream.worker.queue` are printed to the `rocblas_internal_ostream` file, the `std::shared_ptr` to the `ostream.worker` is destroyed, and if the reference count to the worker becomes 0, the worker's thread is sent a 0-length string to tell it to exit.

6.5 rocBLAS Numerical Checking

Note that performance will degrade when numerical checking is enabled.

rocBLAS provides the environment variable `ROCBLAS_CHECK_NUMERICS`, which allows users to debug numerical abnormalities. Setting a value of `ROCBLAS_CHECK_NUMERICS` enables checks on the input and the output vectors/matrices of the rocBLAS functions for (not-a-number) NaN's, zeros, infinities, and denormal/subnormal values. Numerical checking is available to check the input and the output vectors for all level 1 and 2 functions. In level 2 functions, only the general (ge) type input and the output matrix can be checked for numerical abnormalities. In level 3, GEMM is the only function to have numerical checking.

`ROCBLAS_CHECK_NUMERICS` is a bitwise OR of zero or more bit masks as follows:

- `ROCBLAS_CHECK_NUMERICS = 0`: is not set, then there is no numerical checking
- `ROCBLAS_CHECK_NUMERICS = 1`: fully informative message, prints the results of numerical checking whether the input and the output Matrices/Vectors have NaN's/zeros/infinities/denormal values to the console
- `ROCBLAS_CHECK_NUMERICS = 2`: prints result of numerical checking only if the input and the output Matrices/Vectors has a NaN/infinity/denormal value
- `ROCBLAS_CHECK_NUMERICS = 4`: return `rocblas_status_check_numeric_fail` status if there is a NaN/infinity/denormal value

An example usage of `ROCBLAS_CHECK_NUMERICS` is shown below,

```
ROCBLAS_CHECK_NUMERICS=4 ./rocblas-bench -f gemm -i 1 -j 0
```

The above command will return a `rocblas_status_check_numeric_fail` if the input and the output matrices of BLAS level 3 GEMM function has a NaN/infinity/denormal value. If there are no numerical abnormalities, then `rocblas_status_success` is returned.

6.6 rocBLAS Order of Argument Checking and Logging

6.6.1 Legacy BLAS

Legacy BLAS has two types of argument checking:

- Error-return for incorrect argument (Legacy BLAS implement this with a call to the function XERBLA)
- Quick-return-success when an argument allows for the subprogram to be a no-operation or a constant result

Level 2 and Level 3 BLAS subprograms have both error-return and quick-return-success. Level 1 BLAS subprograms have only quick-return-success

6.6.2 rocBLAS

rocBLAS has 5 types of argument checking:

- `rocblas_status_invalid_handle` if the handle is a NULL pointer
- `rocblas_status_invalid_size` for invalid size, increment or leading dimension argument
- `rocblas_status_invalid_value` for unsupported enum value
- `rocblas_status_success` for quick-return-success
- `rocblas_status_invalid_pointer` for NULL argument pointers

6.6.3 rocBLAS has the Following Differences When Compared To Legacy BLAS

- It is a C API, returning a `rocblas_status` type indicating the success of the call.
- In legacy BLAS, the following functions return a scalar result: `dot`, `nrm2`, `asum`, `amax`, and `amin`. In rocBLAS, a pointers to scalar return value is passed as the last argument.
- The first argument is a `rocblas_handle` argument, an opaque pointer to rocBLAS resources, corresponding to a single HIP stream.
- Scalar arguments like `alpha` and `beta` are pointers on either the host or device, controlled by the rocBLAS handle's pointer mode. In cases where the other arguments do not dictate an early return, if the `alpha` and `beta` pointers are NULL the function will return `rocblas_status_invalid_pointer`.
- Vector and matrix arguments are always pointers to device memory.
- When `rocblas_pointer_mode == rocblas_pointer_mode_host` `alpha` and `beta` values are inspected and based on their values it is determined which vector and matrix pointers must be dereferenced. If these pointers will be dereferenced a NULL pointer will lead to a return value `rocblas_status_invalid_pointer`.
- Otherwise if `rocblas_pointer_mode == rocblas_pointer_mode_device` we do NOT check if these vector or matrix pointers will dereference a NULL pointer as we do not want to slow execution to fetch and inspect `alpha` and `beta` values.
- The `ROCBLAS_LAYER` environment variable controls the option to log argument values.
- There is added functionality like

- batched
- strided_batched
- mixed precision in `gemm_ex`, `gemm_batched_ex`, and `gemm_strided_batched_ex`

6.6.4 To Accommodate the Additions

- See Logging below.
- For batched and strided_batched L2 and L3 functions, there is a quick-return-success for `batch_count == 0`, and an invalid size error for `batch_count < 0`.
- For batched and strided_batched L1 functions, there is a quick-return-success for `batch_count <= 0`
- When `rocblas_pointer_mode == rocblas_pointer_mode_device` alpha and beta are not copied from device to host for quick-return-success checks. In this case, the quick-return-success checks are omitted. This will still give a correct result, but the operation will be slower.
- For strided_batched functions there is no argument checking for stride. To access elements in a strided_batched_matrix, for example the C matrix in `gemm`, the zero based index is calculated as `i1 + i2 * ldc + i3 * stride_c`, where `i1 = 0, 1, 2, ..., m-1`; `i2 = 0, 1, 2, ..., n-1`; `i3 = 0, 1, 2, ..., batch_count - 1`. An incorrect stride can result in a core dump due a segmentation fault. It can also produce an indeterminate result if there is a memory overlap in the output matrix between different values of `i3`.

6.6.5 Device Memory Size Queries

- When `handle->is_device_memory_size_query()` is true, the call is not a normal call, but it is a device memory size query.
- No logging should be performed during device memory size queries.
- If the rocBLAS kernel requires no temporary device memory, the macro `RETURN_ZERO_DEVICE_MEMORY_SIZE_IF_QUERIED(handle)` can be called after checking that `handle != nullptr`.
- If the rocBLAS kernel requires temporary device memory, then it should be set, and the kernel returned, by calling `return handle->set_optimal_device_memory_size(size...)`, where `size...` is a list of one or more sizes for different sub-problems. The sizes are rounded up and added.

Logging

- There is logging before a quick-return-success or error-return, except:
 - When `handle == nullptr`, return `rocblas_status_invalid_handle`.
 - When `handle->is_device_memory_size_query()` returns true.
- Vectors and matrices are logged with their addresses and are always on device memory.
- Scalar values in device memory are logged as their addresses. Scalar values in host memory are logged as their values, with a `nullptr` logged as NaN (`std::numeric_limits<T>::quiet_NaN()`).

6.6.6 rocBLAS Control Flow

1. If `handle == nullptr`, then return `rocblas_status_invalid_handle`.
2. If the function does not require temporary device memory, then call the macro `RETURN_ZERO_DEVICE_MEMORY_SIZE_IF_QUERIED(handle);`.
3. If the function requires temporary device memory, and `handle->is_device_memory_size_query()` is true, then validate any pointers and arguments required to determine the optimal size of temporary device memory, returning `rocblas_status_invalid_pointer` or `rocblas_status_invalid_size` if the arguments are invalid, and otherwise return `handle->set_optimal_device_memory_size(size...);`, where `size...` is a list of one or more sizes of temporary buffers, which are allocated with `handle->device_malloc(size...)` later.
4. Perform logging if enabled, taking care not to dereference `nullptr` arguments.
5. Check for unsupported enum value. Return `rocblas_status_invalid_value` if enum value is invalid.
6. Check for invalid sizes. Return `rocblas_status_invalid_size` if size arguments are invalid.
7. Return `rocblas_status_invalid_pointer` if any pointers used to determine quick return conditions are NULL.
8. If quick return conditions are met:
 - If there is no return value
 - Return `rocblas_status_success`
 - If there is a return value
 - If the return value pointer argument is `nullptr`, return `rocblas_status_invalid_pointer`
 - Else, return `rocblas_status_success`
9. If any pointers not checked in #7 are NULL and MUST be dereferenced return `rocblas_status_invalid_pointer`; only when in `rocblas_pointer_mode == rocblas_pointer_mode_host` can it be determined efficiently if some vector/matrix arguments must be dereferenced.
10. (Optional.) Allocate device memory, returning `rocblas_status_memory_error` if the allocation fails.
11. If all checks above pass, launch the kernel and return `rocblas_status_success`.

6.6.7 Legacy L1 BLAS “single vector”

Below are four code snippets from NETLIB for “single vector” legacy L1 BLAS. They have quick-return-success for `(n <= 0) || (incx <= 0)`:

```
DOUBLE PRECISION FUNCTION DASUM(N,DX,INCX)
IF (N.LE.0 .OR. INCX.LE.0) RETURN

DOUBLE PRECISION FUNCTION DNRM2(N,X,INCX)
IF (N.LT.1 .OR. INCX.LT.1) THEN
    return = ZERO

SUBROUTINE DSCAL(N,DA,DX,INCX)
IF (N.LE.0 .OR. INCX.LE.0) RETURN

INTEGER FUNCTION IDAMAX(N,DX,INCX)
```

(continues on next page)

(continued from previous page)

```

IDAMAX = 0
IF (N.LT.1 .OR. INCX.LE.0) RETURN
IDAMAX = 1
IF (N.EQ.1) RETURN

```

6.6.8 Legacy L1 BLAS “two vector”

Below are seven legacy L1 BLAS codes from NETLIB. There is quick-return-success for ($n \leq 0$). In addition, for DAXPY, there is quick-return-success for ($\alpha == 0$):

```

SUBROUTINE DAXPY(N,alpha,DX,INCX,DY,INCY)
IF (N.LE.0) RETURN
IF (alpha.EQ.0.0d0) RETURN

SUBROUTINE DCOPY(N,DX,INCX,DY,INCY)
IF (N.LE.0) RETURN

DOUBLE PRECISION FUNCTION DDOT(N,DX,INCX,DY,INCY)
IF (N.LE.0) RETURN

SUBROUTINE DROT(N,DX,INCX,DY,INCY,C,S)
IF (N.LE.0) RETURN

SUBROUTINE DSWAP(N,DX,INCX,DY,INCY)
IF (N.LE.0) RETURN

DOUBLE PRECISION FUNCTION DSDOT(N,SX,INCX,SY,INCY)
IF (N.LE.0) RETURN

SUBROUTINE DROTM(N,DX,INCX,DY,INCY,DPARAM)
DFLAG = DPARAM(1)
IF (N.LE.0 .OR. (DFLAG+TWO.EQ.ZERO)) RETURN

```

6.6.9 Legacy L2 BLAS

Below are code snippets from NETLIB for legacy L2 BLAS. They have both argument checking and quick-return-success:

```

SUBROUTINE DGER(M,N,ALPHA,X,INCX,Y,INCY,A,LDA)
INFO = 0
IF (M.LT.0) THEN
    INFO = 1
ELSE IF (N.LT.0) THEN
    INFO = 2
ELSE IF (INCX.EQ.0) THEN
    INFO = 5
ELSE IF (INCY.EQ.0) THEN
    INFO = 7
ELSE IF (LDA.LT.MAX(1,M)) THEN
    INFO = 9

```

(continues on next page)

(continued from previous page)

```

END IF
IF (INFO.NE.0) THEN
    CALL XERBLA('DGER ',INFO)
    RETURN
END IF

IF ((M.EQ.0) .OR. (N.EQ.0) .OR. (ALPHA.EQ.ZERO)) RETURN

```

```

SUBROUTINE DSYR(UPLO,N,ALPHA,X,INCX,A,LDA)

INFO = 0
IF (.NOT.LSAME(UPLO,'U') .AND. .NOT.LSAME(UPLO,'L')) THEN
    INFO = 1
ELSE IF (N.LT.0) THEN
    INFO = 2
ELSE IF (INCX.EQ.0) THEN
    INFO = 5
ELSE IF (LDA.LT.MAX(1,N)) THEN
    INFO = 7
END IF
IF (INFO.NE.0) THEN
    CALL XERBLA('DSYR ',INFO)
    RETURN
END IF

IF ((N.EQ.0) .OR. (ALPHA.EQ.ZERO)) RETURN

```

```

SUBROUTINE DGEMV(TRANS,M,N,ALPHA,A,LDA,X,INCX,BETA,Y,INCY)

INFO = 0
IF (.NOT.LSAME(TRANS,'N') .AND. .NOT.LSAME(TRANS,'T') .AND. .NOT.LSAME(TRANS,'C')) THEN
    INFO = 1
ELSE IF (M.LT.0) THEN
    INFO = 2
ELSE IF (N.LT.0) THEN
    INFO = 3
ELSE IF (LDA.LT.MAX(1,M)) THEN
    INFO = 6
ELSE IF (INCX.EQ.0) THEN
    INFO = 8
ELSE IF (INCY.EQ.0) THEN
    INFO = 11
END IF
IF (INFO.NE.0) THEN
    CALL XERBLA('DGEMV ',INFO)
    RETURN
END IF

IF ((M.EQ.0) .OR. (N.EQ.0) .OR. ((ALPHA.EQ.ZERO) .AND. (BETA.EQ.ONE))) RETURN

```

```

SUBROUTINE DTRSV(UPLO,TRANS,DIAG,N,A,LDA,X,INCX)

```

(continues on next page)

(continued from previous page)

```

INFO = 0
IF (.NOT.LSAME(UPLO,'U') .AND. .NOT.LSAME(UPLO,'L')) THEN
    INFO = 1
ELSE IF (.NOT.LSAME(TRANS,'N') .AND. .NOT.LSAME(TRANS,'T') .AND. .NOT.LSAME(TRANS,'C'))
    THEN
    INFO = 2
ELSE IF (.NOT.LSAME(DIAG,'U') .AND. .NOT.LSAME(DIAG,'N')) THEN
    INFO = 3
ELSE IF (N.LT.0) THEN
    INFO = 4
ELSE IF (LDA.LT.MAX(1,N)) THEN
    INFO = 6
ELSE IF (INCX.EQ.0) THEN
    INFO = 8
END IF
IF (INFO.NE.0) THEN
    CALL XERBLA('DTRSV ',INFO)
    RETURN
END IF

IF (N.EQ.0) RETURN

```

6.6.10 Legacy L3 BLAS

Below is a code snippet from NETLIB for legacy L3 BLAS dgemm. It has both argument checking and quick-return-success:

```

SUBROUTINE DGEMM(TRANSA,TRANSB,M,N,K,ALPHA,A,LDA,B,LDB,BETA,C,LDC)

    NOTA = LSAME(TRANSA,'N')
    NOTB = LSAME(TRANSB,'N')
    IF (NOTA) THEN
        NROWA = M
        NCOLA = K
    ELSE
        NROWA = K
        NCOLA = M
    END IF
    IF (NOTB) THEN
        NROWB = K
    ELSE
        NROWB = N
    END IF

    // Test the input parameters.

    INFO = 0
    IF ((.NOT.NOTA) .AND. (.NOT.LSAME(TRANSA,'C')) .AND.
+      (.NOT.LSAME(TRANSA,'T')) THEN
        INFO = 1

```

(continues on next page)

(continued from previous page)

```

ELSE IF ((.NOT.NOTB) .AND. (.NOT.LSAME(TRANSB,'C')) .AND.
+      (.NOT.LSAME(TRANSB,'T')))) THEN
    INFO = 2
ELSE IF (M.LT.0) THEN
    INFO = 3
ELSE IF (N.LT.0) THEN
    INFO = 4
ELSE IF (K.LT.0) THEN
    INFO = 5
ELSE IF (LDA.LT.MAX(1,NROWA)) THEN
    INFO = 8
ELSE IF (LDB.LT.MAX(1,NROWB)) THEN
    INFO = 10
ELSE IF (LDC.LT.MAX(1,M)) THEN
    INFO = 13
END IF
IF (INFO.NE.0) THEN
    CALL XERBLA('DGEMM ',INFO)
    RETURN
END IF

// Quick return if possible.

IF ((M.EQ.0) .OR. (N.EQ.0) .OR. (((ALPHA.EQ.ZERO).OR. (K.EQ.0)).AND. (BETA.EQ.ONE)))
↳RETURN

```

6.7 rocBLAS Benchmarking and Testing

There are three client executables that can be used with rocBLAS. They are:

- rocblas-bench
- rocblas-gemm-tune
- rocblas-test

These three clients can be built by following the instructions in the Building and Installing section of the User Guide. After building the rocBLAS clients, they can be found in the directory rocBLAS/build/release/clients/staging.

The next three sections will cover a brief explanation and the usage of each rocBLAS client.

6.7.1 rocblas-bench

rocblas-bench is used to measure performance and verify the correctness of rocBLAS functions.

It has a command line interface. For more information:

```
rocBLAS/build/release/clients/staging/rocblas-bench --help
```

- The following table shows all the data types in rocBLAS:

Table 6.1: Data types in rocBLAS

Data type	acronym
real 16 bit Brain Floating Point	bf16_r
real half	f16_r (h)
real float	f32_r (s)
real double	f64_r (d)
Complex float	f32_c (c)
Complex double	f64_c (z)
Integer 32	i32_r
Integer 8	i8_r

- All options for problem types in rocBLAS for gemm are shown here:

N: not transposed

T: transposed

C: complex conjugate (for real data type C is the same as T)

Table 6.2: various matrix operations

Problem Types	problem_type	data type
NN	Cijk_Ailk_Bljk	real/complex
NT	Cijk_Ailk_Bjlk	real/complex
TN	Cijk_Alik_Bljk	real/complex
TT	Cijk_Alik_Bjlk	real/complex
NC	Cijk_Ailk_BjlkC	complex
CN	Cijk_AlikC_Bljk	complex
CC	Cijk_AlikC_BjlkC	complex
TC	Cijk_Alik_BjlkC	complex
CT	Cijk_AlikC_Bjlk	complex

For example, NT means $A * B^T$.

- Gemm functions can be divided into two main categories:
 1. HPA functions (HighPrecisionAccumulate) where the compute data type is different from the input data type (A/B). All HPA functions must be called using *gemm_ex* API in rocblas-bench (and not *gemm*). *gemm_ex* function name consists of three letters: A/B data type, C/D data type, compute data type.
 2. Non-HPA functions where the input (A/B), output (C/D), and compute data types are all the same. Non-HPA cases can be called using *gemm* or *gemm_ex*. But using *gemm* is recommended.

The following table shows all possible *gemm* functions in rocBLAS.

Table 6.3: all *gemm* functions in rocBLAS

function	Kernel name	A/B data type	C/D data type	compute data type
hgemm	<arch>_<problem_type>_HB	f16_r	f16_r	f16_r
hgemm_batched	<arch>_<problem_type>_HB_GB	f16_r	f16_r	f16_r
hgemm_strided_batched	<arch>_<problem_type>_HB	f16_r	f16_r	f16_r
sgemm	<arch>_<problem_type>_SB	f32_r	f32_r	f32_r
sgemm_batched	<arch>_<problem_type>_SB_GB	f32_r	f32_r	f32_r
sgemm_strided_batched	<arch>_<problem_type>_SB	f32_r	f32_r	f32_r
dgemm	<arch>_<problem_type>_DB	f64_r	f64_r	f64_r
dgemm_batched	<arch>_<problem_type>_DB_GB	f64_r	f64_r	f64_r
dgemm_strided_batched	<arch>_<problem_type>_DB	f64_r	f64_r	f64_r
cgemm	<arch>_<problem_type>_CB	f32_c	f32_c	f32_c
cgemm_batched	<arch>_<problem_type>_CB_GB	f32_c	f32_c	f32_c
cgemm_strided_batched	<arch>_<problem_type>_CB	f32_c	f32_c	f32_c
zgemm	<arch>_<problem_type>_ZB	f64_c	f64_c	f64_c
zgemm_batched	<arch>_<problem_type>_ZB_GB	f64_c	f64_c	f64_c
zgemm_strided_batched	<arch>_<problem_type>_ZB	f64_c	f64_c	f64_c
HHS	<arch>_<problem_type>_HHS_BH	f16_r	f16_r	f32_r
HHS_batched	<arch>_<problem_type>_HHS_BH_GB	f16_r	f16_r	f32_r
HHS_strided_batched	<arch>_<problem_type>_HHS_BH	f16_r	f16_r	f32_r
HSS	<arch>_<problem_type>_HSS_BH	f16_r	f32_r	f32_r
HSS_batched	<arch>_<problem_type>_HSS_BH_GB	f16_r	f32_r	f32_r
HSS_strided_batched	<arch>_<problem_type>_HSS_BH	f16_r	f32_r	f32_r
BBS	<arch>_<problem_type>_BBS_BH	bf16_r	bf16_r	f32_r
BBS_batched	<arch>_<problem_type>_BBS_BH_GB	bf16_r	bf16_r	f32_r
BBS_strided_batched	<arch>_<problem_type>_BBS_BH	bf16_r	bf16_r	f32_r
BSS	<arch>_<problem_type>_BSS_BH	bf16_r	f32_r	f32_r
BSS_batched	<arch>_<problem_type>_BSS_BH_GB	bf16_r	f32_r	f32_r
BSS_strided_batched	<arch>_<problem_type>_BSS_BH	bf16_r	f32_r	f32_r
I8II	<arch>_<problem_type>_I8II_BH	I8	I	I
I8II_batched	<arch>_<problem_type>_I8II_BH_GB	I8	I	I
I8II_strided_batched	<arch>_<problem_type>_I8II_BH	I8	I	I

- How to benchmark the performance of a gemm function using rocblas-bench:

This method is good only if you want to test a few sizes, otherwise, refer to the next section. The following listing shows how to configure rocblas-bench to call each of the gemm functions:

Non-HPA cases (gemm)

```
#dgemm
$ ./rocblas-bench -f gemm --transposeA N --transposeB T -m 1024 -n 2048 -k 512 -r d --
↳ lda 1024 --ldb 2048 --ldc 1024 --ldd 1024 --alpha 1.1 --beta 1.0
# dgemm batched
$ ./rocblas-bench -f gemm_batched --transposeA N --transposeB T -m 1024 -n 2048 -k 512 -
↳ r d --lda 1024 --ldb 2048 --ldc 1024 --ldd 1024 --alpha 1.1 --beta 1 --batch_count 5
# dgemm strided batched
$ ./rocblas-bench -f gemm_strided_batched --transposeA N --transposeB T -m 1024 -n 2048 -
↳ k 512 -r d --lda 1024 --stride_a 4096 --ldb 2048 --stride_b 4096 --ldc 1024 --stride_c_
↳ 2097152 --ldd 1024 --stride_d 2097152 --alpha 1.1 --beta 1 --batch_count 5

# sgemm
$ ./rocblas-bench -f gemm --transposeA N --transposeB T -m 1024 -n 2048 -k 512 -r s --
↳ lda 1024 --ldb 2048 --ldc 1024 --ldd 1024 --alpha 1.1 --beta 1
# sgemm batched
$ ./rocblas-bench -f gemm_batched --transposeA N --transposeB T -m 1024 -n 2048 -k 512 -
↳ r s --lda 1024 --ldb 2048 --ldc 1024 --ldd 1024 --alpha 1.1 --beta 1 --batch_count 5
# sgemm strided batched
$ ./rocblas-bench -f gemm_strided_batched --transposeA N --transposeB T -m 1024 -n 2048 -
↳ k 512 -r s --lda 1024 --stride_a 4096 --ldb 2048 --stride_b 4096 --ldc 1024 --stride_c_
↳ 2097152 --ldd 1024 --stride_d 2097152 --alpha 1.1 --beta 1 --batch_count 5

# hgemm (this function is not really very fast. Use HHS instead, which is faster and_
↳ more accurate)
$ ./rocblas-bench -f gemm --transposeA N --transposeB T -m 1024 -n 2048 -k 512 -r h --
↳ lda 1024 --ldb 2048 --ldc 1024 --ldd 1024 --alpha 1.1 --beta 1
# hgemm batched
$ ./rocblas-bench -f gemm_batched --transposeA N --transposeB T -m 1024 -n 2048 -k 512 -
↳ r h --lda 1024 --ldb 2048 --ldc 1024 --ldd 1024 --alpha 1.1 --beta 1 --batch_count 5
# hgemm strided batched
$ ./rocblas-bench -f gemm_strided_batched --transposeA N --transposeB T -m 1024 -n 2048 -
↳ k 512 -r h --lda 1024 --stride_a 4096 --ldb 2048 --stride_b 4096 --ldc 1024 --stride_c_
↳ 2097152 --ldd 1024 --stride_d 2097152 --alpha 1.1 --beta 1 --batch_count 5

# cgemm
$ ./rocblas-bench -f gemm --transposeA N --transposeB T -m 1024 -n 2048 -k 512 -r c --
↳ lda 1024 --ldb 2048 --ldc 1024 --ldd 1024 --alpha 1.1 --beta 1
# cgemm batched
$ ./rocblas-bench -f gemm_batched --transposeA N --transposeB T -m 1024 -n 2048 -k 512 -
↳ r c --lda 1024 --ldb 2048 --ldc 1024 --ldd 1024 --alpha 1.1 --beta 1 --batch_count 5
# cgemm strided batched
$ ./rocblas-bench -f gemm_strided_batched --transposeA N --transposeB T -m 1024 -n 2048 -
↳ k 512 -r c --lda 1024 --stride_a 4096 --ldb 2048 --stride_b 4096 --ldc 1024 --stride_c_
↳ 2097152 --ldd 1024 --stride_d 2097152 --alpha 1.1 --beta 1 --batch_count 5

# zgemm
$ ./rocblas-bench -f gemm --transposeA N --transposeB T -m 1024 -n 2048 -k 512 -r z --
```

(continues on next page)

(continued from previous page)

```

↪ lda 1024 --ldb 2048 --ldc 1024 --ldd 1024 --alpha 1.1 --beta 1
# zgemv batched
$ ./rocblas-bench -f gemm_batched --transposeA N --transposeB T -m 1024 -n 2048 -k 512 -
↪ r z --lda 1024 --ldb 2048 --ldc 1024 --ldd 1024 --alpha 1.1 --beta 1 --batch_count 5
# zgemv strided batched
$ ./rocblas-bench -f gemm_strided_batched --transposeA N --transposeB T -m 1024 -n 2048 -
↪ k 512 -r z --lda 1024 --stride_a 4096 --ldb 2048 --stride_b 4096 --ldc 1024 --stride_c_
↪ 2097152 --ldd 1024 --stride_d 2097152 --alpha 1.1 --beta 1 --batch_count 5

# cgemv (NC)
$ ./rocblas-bench -f gemm --transposeA N --transposeB C -m 1024 -n 2048 -k 512 -r c --
↪ lda 1024 --ldb 2048 --ldc 1024 --ldd 1024 --alpha 1.1 --beta 1
# cgemv batched (NC)
$ ./rocblas-bench -f gemm_batched --transposeA N --transposeB C -m 1024 -n 2048 -k 512 -
↪ r c --lda 1024 --ldb 2048 --ldc 1024 --ldd 1024 --alpha 1.1 --beta 1 --batch_count 5
# cgemv strided batched (NC)
$ ./rocblas-bench -f gemm_strided_batched --transposeA N --transposeB C -m 1024 -n 2048 -
↪ k 512 -r c --lda 1024 --stride_a 4096 --ldb 2048 --stride_b 4096 --ldc 1024 --stride_c_
↪ 2097152 --ldd 1024 --stride_d 2097152 --alpha 1.1 --beta 1 --batch_count 5

```

HPA cases (gemm_ex)

```
# HHS
$ ./rocblas-bench -f gemm_ex --transposeA N --transposeB T -m 1024 -n 2048 -k 512 --a_
↳ type h --lda 1024 --b_type h --ldb 2048 --c_type h --ldc 1024 --d_type h --ldd 1024 --
↳ compute_type s --alpha 1.1 --beta 1
# HHS batched
$ ./rocblas-bench -f gemm_batched_ex --transposeA N --transposeB T -m 1024 -n 2048 -k_
↳ 512 --a_type h --lda 1024 --b_type h --ldb 2048 --c_type h --ldc 1024 --d_type h --ldd_
↳ 1024 --compute_type s --alpha 1.1 --beta 1 --batch_count 5
# HHS strided batched
$ ./rocblas-bench -f gemm_strided_batched_ex --transposeA N --transposeB T -m 1024 -n_
↳ 2048 -k 512 --a_type h --lda 1024 --stride_a 4096 --b_type h --ldb 2048 --stride_b_
↳ 4096 --c_type h --ldc 1024 --stride_c 2097152 --d_type h --ldd 1024 --stride_d 2097152_
↳ --compute_type s --alpha 1.1 --beta 1 --batch_count 5

# HSS
$ ./rocblas-bench -f gemm_ex --transposeA N --transposeB T -m 1024 -n 2048 -k 512 --a_
↳ type h --lda 1024 --b_type h --ldb 2048 --c_type s --ldc 1024 --d_type s --ldd 1024 --
↳ compute_type s --alpha 1.1 --beta 1
# HSS batched
$ ./rocblas-bench -f gemm_batched_ex --transposeA N --transposeB T -m 1024 -n 2048 -k_
↳ 512 --a_type h --lda 1024 --b_type h --ldb 2048 --c_type s --ldc 1024 --d_type s --ldd_
↳ 1024 --compute_type s --alpha 1.1 --beta 1 --batch_count 5
# HSS strided batched
$ ./rocblas-bench -f gemm_strided_batched_ex --transposeA N --transposeB T -m 1024 -n_
↳ 2048 -k 512 --a_type h --lda 1024 --stride_a 4096 --b_type h --ldb 2048 --stride_b_
↳ 4096 --c_type s --ldc 1024 --stride_c 2097152 --d_type s --ldd 1024 --stride_d 2097152_
↳ --compute_type s --alpha 1.1 --beta 1 --batch_count 5

# BBS
$ ./rocblas-bench -f gemm_ex --transposeA N --transposeB T -m 1024 -n 2048 -k 512 --a_
↳ type bf16_r --lda 1024 --b_type bf16_r --ldb 2048 --c_type bf16_r --ldc 1024 --d_type_
↳ bf16_r --ldd 1024 --compute_type s --alpha 1.1 --beta 1
# BBS batched
$ ./rocblas-bench -f gemm_batched_ex --transposeA N --transposeB T -m 1024 -n 2048 -k_
↳ 512 --a_type bf16_r --lda 1024 --b_type bf16_r --ldb 2048 --c_type bf16_r --ldc 1024 --
↳ d_type bf16_r --ldd 1024 --compute_type s --alpha 1.1 --beta 1 --batch_count 5
# BBS strided batched
$ ./rocblas-bench -f gemm_strided_batched_ex --transposeA N --transposeB T -m 1024 -n_
↳ 2048 -k 512 --a_type bf16_r --lda 1024 --stride_a 4096 --b_type bf16_r --ldb 2048 --
↳ stride_b 4096 --c_type bf16_r --ldc 1024 --stride_c 2097152 --d_type bf16_r --ldd 1024_
↳ --stride_d 2097152 --compute_type s --alpha 1.1 --beta 1 --batch_count 5

# BSS
$ ./rocblas-bench -f gemm_ex --transposeA N --transposeB T -m 1024 -n 2048 -k 512 --a_
↳ type bf16_r --lda 1024 --b_type bf16_r --ldb 2048 --c_type s --ldc 1024 --d_type s --
↳ ldd 1024 --compute_type s --alpha 1.1 --beta 1
# BSS batched
$ ./rocblas-bench -f gemm_batched_ex --transposeA N --transposeB T -m 1024 -n 2048 -k_
↳ 512 --a_type bf16_r --lda 1024 --b_type bf16_r --ldb 2048 --c_type s --ldc 1024 --d_
↳ type s --ldd 1024 --compute_type s --alpha 1.1 --beta 1 --batch_count 5
# BSS strided batched
```

(continues on next page)

(continued from previous page)

```
$ ./rocblas-bench -f gemm_strided_batched_ex --transposeA N --transposeB T -m 1024 -n 2048 -k 512 --a_type bf16_r --lda 1024 --stride_a 4096 --b_type bf16_r --ldb 2048 --stride_b 4096 --c_type s --ldc 1024 --stride_c 2097152 --d_type s --ldd 1024 --stride_d 2097152 --compute_type s --alpha 1.1 --beta 1 --batch_count 5

# I8II
$ ./rocblas-bench -f gemm_ex --transposeA N --transposeB T -m 1024 -n 2048 -k 512 --a_type i8_r --lda 1024 --b_type i8_r --ldb 2048 --c_type i32_r --ldc 1024 --d_type i32_r --ldd 1024 --compute_type i32_r --alpha 1.1 --beta 1

# I8II batched
$ ./rocblas-bench -f gemm_batched_ex --transposeA N --transposeB T -m 1024 -n 2048 -k 512 --a_type i8_r --lda 1024 --b_type i8_r --ldb 2048 --c_type i32_r --ldc 1024 --d_type i32_r --ldd 1024 --compute_type i32_r --alpha 1.1 --beta 1 --batch_count 5

# I8II strided batched
$ ./rocblas-bench -f gemm_strided_batched_ex --transposeA N --transposeB T -m 1024 -n 2048 -k 512 --a_type i8_r --lda 1024 --stride_a 4096 --b_type i8_r --ldb 2048 --stride_b 4096 --c_type i32_r --ldc 1024 --stride_c 2097152 --d_type i32_r --ldd 1024 --stride_d 2097152 --compute_type i32_r --alpha 1.1 --beta 1 --batch_count 5
```

- How to set rocblas-bench parameters in a yaml file:

If you want to benchmark many sizes, it is recommended to use rocblas-bench with the batch call to eliminate the latency in loading the Tensile library which rocblas links to. The batch call takes a yaml file with a list of all problem sizes. You can have multiple sizes of different types in one yaml file. The benchmark setting is different from the direct call to the rocblas-bench. A sample setting for each function is listed below. Once you have the yaml file, you can benchmark the sizes as follows:

```
rocBLAS/build/release/clients/staging/rocblas-bench --yaml problem-sizes.yaml
```

Here are the configurations for each function:

Non-HPA cases (gemm)

```
# dgemm
- { rocblas_function: "rocblas_dgemm",          transA: "N", transB: "T", M: 1024, N: 1024, K: 512, lda: 1024, ldb: 2048, ldc: 1024, ldd: 1024, cold_iters: 2,
  ↪ iters: 10 }
# dgemm batched
- { rocblas_function: "rocblas_dgemm_batched", transA: "N", transB: "T", M: 1024, N: 1024, K: 512, lda: 1024, ldb: 2048, ldc: 1024, ldd: 1024, cold_iters: 2,
  ↪ iters: 10, batch_count: 5 }
# dgemm strided batched
- { rocblas_function: "rocblas_dgemm_strided_batched", transA: "N", transB: "T", M: 1024, N: 2048, K: 512, lda: 1024, ldb: 2048, ldc: 1024, ldd: 1024, cold_
  ↪ iters: 2, iters: 10, batch_count: 5, stride_a: 4096, stride_b: 4096, stride_c: 2097152,
  ↪ stride_d: 2097152 }

# sgemm
- { rocblas_function: "rocblas_sgemm",          transA: "N", transB: "T", M: 1024, N: 1024, K: 512, lda: 1024, ldb: 2048, ldc: 1024, ldd: 1024, cold_iters: 2,
  ↪ iters: 10 }
# sgemm batched
- { rocblas_function: "rocblas_sgemm_batched", transA: "N", transB: "T", M: 1024, N: 1024, K: 512, lda: 1024, ldb: 2048, ldc: 1024, ldd: 1024, cold_iters: 2,
  ↪ iters: 10, batch_count: 5 }
# sgemm strided batched
- { rocblas_function: "rocblas_sgemm_strided_batched", transA: "N", transB: "T", M: 1024, N: 2048, K: 512, lda: 1024, ldb: 2048, ldc: 1024, ldd: 1024, cold_
  ↪ iters: 2, iters: 10, batch_count: 5, stride_a: 4096, stride_b: 4096, stride_c: 2097152,
  ↪ stride_d: 2097152 }

# hgemm
- { rocblas_function: "rocblas_hgemm",          transA: "N", transB: "T", M: 1024, N: 1024, K: 512, lda: 1024, ldb: 2048, ldc: 1024, ldd: 1024, cold_iters: 2,
  ↪ iters: 10 }
# hgemm batched
- { rocblas_function: "rocblas_hgemm_batched", transA: "N", transB: "T", M: 1024, N: 1024, K: 512, lda: 1024, ldb: 2048, ldc: 1024, ldd: 1024, cold_iters: 2,
  ↪ iters: 10, batch_count: 5 }
# hgemm strided batched
- { rocblas_function: "rocblas_hgemm_strided_batched", transA: "N", transB: "T", M: 1024, N: 2048, K: 512, lda: 1024, ldb: 2048, ldc: 1024, ldd: 1024, cold_
  ↪ iters: 2, iters: 10, batch_count: 5, stride_a: 4096, stride_b: 4096, stride_c: 2097152,
  ↪ stride_d: 2097152 }
```

(continues on next page)

(continued from previous page)

```

# cgemmm
- { rocblas_function: "rocblas_cgemmm",          transA: "N", transB: "T", M: 1024, N: 2048, K: 512, lda: 1024, ldb: 2048, ldc: 1024, ldd: 1024, cold_iters: 2,
  ↪ iters: 10 }
# cgemmm batched
- { rocblas_function: "rocblas_cgemmm_batched", transA: "N", transB: "T", M: 1024, N: 2048, K: 512, lda: 1024, ldb: 2048, ldc: 1024, ldd: 1024, cold_iters: 2,
  ↪ iters: 10, batch_count: 5 }
# cgemmm strided batched
- { rocblas_function: "rocblas_cgemmm_strided_batched", transA: "N", transB: "T", M: 1024, N: 2048, K: 512, lda: 1024, ldb: 2048, ldc: 1024, ldd: 1024, cold_
  ↪ iters: 2, iters: 10, batch_count: 5, stride_a: 4096, stride_b: 4096, stride_c: 2097152,
  ↪ stride_d: 2097152 }

# zgemmm
- { rocblas_function: "rocblas_zgemmm",          transA: "N", transB: "T", M: 1024, N: 2048, K: 512, lda: 1024, ldb: 2048, ldc: 1024, ldd: 1024, cold_iters: 2,
  ↪ iters: 10 }
# zgemmm batched
- { rocblas_function: "rocblas_zgemmm_batched", transA: "N", transB: "T", M: 1024, N: 2048, K: 512, lda: 1024, ldb: 2048, ldc: 1024, ldd: 1024, cold_iters: 2,
  ↪ iters: 10, batch_count: 5 }
# zgemmm strided batched
- { rocblas_function: "rocblas_zgemmm_strided_batched", transA: "N", transB: "T", M: 1024, N: 2048, K: 512, lda: 1024, ldb: 2048, ldc: 1024, ldd: 1024, cold_
  ↪ iters: 2, iters: 10, batch_count: 5, stride_a: 4096, stride_b: 4096, stride_c: 2097152,
  ↪ stride_d: 2097152 }

# cgemmm
- { rocblas_function: "rocblas_cgemmm",          transA: "N", transB: "C", M: 1024, N: 2048, K: 512, lda: 1024, ldb: 2048, ldc: 1024, ldd: 1024, cold_iters: 2,
  ↪ iters: 10 }
# cgemmm batched
- { rocblas_function: "rocblas_cgemmm_batched", transA: "N", transB: "C", M: 1024, N: 2048, K: 512, lda: 1024, ldb: 2048, ldc: 1024, ldd: 1024, cold_iters: 2,
  ↪ iters: 10, batch_count: 5 }
# cgemmm strided batched
- { rocblas_function: "rocblas_cgemmm_strided_batched", transA: "N", transB: "C", M: 1024, N: 2048, K: 512, lda: 1024, ldb: 2048, ldc: 1024, ldd: 1024, cold_
  ↪ iters: 2, iters: 10, batch_count: 5, stride_a: 4096, stride_b: 4096, stride_c: 2097152,
  ↪ stride_d: 2097152 }

```

HPA cases (gemm_ex)

```
# HHS
- { rocblas_function: "rocblas_gemm_ex", transA: "N", transB: "T", a_type: f16_r, b_
→type: f16_r, c_type: f16_r, d_type: f16_r, compute_type: f32_r, M: 1024, N: 2048,
→ K: 512, lda: 1024, ldb: 2048, ldc: 1024, ldd: 1024, cold_iters: 2, iters:
→10 }

# HHS batched
- { rocblas_function: "rocblas_gemm_ex", transA: "N", transB: "T", a_type: f16_r, b_
→type: f16_r, c_type: f16_r, d_type: f16_r, compute_type: f32_r, M: 1024, N: 2048,
→ K: 512, lda: 1024, ldb: 2048, ldc: 1024, ldd: 1024, cold_iters: 2, iters:
→10, batch_count: 5 }

# HHS strided batched
- { rocblas_function: "rocblas_gemm_ex", transA: "N", transB: "T", a_type: f16_r, b_
→type: f16_r, c_type: f16_r, d_type: f16_r, compute_type: f32_r, M: 1024, N: 2048,
→ K: 512, lda: 1024, ldb: 2048, ldc: 1024, ldd: 1024, cold_iters: 2, iters:
→10, batch_count: 5, stride_a: 4096, stride_b: 4096, stride_c: 2097152, stride_d:
→2097152 }

# HSS
- { rocblas_function: "rocblas_gemm_ex", transA: "N", transB: "T", a_type: f16_r, b_
→type: f16_r, c_type: f16_r, d_type: f16_r, compute_type: f32_r, M: 1024, N: 2048,
→ K: 512, lda: 1024, ldb: 2048, ldc: 1024, ldd: 1024, cold_iters: 2, iters:
→10 }

# HSS batched
- { rocblas_function: "rocblas_gemm_ex", transA: "N", transB: "T", a_type: f16_r, b_
→type: f16_r, c_type: f32_r, d_type: f32_r, compute_type: f32_r, M: 1024, N: 2048,
→ K: 512, lda: 1024, ldb: 2048, ldc: 1024, ldd: 1024, cold_iters: 2, iters:
→10, batch_count: 5 }

# HSS strided batched
- { rocblas_function: "rocblas_gemm_ex", transA: "N", transB: "T", a_type: f16_r, b_
→type: f16_r, c_type: f32_r, d_type: f32_r, compute_type: f32_r, M: 1024, N: 2048,
→ K: 512, lda: 1024, ldb: 2048, ldc: 1024, ldd: 1024, cold_iters: 2, iters:
→10, batch_count: 5, stride_a: 4096, stride_b: 4096, stride_c: 2097152, stride_d:
→2097152 }

# BBS
- { rocblas_function: "rocblas_gemm_ex", transA: "N", transB: "T", a_type: bf16_r, b_
→type: bf16_r, c_type: bf16_r, d_type: bf16_r, compute_type: f32_r, M: 1024, N:
→2048, K: 512, lda: 1024, ldb: 2048, ldc: 1024, ldd: 1024, cold_iters: 2,
→iters: 10 }

# BBS batched
- { rocblas_function: "rocblas_gemm_ex", transA: "N", transB: "T", a_type: bf16_r, b_
→type: bf16_r, c_type: bf16_r, d_type: bf16_r, compute_type: f32_r, M: 1024, N:
→2048, K: 512, lda: 1024, ldb: 2048, ldc: 1024, ldd: 1024, cold_iters: 2,
→iters: 10, batch_count: 5 }

# BBS strided batched
- { rocblas_function: "rocblas_gemm_ex", transA: "N", transB: "T", a_type: bf16_r, b_
→type: bf16_r, c_type: bf16_r, d_type: bf16_r, compute_type: f32_r, M: 1024, N:
→2048, K: 512, lda: 1024, ldb: 2048, ldc: 1024, ldd: 1024, cold_iters: 2,
→iters: 10, batch_count: 5, stride_a: 4096, stride_b: 4096, stride_c: 2097152, stride_
→d: 2097152 }
```

(continues on next page)

(continued from previous page)

```

# BSS
- { rocblas_function: "rocblas_gemm_ex", transA: "N", transB: "T", a_type: bf16_r, b_
  ↪type: bf16_r, c_type: f32_r, d_type: f32_r, compute_type: f32_r, M: 1024, N: 2048, K: 512, lda: 1024, ldb: 2048, ldc: 1024, ldd: 1024, cold_iters: 2,
  ↪iters: 10 }
# BSS batched
- { rocblas_function: "rocblas_gemm_ex", transA: "N", transB: "T", a_type: bf16_r, b_
  ↪type: bf16_r, c_type: f32_r, d_type: f32_r, compute_type: f32_r, M: 1024, N: 2048, K: 512, lda: 1024, ldb: 2048, ldc: 1024, ldd: 1024, cold_iters: 2,
  ↪iters: 10, batch_count: 5 }
# BSS strided batched
- { rocblas_function: "rocblas_gemm_ex", transA: "N", transB: "T", a_type: bf16_r, b_
  ↪type: bf16_r, c_type: f32_r, d_type: f32_r, compute_type: f32_r, M: 1024, N: 2048, K: 512, lda: 1024, ldb: 2048, ldc: 1024, ldd: 1024, cold_iters: 2,
  ↪iters: 10, batch_count: 5, stride_a: 4096, stride_b: 4096, stride_c: 2097152, stride_
  ↪d: 2097152 }

# I8II
- { rocblas_function: "rocblas_gemm_ex", transA: "N", transB: "T", a_type: i8_r, b_type: i8_r, c_type: i32_r, d_type: i32_r, compute_type: i32_r, M: 1024, N: 2048, K: 512, lda: 1024, ldb: 2048, ldc: 1024, ldd: 1024, cold_iters: 2, iters: 10 }
# I8II batched
- { rocblas_function: "rocblas_gemm_ex", transA: "N", transB: "T", a_type: i8_r, b_type: i8_r, c_type: i32_r, d_type: i32_r, compute_type: i32_r, M: 1024, N: 2048, K: 512, lda: 1024, ldb: 2048, ldc: 1024, ldd: 1024, cold_iters: 2, iters: 10, batch_count: 5 }
# I8II strided batched
- { rocblas_function: "rocblas_gemm_ex", transA: "N", transB: "T", a_type: i8_r, b_type: i8_r, c_type: i32_r, d_type: i32_r, compute_type: i32_r, M: 1024, N: 2048, K: 512, lda: 1024, ldb: 2048, ldc: 1024, ldd: 1024, cold_iters: 2, iters: 10, batch_count: 5, stride_a: 4096, stride_b: 4096, stride_c: 2097152, stride_d: 2097152 }

```

For example, the performance of sgemm using rocblas-bench on a vega20 machine returns:

```

./rocblas-bench -f gemm -r f32_r --transposeA N --transposeB N -m 4096 -n 4096 -k 4096 --
  ↪alpha 1 --lda 4096 --ldb 4096 --beta 0 --ldc 4096
transA,transB,M,N,K,alpha,lda,ldb,beta,ldc,rocblas-Gflops,us
N,N,4096,4096,4096,1,4096,4096,0,4096,11941.5,11509.4

```

A useful way of finding the parameters that can be used with `./rocblas-bench -f gemm` is to turn on logging by setting environment variable `ROCBLAS_LAYER=2`. For example if the user runs:

```
ROCBLAS_LAYER=2 ./rocblas-bench -f gemm -i 1 -j 0
```

The above command will log:

```

./rocblas-bench -f gemm -r f32_r --transposeA N --transposeB N -m 128 -n 128 -k 128 --
  ↪alpha 1 --lda 128 --ldb 128 --beta 0 --ldc 128

```

The user can copy and change the above command. For example, to change the datatype to IEEE-64 bit and the size to 2048:

```
./rocbblas-bench -f gemm -r f64_r --transposeA N --transposeB N -m 2048 -n 2048 -k 2048 --  
alpha 1 --lda 2048 --ldb 2048 --beta 0 --ldc 2048
```

Logging affects performance, so only use it to log the command to copy and change, then run the command without logging to measure performance.

Note that rocbblas-bench also has the flag `-v 1` for correctness checks.

6.7.2 rocbblas-gemm-tune

rocbblas-gemm-tune is used to find the best performing GEMM kernel for each of a given set of GEMM problems.

It has a command line interface, which mimics the `--yaml` input used by rocbblas-bench (see above section for details).

To generate the expected `--yaml` input, profile logging can be used, by setting environment variable `ROCBLAS_LAYER=4`.

For more information on rocBLAS logging, see [Logging in rocBLAS](#), in the [API Reference Guide](#).

An example input file:

Expected output (note selected GEMM idx may differ):

Where the far right values (`solution_index`) are the indices of the best performing kernels for those GEMMs in the rocBLAS kernel library. These indices can be directly used in future GEMM calls.

See `rocBLAS/samples/example_user_driven_tuning.cpp` for sample code of directly using kernels via their indices.

If the output is stored in a file, the results can be used to override default kernel selection with the kernels found, by setting the environment variable `ROCBLAS_TENSILE_GEMM_OVERRIDE_PATH=<path>`, where `<path>` points to the stored file.

6.7.3 rocbblas-test

rocbblas-test is used in performing rocBLAS unit tests and it uses Googletest framework.

The tests are in four categories:

- quick
- pre_checkin
- nightly
- known_bug

To run the quick tests:

```
./rocbblas-test --gtest_filter=*quick*
```

The other tests can also be run using the above command by replacing `*quick*` with `*pre_checkin*`, `*nightly*`, and `*known_bug*`.

The pattern for `--gtest_filter` is:

```
--gtest_filter=POSTIVE_PATTERNS[-NEGATIVE_PATTERNS]
```

`gtest_filter` can also be used to run tests for a particular function, and a particular set of input parameters. For example, to run all quick tests for the function `rocbblas_saxpy`:

```
./rocbblas-test --gtest_filter=*quick*axpy*f32_r*
```

The number of lines of output can be reduced with:

```
GTEST_LISTENER=NO_PASS_LINE_IN_LOG ./rocbblas-test --gtest_filter=*quick*
```

rocbblas-test can be driven by tests specified in a yaml file using the `--yaml` argument. As the test categories `pre_checkin` and `nightly` can require hours to run, a short smoke test set is provided in a yaml file. This `rocbblas_smoke.yaml` test set should only require a few minutes to test a few small problem sizes for every function:

```
./rocbblas-test --yaml rocbblas_smoke.yaml
```

- yaml extension for lock step multiple variable scanning

Both `rocbblas-test` and `rocbblas-bench` can use an extension added to scan over multiple variables in lock step implemented by the `Arguments` class. For this purpose set the `Arguments` member variable `scan` to the range to scan over and use `*c_scan_value` to retrieve the values. This can be used to avoid all combinations of yaml variable values that are normally generated. For example, `` - { scan: [32..256..32], M: *c_scan_value, N: *c_scan_value, lda: *c_scan_value } ``

6.7.4 Add New rocBLAS Unit Test

To add new data-driven tests to the rocBLAS Google Test Framework:

I. Create a C++ header file with the name `testing_<function>.hpp` in the `include` subdirectory, with templated functions for a specific rocBLAS routine. Examples:

```
testing_gemm.hpp
testing_gemm_ex.hpp
```

In this `testing_*.hpp` file, create a templated function which returns `void` and accepts a `const Arguments&` parameter. Example:

```
template<typename Ti, typename To, typename Tc>
void testing_gemm_ex(const Arguments& arg)
{
    // ...
}
```

This function is used for yaml file driven argument testing. It will be invoked by the dispatch code for each permutation of the yaml driven parameters. Additionally a template function for bad argument handling tests should be created. Example:

```
template <typename T>
void testing_gemv_bad_arg(const Arguments& arg)
{
    // ...
}
```

These `bad_arg` test function templates should be used to set arguments programmatically where it is simpler than the yaml approach. E.g. to pass `NULL` pointers. It is expected that member variable values in the `Arguments` parameter will not be utilized with the common exception of `arg.fortran` which can drive selection of C and FORTRAN API bad argument tests.

All functions should be generalized with template parameters as much as possible, to avoid copy-and-paste code.

In this function, use the following macros and functions to check results:

HIP_CHECK_ERROR	Verifies that a HIP call returns success
ROCBLAS_CHECK_ERROR	Verifies that a rocBLAS call returns success
EXPECT_ROCBLAS_STATUS	Verifies that a rocBLAS call returns a certain status
unit_check_general	Check that two answers agree (see unit.hpp)
near_check_general	Check that two answers are close (see near.hpp)

In addition, you can use Google Test Macros such as the below, as long as they are guarded by `#ifdef GOOGLE_TEST`:

```
EXPECT_EQ
ASSERT_EQ
EXPECT_TRUE
ASSERT_TRUE
...
```

Note: The `device_vector` template allocates memory on the device. You must check whether converting the `device_vector` to `bool` returns `false`, and if so, report a HIP memory error and then exit the current function. Example:

```
// allocate memory on device
device_vector<T> dx(size_x);
device_vector<T> dy(size_y);
if(!dx || !dy)
{
    CHECK_HIP_ERROR(hipErrorOutOfMemory);
    return;
}
```

The general outline of the function should be:

1. Convert any scalar arguments (e.g., `alpha` and `beta`) to `double`.
2. If the problem size arguments are invalid, use a `safe_size` to allocate arrays, call the rocBLAS routine with the original arguments, and verify that it returns `rocblas_status_invalid_size`. Return.
3. Set up host and device arrays (see `rocblas_vector.hpp` and `rocblas_init.hpp`).
4. Call a CBLAS or other reference implementation on the host arrays.
5. Call rocBLAS using both device pointer mode and host pointer mode, verifying that every rocBLAS call is successful by wrapping it in `ROCBLAS_CHECK_ERROR()`.
6. If `arg.unit_check` is enabled, use `unit_check_general` or `near_check_general` to validate results.
7. (Deprecated) If `arg.norm_check` is enabled, calculate and print out norms.
8. If `arg.timing` is enabled, perform benchmarking (currently under refactoring).

II. Create a C++ file with the name `<function>_gtest.cpp` in the `gtest` subdirectory, where `<function>` is a non-type-specific shorthand for the function(s) being tested. Example:

```
gemm_gtest.cpp
trsm_gtest.cpp
blasl_gtest.cpp
```

In the C++ file, follow these steps:

A. Include the header files related to the tests, as well as `type_dispatch.hpp`. Example:

```
#include "testing_syr.hpp"
#include "type_dispatch.hpp"
```

B. Wrap the body with an anonymous namespace, to minimize namespace collisions:

```
namespace {
```

C. Create a templated class which accepts any number of type parameters followed by one anonymous trailing type parameter defaulted to `void` (to be used with `enable_if`).

Choose the number of type parameters based on how likely in the future that the function will support a mixture of that many different types, e.g. Input type (Ti), Output type (To), Compute type (Tc). If the function will never support more than 1-2 type parameters, then that many can be used. But if the function may be expanded later to support mixed types, then those should be planned for ahead of time and placed in the template parameters.

Unless the number of type parameters is greater than one and is always fixed, then later type parameters should default to earlier ones, so that a subset of type arguments can be used, and so that code which works for functions which take one type parameter may be used for functions which take one or more type parameters. Example:

```
template< typename Ti, typename To = Ti, typename Tc = To, typename = void>
```

Make the primary definition of this class template derive from the `rocblas_test_invalid` class. Example:

```
template <typename T, typename = void>
struct syr_testing : rocblas_test_invalid
{
};
```

D. Create one or more partial specializations of the class template conditionally enabled by the type parameters matching legal combinations of types.

If the first type argument is `void`, then these partial specializations must not apply, so that the default based on `rocblas_test_invalid` can perform the correct behavior when `void` is passed to indicate failure.

In the partial specialization(s), derive from the `rocblas_test_valid` class.

In the partial specialization(s), create a functional operator() which takes a `const Arguments&` parameter and calls templated test functions (usually in `include/testing_*.hpp`) with the specialization's template arguments when the `arg.function` string matches the function name. If `arg.function` does not match any function related to this test, mark it as a test failure. Example:

```
template <typename T>
struct syr_testing<T,
    std::enable_if_t<std::is_same_v<T, float> || std::is_same_v<T, double>
    > : rocblas_test_valid
{
    void operator()(const Arguments& arg)
    {
        if(!strcmp(arg.function, "syr"))
            testing_syr<T>(arg);
        else
            FAIL() << "Internal error: Test called with unknown function: "
                << arg.function;
```

(continues on next page)

(continued from previous page)

```

    }
};

```

- E. If necessary, create a type dispatch function for this function (or group of functions it belongs to) in `include/type_dispatch.hpp`. If possible, use one of the existing dispatch functions, even if it covers a superset of allowable types. The purpose of `type_dispatch.hpp` is to perform runtime type dispatch in a single place, rather than copying it across several test files.

The type dispatch function takes a `template` template parameter of `template<typename...> class` and a function parameter of type `const Arguments&`. It looks at the runtime type values in `Arguments`, and instantiates the template with one or more static type arguments, corresponding to the dynamic runtime type arguments.

It treats the passed template as a functor, passing the `Arguments` argument to a particular instantiation of it.

The combinations of types handled by this “runtime type to template type instantiation mapping” function can be general, because the type combinations which do not apply to a particular test case will have the template argument set to derive from `rocblas_test_invalid`, which will not create any unresolved instantiations. If unresolved instantiation compile or link errors occur, then the `enable_if<>` condition in step D needs to be refined to be `false` for type combinations which do not apply.

The return type of this function needs to be `auto`, picking up the return type of the functor.

If the runtime type combinations do not apply, then this function should return `TEST<void>{}(arg)`, where `TEST` is the template parameter. However, this is less important than step D above in excluding invalid type combinations with `enable_if`, since this only excludes them at run-time, and they need to be excluded by step D at compile-time in order to avoid unresolved references or invalid instantiations. Example:

```

template <template <typename...> class TEST>
auto rocblas_simple_dispatch(const Arguments& arg)
{
    switch(arg.a_type)
    {
        case rocblas_datatype_f16_r: return TEST<rocblas_half>{}(arg);
        case rocblas_datatype_f32_r: return TEST<float>{}(arg);
        case rocblas_datatype_f64_r: return TEST<double>{}(arg);
        case rocblas_datatype_bf16_r: return TEST<rocblas_bfloat16>{}(arg);
        case rocblas_datatype_f16_c: return TEST<rocblas_half_complex>{}(arg);
        case rocblas_datatype_f32_c: return TEST<rocblas_float_complex>{}(arg);
        case rocblas_datatype_f64_c: return TEST<rocblas_double_complex>{}(arg);
        default: return TEST<void>{}(arg);
    }
}

```

- F. Create a (possibly-templated) test implementation class which derives from the `RocBLAS_Test` template class, passing itself to `RocBLAS_Test` (the CRTP pattern) as well as the template class defined above. Example:

```

struct syr : RocBLAS_Test<syr, syr_testing>
{
    // ...
};

```

In this class, implement three static functions:

`static bool type_filter(const Arguments& arg)` returns true if the types described by `*_type` in the `Arguments` structure, match a valid type combination.

This is usually implemented simply by calling the dispatch function in step E, passing it the helper `type_filter_func` template class defined in `RocBLAS_Test`. This functor uses the same runtime type checks as are used to instantiate test functions with particular type arguments, but instead, this returns `true` or `false` depending on whether a function would have been called. It is used to filter out tests whose runtime parameters do not match a valid test.

Since `RocBLAS_Test` is a dependent base class if this test implementation class is templated, you may need to use a fully-qualified name (`A::B`) to resolve `type_filter_func`, and in the last part of this name, the keyword `template` needs to precede `type_filter_func`. The first half of the fully-qualified name can be this class itself, or the full instantiation of `RocBLAS_Test<...>`. Example:

```
static bool type_filter(const Arguments& arg)
{
    return rocblas_blas1_dispatch<
        blas1_test_template::template type_filter_func>(arg);
}
```

`static bool function_filter(const Arguments& arg)` returns `true` if the function name in `Arguments` matches one of the functions handled by this test. Example:

```
// Filter for which functions apply to this suite
static bool function_filter(const Arguments& arg)
{
    return !strcmp(arg.function, "ger") || !strcmp(arg.function, "ger_bad_arg");
}
```

`static std::string name_suffix(const Arguments& arg)` returns a string which will be used as the Google Test name's suffix. It will provide an alphanumeric representation of the test's arguments.

Use the `RocBLAS_TestName` helper class template to create the name. It accepts ostream output (like `std::cout`), and can be automatically converted to `std::string` after all of the text of the name has been streamed to it.

The `RocBLAS_TestName` helper class constructor accepts a string argument which will be included in the test name. It is generally passed the `Arguments` structure's `name` member.

The `RocBLAS_TestName` helper class template should be passed the name of this test implementation class (including any implicit template arguments) as a template argument, so that every instantiation of this test implementation class creates a unique instantiation of `RocBLAS_TestName`. `RocBLAS_TestName` has some static data that needs to be kept local to each test.

`RocBLAS_TestName` converts non-alphanumeric characters into suitable replacements, and disambiguates test names when the same arguments appear more than once.

Since the conversion of the stream into a `std::string` is a destructive one-time operation, the `RocBLAS_TestName` value converted to `std::string` needs to be an rvalue. Example:

```
static std::string name_suffix(const Arguments& arg)
{
    // Okay: rvalue RocBLAS_TestName object streamed to and returned
    return RocBLAS_TestName<sy>() << rocblas_datatype2string(arg.a_type)
        << '_' << (char) std::toupper(arg.uplo) << '_' << arg.N
        << '_' << arg.alpha << '_' << arg.incx << '_' << arg.ldda;
}

static std::string name_suffix(const Arguments& arg)
{
    RocBLAS_TestName<gemm_test_template> name;
```

(continues on next page)

(continued from previous page)

```

name << rocbblas_datatype2string(arg.a_type);
if(GEMM_TYPE == GEMM_EX || GEMM_TYPE == GEMM_STRIDED_BATCHED_EX)
    name << rocbblas_datatype2string(arg.b_type)
        << rocbblas_datatype2string(arg.c_type)
        << rocbblas_datatype2string(arg.d_type)
        << rocbblas_datatype2string(arg.compute_type);
name << '_' << (char) std::toupper(arg.transA)
        << (char) std::toupper(arg.transB) << '_' << arg.M
        << '_' << arg.N << '_' << arg.K << '_' << arg.alpha << '_'
        << arg.lda << '_' << arg.ldb << '_' << arg.beta << '_'
        << arg ldc;
// name is an lvalue: Must use std::move to convert it to rvalue.
// name cannot be used after it's converted to a string, which is
// why it must be "moved" to a string.
return std::move(name);
}

```

- G. Choose a non-type-specific shorthand name for the test, which will be displayed as part of the test name in the Google Tests output (and hence will be stringified). Create a type alias for this name, unless the name is already the name of the class defined in step F, and it is not templated. For example, for a templated class defined in step F, create an alias for one of its instantiations:

```
using gemm = gemm_test_template<gemm_testing, GEMM>;
```

- H. Pass the name created in step G to the TEST_P macro, along with a broad test category name that this test belongs to (so that Google Test filtering can be used to select all tests in a category). The broad test category suffix should be `_tensile` if it requires Tensile.

In the body following this TEST_P macro, call the dispatch function from step E, passing it the class from step C as a template argument, passing the result of `GetParam()` as an `Arguments` structure, and wrapping the call in the `CATCH_SIGNALS_AND_EXCEPTIONS_AS_FAILURES()` macro. Example:

```

TEST_P(gemm, blas3_tensile) { CATCH_SIGNALS_AND_EXCEPTIONS_AS_FAILURES(rocbblas_gemm_
↪dispatch<gemm_testing>(GetParam())); }

```

The `CATCH_SIGNALS_AND_EXCEPTIONS_AS_FAILURES()` macro detects signals such as SIGSEGV and uncaught C++ exceptions returned from rocBLAS C APIs as failures, without terminating the test program.

- I. Call the `INstantiateTestCategories` macro which instantiates the Google Tests across all test categories (quick, pre_checkin, nightly, known_bug), passing it the same test name as in steps G and H. Example:

```
INstantiateTestCategories(gemm);
```

- J. Don't forget to close the anonymous namespace:

```
} // namespace
```

III. Create a `<function>.yaml` file with the same name as the C++ file, just with a `.yaml` extension.

In the YAML file, define tests with combinations of parameters.

The YAML files are organized as files which `include`: each other (an extension to YAML), define anchors for data types and data structures, list of test parameters or subsets thereof, and `Tests` which describe a combination of parameters including category and function.

category must be one of `quick`, `pre_checkin`, `nightly`, or `known_bug`. The category is automatically changed to `known_bug` if the test matches a test in `known_bugs.yaml`.

function must be one of the functions tested for and recognized in steps D-F.

The syntax and idioms of the YAML files is best described by looking at the existing `*_gtest.yaml` files as examples.

IV. Add the YAML file to `roclblas_gtest.yaml`, to be included. Example:

```
include: blas1_gtest.yaml
```

V. Add the YAML file to the list of dependencies for `roclblas_gtest.data` in `CMakeLists.txt`. Example:

```
add_custom_command( OUTPUT "${ROCBLAS_TEST_DATA}"
                    COMMAND ../common/roclblas_gentest.py -I ../include roclblas_gtest.
↪yaml -o "${ROCBLAS_TEST_DATA}"
                    DEPENDS ../common/roclblas_gentest.py roclblas_gtest.yaml ../include/
↪roclblas_common.yaml known_bugs.yaml blas1_gtest.yaml gemm_gtest.yaml gemm_batched_
↪gtest.yaml gemm_strided_batched_gtest.yaml gemv_gtest.yaml symv_gtest.yaml syr_gtest.
↪yaml ger_gtest.yaml trsm_gtest.yaml trtri_gtest.yaml geam_gtest.yaml set_get_vector_
↪gtest.yaml set_get_matrix_gtest.yaml
                    WORKING_DIRECTORY "${CMAKE_CURRENT_SOURCE_DIR}" )
```

VI. Add the `.cpp` file to the list of sources for `roclblas-test` in `CMakeLists.txt`. Example:

```
set(roclblas_test_source
    roclblas_gtest_main.cpp
    ${Tensile_TEST_SRC}
    set_get_pointer_mode_gtest.cpp
    logging_mode_gtest.cpp
    set_get_vector_gtest.cpp
    set_get_matrix_gtest.cpp
    blas1_gtest.cpp
    gemv_gtest.cpp
    ger_gtest.cpp
    syr_gtest.cpp
    symv_gtest.cpp
    geam_gtest.cpp
    trtri_gtest.cpp
)
```

VII. Aim for a function to have tests in each of the categories: `quick`, `pre_checkin`, `nightly`. Aim for tests for each function to have runtime in the table below:

	quick	pre_checkin	nightly
Level 1	2 - 12 sec	20 - 36 sec	70 - 200 sec
Level 2	6 - 36 sec	35 - 100 sec	200 - 650 sec
Level 3	20 sec - 2 min	2 - 6 min	12 - 24 min

Many examples are available in `gtest/*_gtest.{cpp,yaml}`

CONTRIBUTOR'S GUIDE

7.1 Pull-request guidelines

Our code contribution guidelines closely follows the model of [GitHub pull-requests](#). The [rocBLAS repository](#) follows a workflow which dictates a **master** branch where releases are cut, and a **develop** branch which serves as an integration branch for new code. Pull requests should:

- target the **develop** branch for integration
- ensure code builds successfully.
- do not break existing test cases
- new functionality will only be merged with new unit tests
- new unit tests should integrate within the existing googletest framework.
- tests must have good code coverage
- code must also have benchmark tests, and performance must approach the compute bound limit or memory bound limit.

7.2 Coding Guidelines

1. With the [rocBLAS device memory allocation system](#), rocBLAS kernels should not call `hipMalloc()` or `hipFree()` in their own code, but should use the device memory manager.

`hipMalloc()` and `hipFree()` are synchronizing operations which should be avoided as much as possible.

The device memory allocation system provides:

- A `device_malloc` method for temporarily using device memory which has either been allocated before, or which is allocated on demand.
- A method to reuse device memory across rocBLAS calls, without allocating them and deallocating them at every call.
- A method for users to query how much device memory is needed for a particular kernel call, in order for it to perform optimally.
- A method for users to control how much device memory is allocated, or whether to leave it up to rocBLAS to allocate it on demand.

Extra pointers or size arguments for temporary storage should not be added to the end of public APIs, only private internal ones. Instead, implementations of the public APIs should request and obtain device memory

using the rocBLAS device memory manager. rocBLAS kernels in the public API must also detect and respond to *device memory size queries*.

A kernel must allocate all of its device memory upfront, for use during the entirety of the kernel call. It must not allocate and deallocate device memory at different levels of kernel calls. This means that if a lower-level kernel needs device memory, it must be allocated by higher-level routines and passed down to the lower-level routines. When device memory can be shared between two or more operations, the maximum size needed by all them should be reported or allocated.

When allocating memory, it is recommended to use a variable name which implies that this is allocated workspace memory, such as `workspace` or using a `w_` prefix.

Details are in the [Device Memory Allocation](#) design document. Examples of how to use the device memory allocator are in [TRSV](#) and [TRSM](#).

2. Logging, argument error checking and device memory allocation should only occur at the top-level kernel routines. Therefore, if one rocBLAS routine calls another, the lower-level called routine(s) should not perform logging, argument checking, or device memory allocation. This can be accomplished in one of two ways:

A. (Preferred.) Abstract out the computational part of the kernel into a separate template function (usually named `rocblas_<kernel>_template`, and call it from a higher-level template routine (usually named `rocblas_<kernel>_impl`) which does error-checking, device memory allocation, and logging, and which gets called by the C wrappers:

```
template <...>
rocblas_status rocblas_<kernel>_template(..., T* device_memory)
{
    // Performs fast computation
    // No argument error checking
    // No logging
    // No device memory allocation -- any temporary device memory must be passed in_
    ↪ through pointers
    // Can be called by other computational kernels
    // Called by rocblas_<kernel>_impl
    // Private internal API
}

template <...>
rocblas_status rocblas_<kernel>_impl()
{
    // Argument error checking
    // Logging
    // Responding to device memory size queries
    // Device memory allocation (through handle->device_malloc())
    // Temporarily switching to host pointer mode if scalar constants are used
    // Calls rocblas_<kernel>_template()
    // Private internal API
}

extern "C" rocblas_status rocblas_[hsdcz]<kernel>()
{
    // C wrapper
    // Calls rocblas_<kernel>_impl()
    // Public API
}
```

B. Use a `bool` template argument to specify if the kernel template should perform full functionality or not. Pass

device memory pointer(s) which will be used if full functionality is turned off:

```
template <bool full_function, ...>
rocbblas_status rocbblas_<kernel>_template(..., T* device_memory = nullptr)
{
    if(full_function)
    {
        // Argument error checking
        // Logging
        // Responding to device memory size queries
        // Device memory allocation (memory pointer assumed already allocated)
        // otherwise)*
        // Temporarily switching to host pointer mode if scalar constants are used*
        return rocbblas_<kernel>_template<false, ...>(...);
    }
    // Perform fast computation
    // Private internal API
}
```

*Device memory allocation, and temporarily switching pointer mode, might be difficult to enclose in an if statement with the RAII design, so the code might have to use recursion to call the non-fully-functional version of itself after setting these things up. That's why method A above is preferred, but for some huge functions like GEMM, method B might be more practical to implement, since it disrupts existing code less.

3. The pointer mode should be temporarily switched to host mode during kernels which pass constants to other kernels, so that host-side constants of -1.0, 0.0 and 1.0 can be passed to kernels like GEMM, without causing synchronizing host->device memory copies. For example:

```
// Temporarily switch to host pointer mode, saving current pointer mode, restored
// on return
auto saved_pointer_mode = handle->push_pointer_mode(rocbblas_pointer_mode_host);

// Get alpha
T alpha_h;
if(saved_pointer_mode == rocbblas_pointer_mode_host)
    alpha_h = *alpha;
else
    RETURN_IF_HIP_ERROR(hipMemcpy(&alpha_h, alpha, sizeof(T),
    // hipMemcpyDeviceToHost));
```

saved_pointer_mode can be read to get the old pointer mode. If the old pointer mode was host pointer mode, then the host pointer is dereferenced to get the value of alpha. If the old pointer mode was device pointer mode, then the value of alpha is copied from the device to the host.

After the above code switches to host pointer mode, constant values can be passed to GEMM or other kernels by always assuming host mode:

```
static constexpr T negative_one = -1;
static constexpr T zero = 0;
static constexpr T one = 1;

rocbblas_internal_gemm_template( handle, transA, transB, jb, n, jb, alpha, invA,
// BLOCK, B, ldb, &zero, X, m);
```

When saved_pointer_mode is destroyed, the handle's pointer mode returns to the previous pointer mode.

- When tests are added to `rocbblas-test` and `rocbblas-bench`, refer to [this guide](#).

The test framework is templated, and uses SFINAE (substitution failure is not an error) pattern and `std::enable_if<...>` to enable and disable certain types for certain tests.

YAML files are used to describe tests as combinations of arguments. `rocbblas_gentest.py` is used to parse the YAML files and generate tests in the form of a binary file of `Arguments` records.

The `rocbblas-test` and `rocbblas-bench` [type dispatch file](#) is central to all tests. Basically, rather than duplicate:

```
if(type == rocbblas_datatype_f16_r)
    func1<rocbblas_half>(args);
else if(type == rocbblas_datatype_f32_r)
    func<float>(args);
else if(type == rocbblas_datatype_f64_r)
    func<double>(args);
```

etc. everywhere, it's done only in one place, and a `template` argument is passed to specify which action is actually taken. It's fairly abstract, but it is powerful. There are examples of using the type dispatch in `clients/gtest/*_gtest.cpp` and `clients/benchmarks/client.cpp`.

- Code should not be copied-and-pasted, but rather, templates, macros, SFINAE (substitution failure is not an error) pattern and CRTP (curiously recurring template pattern), etc. should be used to factor out differences in similar code.

A code should be made more generalized, rather than copied and modified, unless it is a completely different kernel function, and the old code is just being used as a start.

If a new function is similar to an existing function, then the existing function should be generalized, or the new function and existing function should be refactored and based on a third templated function or class, rather than duplicating code.

- To differentiate between scalars located on either the host or device memory, a special function has been created, called `load_scalar()`. If its argument is a pointer, it is dereferenced on the device. If the argument is a scalar, it is simply copied. This allows single HIP kernels to be written for both device and host memory:

```
template <typename T, typename U>
ROCBLAS_KERNEL void axpy_kernel(rocbblas_int n, U alpha_device_host, const T* x,
    ↪ rocbblas_int incx, T* y, rocbblas_int incy)
{
    auto alpha = load_scalar(alpha_device_host);
    ptrdiff_t tid = blockIdx.x * blockDim.x + threadIdx.x;

    // bound
    if(tid < n)
        y[tid * incy] += alpha * x[tid * incx];
}
```

Here, `alpha_device_host` can either be a pointer to device memory, or a numeric value passed directly to the kernel from the host. The `load_scalar()` function dereferences it if it's a pointer to device memory, and simply returns its argument if it's numerical. The kernel is called from the host in one of two ways depending on the pointer mode:

```
if(handle->pointer_mode == rocbblas_pointer_mode_device)
    hipLaunchKernelGGL(axpy_kernel, blocks, threads, 0, handle->get_stream(), n,
    ↪ alpha, x, incx, y, incy);
else if(*alpha) // alpha is on host
```

(continues on next page)

(continued from previous page)

```
hipLaunchKernelGGL(axpy_kernel, blocks, threads, 0, handle->get_stream(), n,
↳ *alpha, x, incx, y, incy);
```

When the pointer mode indicates alpha is on the host, the alpha pointer is dereferenced on the host and the numeric value it points to is passed to the kernel. When the pointer mode indicates alpha is on the device, the alpha pointer is passed to the kernel and dereferenced by the kernel on the device. This allows a single kernel to handle both cases, eliminating duplicate code.

7. If new arithmetic datatypes (like `rocblas_bfloat16`) are created, then unless they correspond *exactly* to a predefined system type, they should be wrapped into a `struct`, and not simply be a `typedef` to another type of the same size, so that their type is unique and can be differentiated from other types.

Right now `rocblas_half` is `typedefed` to `uint16_t`, which unfortunately prevents `rocblas_half` and `uint16_t` from being differentiable. If `rocblas_half` were simply a `struct` with a `uint16_t` member, then it would be a distinct type.

It is legal to convert a pointer to a standard-layout `class/struct` to a pointer to its first element, and vice-versa, so the C API is unaffected by whether the type is enclosed in a `struct` or not.

8. RAII (resource acquisition is initialization) patterned classes should be used instead of explicit `new/delete`, `hipMalloc/hipFree`, `malloc/free`, etc. RAII classes are automatically exception-safe because their destructor gets called during unwinding. They only have to be declared once to construct them, and they are automatically destroyed when they go out of scope. This is better than having to match `new/delete` `malloc/free` calls in the code, especially when exceptions or early returns are possible.

Even if an operation does not allocate and free memory, if it represents a change in state which must be undone when a function returns, then it belongs in an RAII class. For example, `handle->push_pointer_mode()` creates an RAII object which saves the pointer mode on construction, and restores it on destruction.

9. When writing function templates, place any non-type parameters before type parameters, i.e., leave the type parameters at the end. For example:

```
template <rocblas_int NB, typename T> // T is at end
static rocblas_status rocblas_trtri_batched_template(rocblas_handle handle,
                                                    rocblas_fill uplo,
                                                    rocblas_diagonal diag,
                                                    rocblas_int n,
                                                    const T* A,
                                                    rocblas_int lda,
                                                    rocblas_int bsa,
                                                    T* invA,
                                                    rocblas_int ldinvA,
                                                    rocblas_int bsinvA,
                                                    rocblas_int batch_count,
                                                    T* C_tmp)
{
    if(!n || !batch_count)
        return rocblas_status_success;

    if(n <= NB)
        return rocblas_trtri_small_batched<NB>( // T is automatically deduced
            handle, uplo, diag, n, A, lda, bsa, invA, ldinvA, bsinvA, batch_count);
    else
        return rocblas_trtri_large_batched<NB>( // T is automatically deduced
            handle, uplo, diag, n, A, lda, bsa, invA, ldinvA, bsinvA, batch_count,

```

(continues on next page)

(continued from previous page)

```
↪ C_tmp);
}
```

The reason for this, is that the type template arguments can be automatically deduced from the actual function arguments, so that you don't have to pass the types in calls to the function, as shown in the example above when calling `rocbblas_trtri_small_batched` and `rocbblas_trtri_large_batched`. They have a `typename T` parameter too, but it can be automatically deduced, so it doesn't need to be explicitly passed.

10. When writing functions like the above which are heavily dependent on block sizes, especially if they are in header files included by other files, template parameters for block sizes are strongly preferred to `#define` macros or `constexpr` variables. For `.cpp` files which are not included in other files, a `static constexpr` variable can be used. **Macros should never be used for constants.**

Note: For constants inside of functions, `static constexpr` is preferred to just `constexpr`, so that the variables do not need to be initialized at runtime.

Note: C++14 variable templates can sometimes be used to provide constants. For example:

```
template <typename T>
static constexpr T negative_one = -1;

template <typename T>
static constexpr T zero = 0;

template <typename T>
static constexpr T one = 1;
```

11. static duration variables which aren't constants should usually be made function-local `static` variables, rather than namespace or class static variables. This is to avoid the static initialization order fiasco. For example:

```
static auto& get_table()
{
    // Placed inside function to avoid dependency on initialization order
    static std::unordered_map<std::string, size_t>* table = test_cleanup::allocate(&
↪table);
    return *table;
}
```

This is sometimes called the *singleton* pattern. A static variable is made local to a function rather than a namespace or class, and it gets initialized the first time the function is called. A reference to the static variable is returned from the function, and the function is used everywhere access to the variable is needed. In the case of multithreaded programs, the C++11 and later standards guarantee that there won't be any race conditions. It is preferred to initialize function-local static variables than it is to explicitly call `std::call_once`. For example:

```
void my_func()
{
    static int dummy = (func_to_call_once(), 0);
}
```

This is much simpler and faster than explicitly calling `std::call_once`, since the compiler has special ways of optimizing static initialization. The first time `my_func()` is called, it will call `func_to_call_once()` once in a thread-safe way. After that, there is almost no overhead in later calls to `my_func()`.

12. Functions are preferred to macros. Functions or functors inside of class / struct templates can be used when

partial template specializations are needed.

When C preprocessor macros are needed (such as if they contain a `return` statement to return from the calling function), if the macro's definition contains more than one simple expression, then it should be wrapped in a `{ } while(0)`, without a terminating semicolon. This is to allow them to be used inside `if` statements. For example:

```
#define RETURN_ZERO_DEVICE_MEMORY_SIZE_IF_QUERIED(h) \
    do \
    { \
        if((h)->is_device_memory_size_query()) \
            return rocblas_status_size_unchanged; \
    } while(0)
```

The `do { } while(0)` allows the macro expansion to be a single statement which can be terminated with a semicolon, and which can be used anywhere a regular function call can be used.

13. For most template functions which are used in other compilation units, it is preferred that they be put in header files, rather than `.cpp` files, because putting them in `.cpp` files requires explicit instantiation of them for all possible arguments, and there are less opportunities for inlining and interprocedural optimization.

The C++ standard explicitly says that unused templates can be omitted from the output, so including unused templates in a header file does not increase the size of the program, since only the used ones are in the final output.

For template functions which are only used in one `.cpp` file, they can be placed in the `.cpp` file.

Templates, like inline functions, are granted an exception to the one definition rule (ODR) as long as the sequence of tokens in each compilation unit is identical.

14. Functions and namespace-scope variables which are not a part of the public interface of rocBLAS, should either be marked static, be placed in an unnamed namespace, or be placed in `namespace rocblas`. For example:

```
namespace
{
    // Private internal implementation
} // namespace

extern "C"
{
    // Public C interfaces
} // extern "C"
```

However, unnamed namespaces should not be used in header files. If it is absolutely necessary to mark a function or variable as private to a compilation unit but defined in a header file, it should be declared `static`, `constexpr` and/or `inline` (`constexpr` implies `static` for non-template variables and `inline` for functions).

Even though rocBLAS goes into a shared library which exports a limited number of symbols, this is still a good idea, to decrease the chances of name collisions *inside* of rocBLAS.

15. `std::string` should only be used for strings which can grow, or which must be dynamically allocated as read-write strings. For simple static strings, strings returned from functions like `getenv()`, or strings which are initialized once and then used read-only, `const char*` should be used to refer to the string or pass it as an argument.

`std::string` involves dynamic memory allocation and copying of temporaries, which can be slow. `std::string_view` is supposed to help alleviate that, which became available in C++17. `const char*` can be used for read-only views of strings, in the interest of efficiency.

16. For code brevity and readability, when converting to *numeric* types, uniform initialization or function-style casts are preferred to `static_cast<>()` or C-style casts. For example, `T{x}` or `T(x)` is preferred to `static_cast<T>(x)` or `(T)x`. `T{x}` differs from `T(x)` in that narrowing conversions, which reduce the precision of an integer or floating-point, are not allowed.

When writing general containers or templates which can accept *arbitrary* types as parameters, not just *numeric* types, then the specific cast (`static_cast`, `const_cast`, `reinterpret_cast`) should be used, to avoid surprises.

But when converting to *numeric* types, which have very well-understood behavior and are *side-effect free*, `type{x}` or `type(x)` are more compact and clearer than `static_cast<type>(x)`. For pointers, C-style casts are okay, such as `(T*)A`.

17. For BLAS2 functions and BLAS1 functions with two vectors, the `incx` and/or `incy` arguments can be negative, which means the vector is treated backwards from the end. A simple trick to handle this, is to adjust the pointer to the end of the vector if the increment is negative, as in:

```
if(incx < 0)
    x -= ptrdiff_t(incx) * (n - 1);
if(incy < 0)
    y -= ptrdiff_t(incy) * (n - 1);
```

After that adjustment, the code does not need to treat negative increments any differently than positive ones.

Note: Some blocked matrix-vector algorithms which call other BLAS kernels may not work if this simple transformation is used; see [TRSV](#) for an example, and how it's handled there.

18. For reduction operations, the file `reduction.h` has been created to systematize reductions and perform their device kernels in one place. This works for `amax`, `amin`, `asum`, `nrm2`, and (partially) `dot` and `gemv`. `rocblas_reduction_kernel` is a generalized kernel which takes 3 *functors* as template arguments:

- One to *fetch* values (such as fetching a complex value and taking the sum of the squares of its real and imaginary parts before reducing it)
- One to *reduce* values (such as to compute a sum or maximum)
- One to *finalize* the reduction (such as taking the square root of a sum of squares)

There is a `default_value()` function which returns the default value for a reduction. The default value is the value of the reduction when the size is 0, and reducing a value with the `default_value()` does not change the value of the reduction.

19. When type punning is needed, `union` should be used instead of pointer-casting, which violates *strict aliasing*. For example:

```
// zero extend lower 16 bits of bfloat16 to convert to IEEE float
explicit __host__ __device__ operator float() const
{
    union
    {
        uint32_t int32;
        float    fp32;
    } u = {uint32_t(data) << 16};
    return u.fp32; // Legal in C, nonstandard extension in C++
}
```

This violates the strict aliasing rule of C and C++:

```
// zero extend lower 16 bits of bfloat16 to convert to IEEE float
explicit __host__ __device__ operator float() const
{
    uint32_t int32 = uint32_t(data) << 16;
    return *(float *) &int32; // Violates strict aliasing rule in both C and C++
}
```

The only 100% standard C++ way to do it, is to use `memcpy()`, but this should not be required as long as GCC or Clang are used:

```
// zero extend lower 16 bits of bfloat16 to convert to IEEE float
explicit __host__ __device__ operator float() const
{
    uint32_t int32 = uint32_t(data) << 16;
    float fp32;
    static_assert(sizeof(int32) == sizeof(fp32), "Different sizes");
    memcpy(&fp32, &int32, sizeof(fp32));
    return fp32;
}
```

20. `<type_traits>` classes which return Boolean values can be converted to `bool` in Boolean contexts. Hence many traits can be tested by simply creating an instance of them with `{}`. However, for `type_traits` accessors such as `::value` or `::type`, these can be replaced by suffixes added in C++17 such as `is_same_v` and `enable_if_t`:

```
template<typename T, typename = typename std::enable_if_t<std::is_same_v<T, float>>_
↳ ||
                                                    std::is_same_v<T, double>>
↳ >
void function(T x)
{
}
```

For other traits created with the `{}` syntax the resulting temporary objects can be explicitly converted to `bool`, which is what occurs when an object appears in a conditional expression (`if`, `while`, `for`, `&&`, `||`, `!`, `?` `:`, etc.).

21. `roclblas_cout` and `roclblas_cerr` should be used instead of `std::cout`, `std::cerr`, `stdout` or `stderr`, and `roclblas_internal_ostream` should be used instead of `std::ostream`, `std::ofstream` or `std::ostringstream`.

In `roclblas-bench` and `roclblas-test`, `std::cout`, `std::cerr`, `printf`, `fprintf`, `stdout`, `stderr`, `puts()`, `fputs()`, and other symbols are “poisoned”, to remind you to use `roclblas_cout`, `roclblas_cerr`, and `roclblas_internal_ostream` instead.

`roclblas_cout` and `roclblas_cerr` are instances of `roclblas_internal_ostream` which output to standard output and standard error, but in a way that prevents interlacing of different threads’ output.

`roclblas_internal_ostream` provides standardized thread-safe formatted output for rocBLAS datatypes. It can be constructed in 3 ways: - By default, in which case it behaves like a `std::ostringstream` - With a file descriptor number, in which case the file descriptor is `dup()`’ed and the same file it points to is outputted to - With a string, in which case a new file is opened for writing, with file creation, truncation and appending enabled (```O_WRONLY | O_CREAT | O_TRUNC | O_APPEND | O_CLOEXEC`)

`std::endl` or `std::flush` should be used at the end of an output sequence when an atomic flush of the output is needed (atomic meaning that multiple threads can be writing to the same file, but that their flushes will be atomic). Until then, the output will accumulate in the `roclblas_internal_ostream` and will not be flushed

until either `roclblas_internal_ostream::flush()` is called, `std::endl` or `std::flush` is outputted, or the `roclblas_internal_ostream` is destroyed.

The `roclblas_internal_ostream::yaml_on` and `roclblas_internal_ostream::yaml_off` IO modifiers enable or disable YAML formatting, for when outputting arbitrary types as YAML source code. For example, to output a key: value pair as YAML source code, you would use:

```
os << key << ": " << roclblas_internal_ostream::yaml_on << value << roclblas_internal_
    ↪ ostream::yaml_off;
```

The key is outputted normally as a bare string, but the value uses YAML metacharacters and lexical syntax to output the value, so that when it's read in as YAML, it has the type and value of value.

22. C++ templates, including variadic templates, are preferred to macros or runtime interpreting of values, although it is understood that sometimes macros are necessary.

For example, when creating a class which models zero or more rocBLAS kernel arguments, it is preferable to use:

```
template<roclblas_argument... Args>
class ArgumentModel
{
public:
    void func()
    {
        for(auto arg: { Args... })
        {
            //do something with argument arg
        }
    }
};
ArgumentModel<e_A, e_B>{}.func();
```

instead of:

```
class ArgumentModel
{
    std::vector<roclblas_argument> args;
public:
    ArgumentModel(const std::vector<roclblas_argument>& args): args(args)
    {
    }

    void func()
    {
        for(auto arg: args)
        {
            //do something with argument arg
        }
    }
};
ArgumentModel model({e_A, e_B});
model.func();
```

The former denotes the rocBLAS arguments as a list which is passed as a variadic template argument, and whose properties are known and can be optimized at compile-time, and which can be passed on as arguments to other

templates, while the latter requires creating a dynamically-allocated runtime object which must be interpreted at runtime, such as by using `switch` statements on the arguments. The `switch` statement will need to list out and handle every possible argument, while the template solution simply passes the argument as another template argument, and hence can be resolved at compile-time.

23. Automatically-generated files should always go into `build/` directories, and should not go into source directories (even if marked `.gitignore`). The CMake philosophy is such that you can create any `build/` directory, run `cmake` from there, and then have a self-contained build environment which will not touch any files outside of it.
24. The `library/include` subdirectory of rocBLAS, to be distinguished from the `library/src/include` subdirectory, shall consist only of C-compatible header files for public rocBLAS APIs. It should not include internal APIs, even if they are used in other projects, e.g., rocSOLVER, and the headers must be compilable with a C compiler, and must use `.h` extensions.
25. Macro parameters should only be evaluated once when practical, and should be parenthesized if there is a chance of ambiguous precedence. They should be stored in a local temporary variable if needed more than once.

Macros which expand to code with local variables, should use double-underscore suffixes in the local variable names, to prevent their conflict with variables passed in macro parameters. However, if they are in a completely separate block scope than the macro parameter is expanded in, or if they are only passed to another macro/function, then they do not need to use trailing underscores.

```
#define CHECK_DEVICE_ALLOCATION(ERROR) \
do \
{ \
    /* Use error__ in case ERROR contains "error" */ \
    hipError_t error__ = (ERROR); \
    if(error__ != hipSuccess) \
    { \
        if(error__ == hipErrorOutOfMemory) \
            SUCCEED() << LIMITED_MEMORY_STRING; \
        else \
            FAIL() << hipGetErrorString(error__); \
        return; \
    } \
} while(0)
```

The `ERROR` macro parameter is evaluated only once, and is stored in the temporary variable `error__`, for use multiple times later.

The `ERROR` macro parameter is parenthesized when initializing `error__`, to avoid ambiguous precedence, such as if `ERROR` contains a comma expression.

The `error__` variable name is used, to prevent it from conflicting with variables passed in the `ERROR` macro parameter, such as `error`.

26. Do not use variable-length arrays (VLA), which allocate on the stack, for arrays of unknown size.

```
Ti* hostA[batch_count];
Ti* hostB[batch_count];
To* hostC[batch_count];
To* hostD[batch_count];

func(hostA, hostB, hostC, hostD);
```

Instead, allocate on the heap, using smart pointers to avoid memory leaks:

```
auto hostA = std::make_unique<Ti*>(batch_count);
auto hostB = std::make_unique<Ti*>(batch_count);
auto hostC = std::make_unique<To*>(batch_count);
auto hostD = std::make_unique<To*>(batch_count);

func(&hostA[0], &hostB[0], &hostC[0], &hostD[0]);
```

27. Do not define unnamed (anonymous) namespaces in header files (for explanation see DCL59-CPP)

If the reason for using an unnamed namespace in a header file is to prevent multiple definitions, keep in mind that the following are allowed to be defined in multiple compilation units, such as if they all come from the same header file, as long as they are defined with identical token sequences in each compilation unit:

- classes
- typedefs or type aliases
- enums
- template functions
- inline functions
- constexpr functions (implies inline)
- inline or constexpr variables or variable template`s (only for C++17 or later, although some C++14 compilers treat ``constexpr variables as inline)

If functions defined in header files are declared `template`, then multiple instantiations with the same `template` arguments are automatically merged, something which cannot happen if the `template` functions are declared `static`, or appear in unnamed namespaces, in which case the instantiations are local to each compilation unit, and are not combined.

If a function defined in a header file at namespace scope (outside of a class) contains `static_local` variables which are expected to be singletons holding state throughout the entire library, then the function cannot be marked `static` or be part of an unnamed namespace, because then each compilation unit will have its own separate copy of that function and its local `static` variables. (static member functions of classes always have external linkage, and it is okay to define static class member functions in-place inside of header files, because all in-place static member function definitions, including their static local variables, will be automatically merged.)

Guidelines:

- Do not use unnamed namespaces inside of header files.
 - Use either `template` or `inline` (or both) for functions defined outside of classes in header files.
 - Do not declare namespace-scope (not class-scope) functions `static` inside of header files unless there is a very good reason, that the function does not have any non-const `static` local variables, and that it is acceptable that each compilation unit will have its own independent definition of the function and its `static` local variables. (static class member functions defined in header files are okay.)
 - Use `static` for `constexpr template` variables until C++17, after which `constexpr` variables become `inline` variables, and thus can be defined in multiple compilation units. It is okay if the `constexpr` variables remain `static` in C++17; it just means there might be a little bit of redundancy between compilation units.

7.2.1 Format

C and C++ code is formatted using `clang-format`. To run `clang-format` use the version in the `/opt/rocm/llvm/bin` directory. Please do not use your system's built-in `clang-format`, as this may be an older version that will result in different results.

To format a file, use:

```
/opt/rocm/llvm/bin/clang-format -style=file -i <path-to-source-file>
```

To format all files, run the following script in rocBLAS directory:

```
#!/bin/bash
git ls-files -z *.cc *.cpp *.h *.hpp *.cl *.h.in *.hpp.in *.cpp.in | xargs -0 /opt/rocm/
↳ llvm/bin/clang-format -style=file -i
```

Also, githooks can be installed to format the code per-commit:

```
./.githooks/install
```

7.3 Static Code Analysis

`cppcheck` is an open-source static analysis tool. This project uses this tool for performing static code analysis.

Users can use the following command to run `cppcheck` locally to generate the report for all files.

```
$ cd rocBLAS-internal
$ cppcheck --enable=all --inconclusive --library=googletest --inline-suppr -i./build --
↳ suppressions-list=./CppCheckSuppressions.txt --template="{file}:{line}: {severity}:
↳ {id} :{message}" . 2> cppcheck_report.txt
```

Also, githooks can be installed to perform static analysis on new/modified files using pre-commit:

```
./.githooks/install
```

For more information on the command line options, refer to the `cppcheck` manual on the web.

ACKNOWLEDGEMENT

AMD would like to thank the following for their code contributions to rocBLAS:

- Ahmad Abdelfattah of the University of Tennessee and King Abdullah University of Science and Technology , for portions of trmm and gemv
- Mark Gates of the University of Tennessee, for portions of symv
- Jonathan Hogg of STFC Rutherford Appleton Laboratory, for portions of trsv

BIBLIOGRAPHY

- [Level1] C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh, Basic Linear Algebra Subprograms for FORTRAN usage, *ACM Trans. Math. Soft.*, 5 (1979), pp. 308–323.
- [Level2] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, An extended set of FORTRAN Basic Linear Algebra Subprograms, *ACM Trans. Math. Soft.*, 14 (1988), pp. 1–17
- [Level3] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, Algorithm 656: An extended set of FORTRAN Basic Linear Algebra Subprograms, *ACM Trans. Math. Soft.*, 14 (1988), pp. 18–32

R

rocnblas_atomics_mode (C++ *enum*), 26
 rocnblas_atomics_mode::rocblas_atomics_allowed (C++ *enumerator*), 26
 rocnblas_atomics_mode::rocblas_atomics_not_allowed (C++ *enumerator*), 26
 rocnblas_axpy_batched_ex (C++ *function*), 207
 rocnblas_axpy_ex (C++ *function*), 206
 rocnblas_axpy_strided_batched_ex (C++ *function*), 208
 rocnblas_bfdot (C++ *function*), 42
 rocnblas_bfdot_batched (C++ *function*), 43
 rocnblas_bfdot_strided_batched (C++ *function*), 45
 rocnblas_bfloat16 (C++ *struct*), 22
 rocnblas_caxpy (C++ *function*), 38
 rocnblas_caxpy_batched (C++ *function*), 39
 rocnblas_caxpy_strided_batched (C++ *function*), 39
 rocnblas_ccopy (C++ *function*), 40
 rocnblas_ccopy_batched (C++ *function*), 41
 rocnblas_ccopy_strided_batched (C++ *function*), 41
 rocnblas_cdgmm (C++ *function*), 240
 rocnblas_cdgmm_batched (C++ *function*), 241
 rocnblas_cdgmm_strided_batched (C++ *function*), 242
 rocnblas_cdotc (C++ *function*), 42
 rocnblas_cdotc_batched (C++ *function*), 43
 rocnblas_cdotc_strided_batched (C++ *function*), 45
 rocnblas_cdotu (C++ *function*), 42
 rocnblas_cdotu_batched (C++ *function*), 43
 rocnblas_cdotu_strided_batched (C++ *function*), 45
 rocnblas_cgbmv (C++ *function*), 62
 rocnblas_cgbmv_batched (C++ *function*), 64
 rocnblas_cgbmv_strided_batched (C++ *function*), 65
 rocnblas_cgeam (C++ *function*), 237
 rocnblas_cgeam_batched (C++ *function*), 238
 rocnblas_cgeam_strided_batched (C++ *function*), 239
 rocnblas_cgemm (C++ *function*), 147
 rocnblas_cgemm_batched (C++ *function*), 148
 rocnblas_cgemm_strided_batched (C++ *function*), 150
 rocnblas_cgemv (C++ *function*), 67
 rocnblas_cgemv_batched (C++ *function*), 68
 rocnblas_cgemv_strided_batched (C++ *function*), 69
 rocnblas_cgerc (C++ *function*), 70
 rocnblas_cgerc_batched (C++ *function*), 71
 rocnblas_cgerc_strided_batched (C++ *function*), 73
 rocnblas_cgeru (C++ *function*), 70
 rocnblas_cgeru_batched (C++ *function*), 71
 rocnblas_cgeru_strided_batched (C++ *function*), 72
 rocnblas_chbmv (C++ *function*), 122
 rocnblas_chbmv_batched (C++ *function*), 123
 rocnblas_chbmv_strided_batched (C++ *function*), 125
 rocnblas_chemm (C++ *function*), 189
 rocnblas_chemm_batched (C++ *function*), 190
 rocnblas_chemm_strided_batched (C++ *function*), 191
 rocnblas_chemv (C++ *function*), 119
 rocnblas_chemv_batched (C++ *function*), 120
 rocnblas_chemv_strided_batched (C++ *function*), 121
 rocnblas_cher (C++ *function*), 132
 rocnblas_cher2 (C++ *function*), 135
 rocnblas_cher2_batched (C++ *function*), 135
 rocnblas_cher2_strided_batched (C++ *function*), 137
 rocnblas_cher2k (C++ *function*), 196
 rocnblas_cher2k_batched (C++ *function*), 197
 rocnblas_cher2k_strided_batched (C++ *function*), 198
 rocnblas_cher_batched (C++ *function*), 132
 rocnblas_cher_strided_batched (C++ *function*), 133
 rocnblas_cherk (C++ *function*), 193
 rocnblas_cherk_batched (C++ *function*), 194
 rocnblas_cherk_strided_batched (C++ *function*), 195
 rocnblas_cherkx (C++ *function*), 200
 rocnblas_cherkx_batched (C++ *function*), 201
 rocnblas_cherkx_strided_batched (C++ *function*), 202
 rocnblas_chpmv (C++ *function*), 127
 rocnblas_chpmv_batched (C++ *function*), 128
 rocnblas_chpmv_strided_batched (C++ *function*),

130
rocblas_chpr (C++ function), 138
rocblas_chpr2 (C++ function), 142
rocblas_chpr2_batched (C++ function), 143
rocblas_chpr2_strided_batched (C++ function), 145
rocblas_chpr_batched (C++ function), 139
rocblas_chpr_strided_batched (C++ function), 140
rocblas_create_handle (C++ function), 28
rocblas_crot (C++ function), 48
rocblas_crot_batched (C++ function), 48
rocblas_crot_strided_batched (C++ function), 49
rocblas_crotg (C++ function), 50
rocblas_crotg_batched (C++ function), 51
rocblas_crotg_strided_batched (C++ function), 52
rocblas_cscal (C++ function), 58
rocblas_cscal_batched (C++ function), 59
rocblas_cscal_strided_batched (C++ function), 59
rocblas_cspr (C++ function), 79
rocblas_cspr_batched (C++ function), 81
rocblas_cspr_strided_batched (C++ function), 82
rocblas_csrot (C++ function), 48
rocblas_csrot_batched (C++ function), 48
rocblas_csrot_strided_batched (C++ function), 49
rocblas_csscal (C++ function), 58
rocblas_csscal_batched (C++ function), 59
rocblas_csscal_strided_batched (C++ function), 59
rocblas_cswap (C++ function), 60
rocblas_cswap_batched (C++ function), 61
rocblas_cswap_strided_batched (C++ function), 61
rocblas_csymm (C++ function), 151
rocblas_csymm_batched (C++ function), 152
rocblas_csymm_strided_batched (C++ function), 154
rocblas_csymv (C++ function), 88
rocblas_csymv_batched (C++ function), 89
rocblas_csymv_strided_batched (C++ function), 90
rocblas_csyrr (C++ function), 91
rocblas_csyrr2 (C++ function), 94
rocblas_csyrr2_batched (C++ function), 95
rocblas_csyrr2_strided_batched (C++ function), 96
rocblas_csyrr2k (C++ function), 159
rocblas_csyrr2k_batched (C++ function), 161
rocblas_csyrr2k_strided_batched (C++ function), 162
rocblas_csyrr_batched (C++ function), 92
rocblas_csyrr_strided_batched (C++ function), 93
rocblas_csyrrk (C++ function), 155
rocblas_csyrrk_batched (C++ function), 156
rocblas_csyrrk_strided_batched (C++ function), 158
rocblas_csyrrkx (C++ function), 164
rocblas_csyrrkx_batched (C++ function), 166
rocblas_csyrrkx_strided_batched (C++ function), 167
rocblas_ctbmv (C++ function), 97
rocblas_ctbmv_batched (C++ function), 99
rocblas_ctbmv_strided_batched (C++ function), 100
rocblas_ctbsv (C++ function), 103
rocblas_ctbsv_batched (C++ function), 104
rocblas_ctbsv_strided_batched (C++ function), 105
rocblas_ctpmv (C++ function), 107
rocblas_ctpmv_batched (C++ function), 108
rocblas_ctpmv_strided_batched (C++ function), 109
rocblas_ctpsv (C++ function), 110
rocblas_ctpsv_batched (C++ function), 111
rocblas_ctpsv_strided_batched (C++ function), 112
rocblas_ctrmm (C++ function), 169
rocblas_ctrmm_batched (C++ function), 174
rocblas_ctrmm_strided_batched (C++ function), 178
rocblas_ctrmv (C++ function), 113
rocblas_ctrmv_batched (C++ function), 114
rocblas_ctrmv_strided_batched (C++ function), 115
rocblas_ctrsm (C++ function), 184
rocblas_ctrsm_batched (C++ function), 186
rocblas_ctrsm_strided_batched (C++ function), 187
rocblas_ctrsv (C++ function), 116
rocblas_ctrsv_batched (C++ function), 117
rocblas_ctrsv_strided_batched (C++ function), 118
rocblas_dasum (C++ function), 36
rocblas_dasum_batched (C++ function), 37
rocblas_dasum_strided_batched (C++ function), 37
rocblas_datatype (C++ enum), 25
rocblas_datatype::rocblas_datatype_bf16_c (C++ enumerator), 26
rocblas_datatype::rocblas_datatype_bf16_r (C++ enumerator), 26
rocblas_datatype::rocblas_datatype_f16_c (C++ enumerator), 25
rocblas_datatype::rocblas_datatype_f16_r (C++ enumerator), 25
rocblas_datatype::rocblas_datatype_f32_c (C++ enumerator), 25
rocblas_datatype::rocblas_datatype_f32_r (C++ enumerator), 25
rocblas_datatype::rocblas_datatype_f64_c (C++ enumerator), 25
rocblas_datatype::rocblas_datatype_f64_r (C++ enumerator), 25

roclblas_datatype::roclblas_datatype_i32_c
 (C++ *enumerator*), 26
 roclblas_datatype::roclblas_datatype_i32_r
 (C++ *enumerator*), 25
 roclblas_datatype::roclblas_datatype_i8_c
 (C++ *enumerator*), 25
 roclblas_datatype::roclblas_datatype_i8_r
 (C++ *enumerator*), 25
 roclblas_datatype::roclblas_datatype_invalid
 (C++ *enumerator*), 26
 roclblas_datatype::roclblas_datatype_u32_c
 (C++ *enumerator*), 26
 roclblas_datatype::roclblas_datatype_u32_r
 (C++ *enumerator*), 25
 roclblas_datatype::roclblas_datatype_u8_c
 (C++ *enumerator*), 25
 roclblas_datatype::roclblas_datatype_u8_r
 (C++ *enumerator*), 25
 roclblas_daxpy (C++ *function*), 38
 roclblas_daxpy_batched (C++ *function*), 39
 roclblas_daxpy_strided_batched (C++ *function*), 39
 roclblas_dcopy (C++ *function*), 40
 roclblas_dcopy_batched (C++ *function*), 41
 roclblas_dcopy_strided_batched (C++ *function*), 41
 roclblas_ddgmm (C++ *function*), 240
 roclblas_ddgmm_batched (C++ *function*), 241
 roclblas_ddgmm_strided_batched (C++ *function*),
 242
 roclblas_ddot (C++ *function*), 42
 roclblas_ddot_batched (C++ *function*), 43
 roclblas_ddot_strided_batched (C++ *function*), 44
 roclblas_destroy_handle (C++ *function*), 28
 roclblas_dgbmv (C++ *function*), 62
 roclblas_dgbmv_batched (C++ *function*), 64
 roclblas_dgbmv_strided_batched (C++ *function*), 65
 roclblas_dgeam (C++ *function*), 237
 roclblas_dgeam_batched (C++ *function*), 238
 roclblas_dgeam_strided_batched (C++ *function*),
 239
 roclblas_dgemm (C++ *function*), 147
 roclblas_dgemm_batched (C++ *function*), 148
 roclblas_dgemm_strided_batched (C++ *function*),
 149
 roclblas_dgemv (C++ *function*), 67
 roclblas_dgemv_batched (C++ *function*), 68
 roclblas_dgemv_strided_batched (C++ *function*), 69
 roclblas_dger (C++ *function*), 70
 roclblas_dger_batched (C++ *function*), 71
 roclblas_dger_strided_batched (C++ *function*), 72
 roclblas_diagonal (C++ *enum*), 23
 roclblas_diagonal::roclblas_diagonal_non_unit
 (C++ *enumerator*), 23
 roclblas_diagonal::roclblas_diagonal_unit
 (C++ *enumerator*), 23
 roclblas_dnrm2 (C++ *function*), 46
 roclblas_dnrm2_batched (C++ *function*), 46
 roclblas_dnrm2_strided_batched (C++ *function*), 47
 roclblas_dot_batched_ex (C++ *function*), 210
 roclblas_dot_ex (C++ *function*), 209
 roclblas_dot_strided_batched_ex (C++ *function*),
 211
 roclblas_dotc_batched_ex (C++ *function*), 213
 roclblas_dotc_ex (C++ *function*), 212
 roclblas_dotc_strided_batched_ex (C++ *function*),
 214
 roclblas_double_complex (C++ *struct*), 22
 roclblas_drot (C++ *function*), 48
 roclblas_drot_batched (C++ *function*), 48
 roclblas_drot_strided_batched (C++ *function*), 49
 roclblas_drotg (C++ *function*), 50
 roclblas_drotg_batched (C++ *function*), 51
 roclblas_drotg_strided_batched (C++ *function*), 51
 roclblas_drotm (C++ *function*), 52
 roclblas_drotm_batched (C++ *function*), 53
 roclblas_drotm_strided_batched (C++ *function*), 54
 roclblas_drotmg (C++ *function*), 55
 roclblas_drotmg_batched (C++ *function*), 56
 roclblas_drotmg_strided_batched (C++ *function*),
 57
 roclblas_dsblmv (C++ *function*), 74
 roclblas_dsblmv_batched (C++ *function*), 75
 roclblas_dsblmv_strided_batched (C++ *function*), 75
 roclblas_dscal (C++ *function*), 58
 roclblas_dscal_batched (C++ *function*), 58
 roclblas_dscal_strided_batched (C++ *function*), 59
 roclblas_dspmv (C++ *function*), 77
 roclblas_dspmv_batched (C++ *function*), 77
 roclblas_dspmv_strided_batched (C++ *function*), 78
 roclblas_dspr (C++ *function*), 79
 roclblas_dspr2 (C++ *function*), 84
 roclblas_dspr2_batched (C++ *function*), 85
 roclblas_dspr2_strided_batched (C++ *function*), 86
 roclblas_dspr_batched (C++ *function*), 80
 roclblas_dspr_strided_batched (C++ *function*), 82
 roclblas_dswap (C++ *function*), 60
 roclblas_dswap_batched (C++ *function*), 61
 roclblas_dswap_strided_batched (C++ *function*), 61
 roclblas_dsymm (C++ *function*), 151
 roclblas_dsymm_batched (C++ *function*), 152
 roclblas_dsymm_strided_batched (C++ *function*),
 154
 roclblas_dsymv (C++ *function*), 88
 roclblas_dsymv_batched (C++ *function*), 89
 roclblas_dsymv_strided_batched (C++ *function*), 90
 roclblas_dsyr (C++ *function*), 91
 roclblas_dsyr2 (C++ *function*), 94
 roclblas_dsyr2_batched (C++ *function*), 95
 roclblas_dsyr2_strided_batched (C++ *function*), 96

rocnblas_dsytr2k (C++ function), 159
 rocnblas_dsytr2k_batched (C++ function), 161
 rocnblas_dsytr2k_strided_batched (C++ function), 162
 rocnblas_dsytr_batched (C++ function), 92
 rocnblas_dsytr_strided_batched (C++ function), 93
 rocnblas_dsytrk (C++ function), 155
 rocnblas_dsytrk_batched (C++ function), 156
 rocnblas_dsytrk_strided_batched (C++ function), 158
 rocnblas_dsytrkx (C++ function), 164
 rocnblas_dsytrkx_batched (C++ function), 165
 rocnblas_dsytrkx_strided_batched (C++ function), 167
 rocnblas_dtbmv (C++ function), 97
 rocnblas_dtbmv_batched (C++ function), 98
 rocnblas_dtbmv_strided_batched (C++ function), 100
 rocnblas_dtbsv (C++ function), 103
 rocnblas_dtbsv_batched (C++ function), 104
 rocnblas_dtbsv_strided_batched (C++ function), 105
 rocnblas_dtpmv (C++ function), 107
 rocnblas_dtpmv_batched (C++ function), 108
 rocnblas_dtpmv_strided_batched (C++ function), 108
 rocnblas_dtpsv (C++ function), 110
 rocnblas_dtpsv_batched (C++ function), 111
 rocnblas_dtpsv_strided_batched (C++ function), 112
 rocnblas_dtrmm (C++ function), 169
 rocnblas_dtrmm_batched (C++ function), 174
 rocnblas_dtrmm_strided_batched (C++ function), 178
 rocnblas_dtrmv (C++ function), 113
 rocnblas_dtrmv_batched (C++ function), 114
 rocnblas_dtrmv_strided_batched (C++ function), 115
 rocnblas_dtrsm (C++ function), 184
 rocnblas_dtrsm_batched (C++ function), 185
 rocnblas_dtrsm_strided_batched (C++ function), 187
 rocnblas_dtrsv (C++ function), 116
 rocnblas_dtrsv_batched (C++ function), 117
 rocnblas_dtrsv_strided_batched (C++ function), 118
 rocnblas_dtrtri (C++ function), 204
 rocnblas_dtrtri_batched (C++ function), 205
 rocnblas_dtrtri_strided_batched (C++ function), 205
 rocnblas_dzasum (C++ function), 36
 rocnblas_dzasum_batched (C++ function), 37
 rocnblas_dzasum_strided_batched (C++ function), 37
 rocnblas_dznrm2 (C++ function), 46
 rocnblas_dznrm2_batched (C++ function), 46
 rocnblas_dznrm2_strided_batched (C++ function), 47
 rocnblas_fill (C++ enum), 23
 rocnblas_fill::rocblas_fill_full (C++ enumerator), 23
 rocnblas_fill::rocblas_fill_lower (C++ enumerator), 23
 rocnblas_fill::rocblas_fill_upper (C++ enumerator), 23
 rocnblas_float_complex (C++ struct), 22
 rocnblas_gemm_algo (C++ enum), 27
 rocnblas_gemm_algo::rocblas_gemm_algo_solution_index (C++ enumerator), 27
 rocnblas_gemm_algo::rocblas_gemm_algo_standard (C++ enumerator), 27
 rocnblas_gemm_batched_ex (C++ function), 226
 rocnblas_gemm_batched_ex_get_solutions (C++ function), 245
 rocnblas_gemm_ex (C++ function), 223
 rocnblas_gemm_ex_get_solutions (C++ function), 244
 rocnblas_gemm_flags (C++ enum), 27
 rocnblas_gemm_flags::rocblas_gemm_flags_check_solution_index (C++ enumerator), 28
 rocnblas_gemm_flags::rocblas_gemm_flags_fp16_alt_impl (C++ enumerator), 28
 rocnblas_gemm_flags::rocblas_gemm_flags_fp16_alt_impl_rnz (C++ enumerator), 28
 rocnblas_gemm_flags::rocblas_gemm_flags_none (C++ enumerator), 27
 rocnblas_gemm_flags::rocblas_gemm_flags_pack_int8x4 (C++ enumerator), 27
 rocnblas_gemm_flags::rocblas_gemm_flags_use_cu_efficiency (C++ enumerator), 28
 rocnblas_gemm_strided_batched_ex (C++ function), 228
 rocnblas_gemm_strided_batched_ex_get_solutions (C++ function), 246
 rocnblas_get_atomics_mode (C++ function), 28
 rocnblas_get_device_memory_size (C++ function), 31
 rocnblas_get_matrix (C++ function), 29
 rocnblas_get_matrix_async (C++ function), 30
 rocnblas_get_pointer_mode (C++ function), 28
 rocnblas_get_stream (C++ function), 28
 rocnblas_get_vector (C++ function), 29
 rocnblas_get_version_string (C++ function), 33
 rocnblas_get_version_string_size (C++ function), 33
 rocnblas_half (C++ struct), 21
 rocnblas_handle (C++ type), 21
 rocnblas_haxpy (C++ function), 38

rocnblas_haxpy_batched (C++ function), 39
 rocnblas_haxpy_strided_batched (C++ function), 39
 rocnblas_hdot (C++ function), 42
 rocnblas_hdot_batched (C++ function), 43
 rocnblas_hdot_strided_batched (C++ function), 44
 rocnblas_hgemm (C++ function), 147
 rocnblas_hgemm_batched (C++ function), 148
 rocnblas_hgemm_strided_batched (C++ function), 149
 rocnblas_icamax (C++ function), 33
 rocnblas_icamax_batched (C++ function), 33
 rocnblas_icamax_strided_batched (C++ function), 34
 rocnblas_icamin (C++ function), 35
 rocnblas_icamin_batched (C++ function), 35
 rocnblas_icamin_strided_batched (C++ function), 36
 rocnblas_idamax (C++ function), 33
 rocnblas_idamax_batched (C++ function), 33
 rocnblas_idamax_strided_batched (C++ function), 34
 rocnblas_idamin (C++ function), 35
 rocnblas_idamin_batched (C++ function), 35
 rocnblas_idamin_strided_batched (C++ function), 36
 rocnblas_initialize (C++ function), 31
 rocnblas_int (C++ type), 21
 rocnblas_is_managing_device_memory (C++ function), 32
 rocnblas_is_user_managing_device_memory (C++ function), 32
 rocnblas_isamax (C++ function), 33
 rocnblas_isamax_batched (C++ function), 33
 rocnblas_isamax_strided_batched (C++ function), 34
 rocnblas_isamin (C++ function), 35
 rocnblas_isamin_batched (C++ function), 35
 rocnblas_isamin_strided_batched (C++ function), 36
 rocnblas_izamax (C++ function), 33
 rocnblas_izamax_batched (C++ function), 33
 rocnblas_izamax_strided_batched (C++ function), 34
 rocnblas_izamin (C++ function), 35
 rocnblas_izamin_batched (C++ function), 35
 rocnblas_izamin_strided_batched (C++ function), 36
 rocnblas_layer_mode (C++ enum), 27
 rocnblas_layer_mode::rocblas_layer_mode_log_barrier (C++ enumerator), 27
 rocnblas_layer_mode::rocblas_layer_mode_log_profile (C++ enumerator), 27
 rocnblas_layer_mode::rocblas_layer_mode_log_trace (C++ enumerator), 27
 rocnblas_layer_mode::rocblas_layer_mode_none (C++ enumerator), 27
 rocnblas_nrm2_batched_ex (C++ function), 216
 rocnblas_nrm2_ex (C++ function), 215
 rocnblas_nrm2_strided_batched_ex (C++ function), 217
 rocnblas_operation (C++ enum), 22
 rocnblas_operation::rocblas_operation_conjugate_transpose (C++ enumerator), 22
 rocnblas_operation::rocblas_operation_none (C++ enumerator), 22
 rocnblas_operation::rocblas_operation_transpose (C++ enumerator), 22
 rocnblas_pointer_mode (C++ enum), 26
 rocnblas_pointer_mode::rocblas_pointer_mode_device (C++ enumerator), 26
 rocnblas_pointer_mode::rocblas_pointer_mode_host (C++ enumerator), 26
 rocnblas_pointer_to_mode (C++ function), 29
 rocnblas_query_int8_layout_flag (C++ function), 28
 rocnblas_rot_batched_ex (C++ function), 218
 rocnblas_rot_ex (C++ function), 218
 rocnblas_rot_strided_batched_ex (C++ function), 219
 rocnblas_sasum (C++ function), 36
 rocnblas_sasum_batched (C++ function), 37
 rocnblas_sasum_strided_batched (C++ function), 37
 rocnblas_saxpy (C++ function), 38
 rocnblas_saxpy_batched (C++ function), 38
 rocnblas_saxpy_strided_batched (C++ function), 39
 rocnblas_scal_batched_ex (C++ function), 221
 rocnblas_scal_ex (C++ function), 221
 rocnblas_scal_strided_batched_ex (C++ function), 222
 rocnblas_scasum (C++ function), 36
 rocnblas_scasum_batched (C++ function), 37
 rocnblas_scasum_strided_batched (C++ function), 37
 rocnblas_scnrm2 (C++ function), 46
 rocnblas_scnrm2_batched (C++ function), 46
 rocnblas_scnrm2_strided_batched (C++ function), 47
 rocnblas_scopy (C++ function), 40
 rocnblas_scopy_batched (C++ function), 41
 rocnblas_scopy_strided_batched (C++ function), 41
 rocnblas_sdgemm (C++ function), 240
 rocnblas_sdgemm_batched (C++ function), 241
 rocnblas_sdgemm_strided_batched (C++ function), 242
 rocnblas_sdot (C++ function), 42
 rocnblas_sdot_batched (C++ function), 43
 rocnblas_sdot_strided_batched (C++ function), 44
 rocnblas_set_atomics_mode (C++ function), 28

[roclblas_set_device_memory_size \(C++ function\), 32](#)
[roclblas_set_matrix \(C++ function\), 29](#)
[roclblas_set_matrix_async \(C++ function\), 30](#)
[roclblas_set_pointer_mode \(C++ function\), 28](#)
[roclblas_set_stream \(C++ function\), 28](#)
[roclblas_set_vector \(C++ function\), 29](#)
[roclblas_set_vector_async \(C++ function\), 30](#)
[roclblas_set_workspace \(C++ function\), 32](#)
[roclblas_sgbmv \(C++ function\), 62](#)
[roclblas_sgbmv_batched \(C++ function\), 63](#)
[roclblas_sgbmv_strided_batched \(C++ function\), 65](#)
[roclblas_sgeam \(C++ function\), 237](#)
[roclblas_sgeam_batched \(C++ function\), 238](#)
[roclblas_sgeam_strided_batched \(C++ function\), 239](#)
[roclblas_sgemm \(C++ function\), 147](#)
[roclblas_sgemm_batched \(C++ function\), 148](#)
[roclblas_sgemm_strided_batched \(C++ function\), 149](#)
[roclblas_sgemv \(C++ function\), 67](#)
[roclblas_sgemv_batched \(C++ function\), 68](#)
[roclblas_sgemv_strided_batched \(C++ function\), 69](#)
[roclblas_sger \(C++ function\), 70](#)
[roclblas_sger_batched \(C++ function\), 71](#)
[roclblas_sger_strided_batched \(C++ function\), 72](#)
[roclblas_side \(C++ enum\), 23](#)
[roclblas_side::roclblas_side_both \(C++ enumerator\), 23](#)
[roclblas_side::roclblas_side_left \(C++ enumerator\), 23](#)
[roclblas_side::roclblas_side_right \(C++ enumerator\), 23](#)
[roclblas_snrm2 \(C++ function\), 46](#)
[roclblas_snrm2_batched \(C++ function\), 46](#)
[roclblas_snrm2_strided_batched \(C++ function\), 47](#)
[roclblas_srot \(C++ function\), 48](#)
[roclblas_srot_batched \(C++ function\), 48](#)
[roclblas_srot_strided_batched \(C++ function\), 49](#)
[roclblas_srotg \(C++ function\), 50](#)
[roclblas_srotg_batched \(C++ function\), 51](#)
[roclblas_srotg_strided_batched \(C++ function\), 51](#)
[roclblas_srotm \(C++ function\), 52](#)
[roclblas_srotm_batched \(C++ function\), 53](#)
[roclblas_srotm_strided_batched \(C++ function\), 54](#)
[roclblas_srotmg \(C++ function\), 55](#)
[roclblas_srotmg_batched \(C++ function\), 56](#)
[roclblas_srotmg_strided_batched \(C++ function\), 56](#)
[roclblas_ssbmv \(C++ function\), 74](#)
[roclblas_ssbmv_batched \(C++ function\), 74](#)
[roclblas_ssbmv_strided_batched \(C++ function\), 75](#)
[roclblas_sscal \(C++ function\), 58](#)
[roclblas_sscal_batched \(C++ function\), 58](#)
[roclblas_sscal_strided_batched \(C++ function\), 59](#)
[roclblas_sspmv \(C++ function\), 77](#)
[roclblas_sspmv_batched \(C++ function\), 77](#)
[roclblas_sspmv_strided_batched \(C++ function\), 78](#)
[roclblas_sspr \(C++ function\), 79](#)
[roclblas_sspr2 \(C++ function\), 84](#)
[roclblas_sspr2_batched \(C++ function\), 85](#)
[roclblas_sspr2_strided_batched \(C++ function\), 86](#)
[roclblas_sspr_batched \(C++ function\), 80](#)
[roclblas_sspr_strided_batched \(C++ function\), 82](#)
[roclblas_sswap \(C++ function\), 60](#)
[roclblas_sswap_batched \(C++ function\), 61](#)
[roclblas_sswap_strided_batched \(C++ function\), 61](#)
[roclblas_ssymm \(C++ function\), 151](#)
[roclblas_ssymm_batched \(C++ function\), 152](#)
[roclblas_ssymm_strided_batched \(C++ function\), 153](#)
[roclblas_ssymv \(C++ function\), 88](#)
[roclblas_ssymv_batched \(C++ function\), 89](#)
[roclblas_ssymv_strided_batched \(C++ function\), 90](#)
[roclblas_ssyr \(C++ function\), 91](#)
[roclblas_ssyr2 \(C++ function\), 94](#)
[roclblas_ssyr2_batched \(C++ function\), 94](#)
[roclblas_ssyr2_strided_batched \(C++ function\), 95](#)
[roclblas_ssyr2k \(C++ function\), 159](#)
[roclblas_ssyr2k_batched \(C++ function\), 160](#)
[roclblas_ssyr2k_strided_batched \(C++ function\), 162](#)
[roclblas_ssyr_batched \(C++ function\), 92](#)
[roclblas_ssyr_strided_batched \(C++ function\), 93](#)
[roclblas_ssyrk \(C++ function\), 155](#)
[roclblas_ssyrk_batched \(C++ function\), 156](#)
[roclblas_ssyrk_strided_batched \(C++ function\), 158](#)
[roclblas_ssyrkx \(C++ function\), 164](#)
[roclblas_ssyrkx_batched \(C++ function\), 165](#)
[roclblas_ssyrkx_strided_batched \(C++ function\), 167](#)
[roclblas_start_device_memory_size_query \(C++ function\), 31](#)
[roclblas_status \(C++ enum\), 24](#)
[roclblas_status::roclblas_status_check_numerics_fail \(C++ enumerator\), 24](#)
[roclblas_status::roclblas_status_continue \(C++ enumerator\), 24](#)
[roclblas_status::roclblas_status_internal_error \(C++ enumerator\), 24](#)
[roclblas_status::roclblas_status_invalid_handle \(C++ enumerator\), 24](#)
[roclblas_status::roclblas_status_invalid_pointer \(C++ enumerator\), 24](#)
[roclblas_status::roclblas_status_invalid_size \(C++ enumerator\), 24](#)

rocnblas_status::rocnblas_status_invalid_value (C++ enumerator), 24
 rocnblas_status::rocnblas_status_memory_error (C++ enumerator), 24
 rocnblas_status::rocnblas_status_not_implemented (C++ enumerator), 24
 rocnblas_status::rocnblas_status_perf_degraded (C++ enumerator), 24
 rocnblas_status::rocnblas_status_size_increased (C++ enumerator), 24
 rocnblas_status::rocnblas_status_size_query_mismatch (C++ enumerator), 24
 rocnblas_status::rocnblas_status_size_unchanged (C++ enumerator), 24
 rocnblas_status::rocnblas_status_success (C++ enumerator), 24
 rocnblas_status_to_string (C++ function), 31
 rocnblas_stbmv (C++ function), 97
 rocnblas_stbmv_batched (C++ function), 98
 rocnblas_stbmv_strided_batched (C++ function), 100
 rocnblas_stbsv (C++ function), 103
 rocnblas_stbsv_batched (C++ function), 104
 rocnblas_stbsv_strided_batched (C++ function), 105
 rocnblas_stop_device_memory_size_query (C++ function), 31
 rocnblas_stpmv (C++ function), 107
 rocnblas_stpmv_batched (C++ function), 108
 rocnblas_stpmv_strided_batched (C++ function), 108
 rocnblas_stpsv (C++ function), 110
 rocnblas_stpsv_batched (C++ function), 110
 rocnblas_stpsv_strided_batched (C++ function), 112
 rocnblas_stride (C++ type), 21
 rocnblas_strmm (C++ function), 169
 rocnblas_strmm_batched (C++ function), 173
 rocnblas_strmm_strided_batched (C++ function), 178
 rocnblas_strmv (C++ function), 113
 rocnblas_strmv_batched (C++ function), 114
 rocnblas_strmv_strided_batched (C++ function), 115
 rocnblas_strsm (C++ function), 184
 rocnblas_strsm_batched (C++ function), 185
 rocnblas_strsm_strided_batched (C++ function), 187
 rocnblas_strsv (C++ function), 116
 rocnblas_strsv_batched (C++ function), 117
 rocnblas_strsv_strided_batched (C++ function), 118
 rocnblas_strtri (C++ function), 204
 rocnblas_strtri_batched (C++ function), 205
 rocnblas_strtri_strided_batched (C++ function), 205
 rocnblas_trsm_batched_ex (C++ function), 233
 rocnblas_trsm_ex (C++ function), 231
 rocnblas_trsm_strided_batched_ex (C++ function), 235
 rocnblas_zaxpy (C++ function), 38
 rocnblas_zaxpy_batched (C++ function), 39
 rocnblas_zaxpy_strided_batched (C++ function), 40
 rocnblas_zcopy (C++ function), 40
 rocnblas_zcopy_batched (C++ function), 41
 rocnblas_zcopy_strided_batched (C++ function), 41
 rocnblas_zdgemm (C++ function), 240
 rocnblas_zdgemm_batched (C++ function), 241
 rocnblas_zdgemm_strided_batched (C++ function), 242
 rocnblas_zdotc (C++ function), 43
 rocnblas_zdotc_batched (C++ function), 44
 rocnblas_zdotc_strided_batched (C++ function), 45
 rocnblas_zdotu (C++ function), 43
 rocnblas_zdotu_batched (C++ function), 44
 rocnblas_zdotu_strided_batched (C++ function), 45
 rocnblas_zdrot (C++ function), 48
 rocnblas_zdrot_batched (C++ function), 49
 rocnblas_zdrot_strided_batched (C++ function), 50
 rocnblas_zdscal (C++ function), 58
 rocnblas_zdscal_batched (C++ function), 59
 rocnblas_zdscal_strided_batched (C++ function), 60
 rocnblas_zgbmv (C++ function), 62
 rocnblas_zgbmv_batched (C++ function), 64
 rocnblas_zgbmv_strided_batched (C++ function), 65
 rocnblas_zgeam (C++ function), 237
 rocnblas_zgeam_batched (C++ function), 238
 rocnblas_zgeam_strided_batched (C++ function), 239
 rocnblas_zgemm (C++ function), 147
 rocnblas_zgemm_batched (C++ function), 148
 rocnblas_zgemm_strided_batched (C++ function), 150
 rocnblas_zgemv (C++ function), 67
 rocnblas_zgemv_batched (C++ function), 68
 rocnblas_zgemv_strided_batched (C++ function), 69
 rocnblas_zgerc (C++ function), 70
 rocnblas_zgerc_batched (C++ function), 72
 rocnblas_zgerc_strided_batched (C++ function), 73
 rocnblas_zgeru (C++ function), 70
 rocnblas_zgeru_batched (C++ function), 71
 rocnblas_zgeru_strided_batched (C++ function), 73
 rocnblas_zhbm (C++ function), 122
 rocnblas_zhbm_batched (C++ function), 123
 rocnblas_zhbm_strided_batched (C++ function), 125
 rocnblas_zhemm (C++ function), 189

`rocnblas_zhemm_batched` (C++ function), 190
`rocnblas_zhemm_strided_batched` (C++ function), 191
`rocnblas_zhemv` (C++ function), 119
`rocnblas_zhemv_batched` (C++ function), 120
`rocnblas_zhemv_strided_batched` (C++ function), 121
`rocnblas_zher` (C++ function), 132
`rocnblas_zher2` (C++ function), 135
`rocnblas_zher2_batched` (C++ function), 136
`rocnblas_zher2_strided_batched` (C++ function), 137
`rocnblas_zher2k` (C++ function), 196
`rocnblas_zher2k_batched` (C++ function), 197
`rocnblas_zher2k_strided_batched` (C++ function), 199
`rocnblas_zher_batched` (C++ function), 132
`rocnblas_zher_strided_batched` (C++ function), 133
`rocnblas_zherk` (C++ function), 193
`rocnblas_zherk_batched` (C++ function), 194
`rocnblas_zherk_strided_batched` (C++ function), 195
`rocnblas_zherkx` (C++ function), 200
`rocnblas_zherkx_batched` (C++ function), 201
`rocnblas_zherkx_strided_batched` (C++ function), 203
`rocnblas_zhpmv` (C++ function), 127
`rocnblas_zhpmv_batched` (C++ function), 128
`rocnblas_zhpmv_strided_batched` (C++ function), 130
`rocnblas_zhpr` (C++ function), 138
`rocnblas_zhpr2` (C++ function), 142
`rocnblas_zhpr2_batched` (C++ function), 143
`rocnblas_zhpr2_strided_batched` (C++ function), 145
`rocnblas_zhpr_batched` (C++ function), 139
`rocnblas_zhpr_strided_batched` (C++ function), 141
`rocnblas_zrot` (C++ function), 48
`rocnblas_zrot_batched` (C++ function), 48
`rocnblas_zrot_strided_batched` (C++ function), 49
`rocnblas_zrotg` (C++ function), 50
`rocnblas_zrotg_batched` (C++ function), 51
`rocnblas_zrotg_strided_batched` (C++ function), 52
`rocnblas_zscal` (C++ function), 58
`rocnblas_zscal_batched` (C++ function), 59
`rocnblas_zscal_strided_batched` (C++ function), 59
`rocnblas_zspr` (C++ function), 79
`rocnblas_zspr_batched` (C++ function), 81
`rocnblas_zspr_strided_batched` (C++ function), 82
`rocnblas_zswap` (C++ function), 60
`rocnblas_zswap_batched` (C++ function), 61
`rocnblas_zswap_strided_batched` (C++ function), 61
`rocnblas_zsymm` (C++ function), 151
`rocnblas_zsymm_batched` (C++ function), 152
`rocnblas_zsymm_strided_batched` (C++ function), 154
`rocnblas_zsymv` (C++ function), 88
`rocnblas_zsymv_batched` (C++ function), 89
`rocnblas_zsymv_strided_batched` (C++ function), 90
`rocnblas_zsyr` (C++ function), 91
`rocnblas_zsyr2` (C++ function), 94
`rocnblas_zsyr2_batched` (C++ function), 95
`rocnblas_zsyr2_strided_batched` (C++ function), 96
`rocnblas_zsyr2k` (C++ function), 159
`rocnblas_zsyr2k_batched` (C++ function), 161
`rocnblas_zsyr2k_strided_batched` (C++ function), 162
`rocnblas_zsyr_batched` (C++ function), 92
`rocnblas_zsyr_strided_batched` (C++ function), 93
`rocnblas_zsyrrk` (C++ function), 155
`rocnblas_zsyrrk_batched` (C++ function), 157
`rocnblas_zsyrrk_strided_batched` (C++ function), 158
`rocnblas_zsyrrkx` (C++ function), 164
`rocnblas_zsyrrkx_batched` (C++ function), 166
`rocnblas_zsyrrkx_strided_batched` (C++ function), 167
`rocnblas_ztbmv` (C++ function), 97
`rocnblas_ztbmv_batched` (C++ function), 99
`rocnblas_ztbmv_strided_batched` (C++ function), 101
`rocnblas_ztbsv` (C++ function), 103
`rocnblas_ztbsv_batched` (C++ function), 104
`rocnblas_ztbsv_strided_batched` (C++ function), 105
`rocnblas_ztpmv` (C++ function), 107
`rocnblas_ztpmv_batched` (C++ function), 108
`rocnblas_ztpmv_strided_batched` (C++ function), 109
`rocnblas_ztpsv` (C++ function), 110
`rocnblas_ztpsv_batched` (C++ function), 111
`rocnblas_ztpsv_strided_batched` (C++ function), 112
`rocnblas_ztrmm` (C++ function), 169
`rocnblas_ztrmm_batched` (C++ function), 174
`rocnblas_ztrmm_strided_batched` (C++ function), 179
`rocnblas_ztrmv` (C++ function), 113
`rocnblas_ztrmv_batched` (C++ function), 114
`rocnblas_ztrmv_strided_batched` (C++ function), 115
`rocnblas_ztrsm` (C++ function), 184
`rocnblas_ztrsm_batched` (C++ function), 186
`rocnblas_ztrsm_strided_batched` (C++ function), 187
`rocnblas_ztrsv` (C++ function), 116
`rocnblas_ztrsv_batched` (C++ function), 117

rocblas_ztrsv_strided_batched (C++ *function*),
118