rocSOLVER Documentation

Release 3.11.0

Advanced Micro Devices

Jun 10, 2021

CONTENTS:

1 Legal Disclaimer **1**

2 User guide **3**

2.1 Introduction . . . 3
2.1.1 Overview . . . 3
2.1.2 Brief description and functionality . . . 3
2.2 Building and installation . . . 6
2.2.1 Prerequisites . . . 6
2.2.2 Installing from pre-built packages . . . 6
2.2.3 Building & installing from source . . . 6
2.2.3.1 Using the install.sh script . . . 6
2.2.3.2 Manual building and installation . . . 8
2.3 Using rocSOLVER . . . 9
2.3.1 QR factorization of a single matrix . . . 9
2.3.2 QR factorization of a batch of matrices . . . 12
2.3.2.1 Strided_batched version . . . 12
2.3.2.2 Batched version . . . 13
2.4 Tuning rocSOLVER Performance . . . 15
2.5 Memory model . . . 15
2.6 rocSOLVER API . . . 15
2.6.1 Types . . . 15
2.6.1.1 Additional Types . . . 15
2.6.2 LAPACK Auxiliary Functions . . . 17
2.6.2.1 Complex vector manipulations . . . 17
2.6.2.2 Matrix permutations and manipulations . . . 18
2.6.2.3 Householder reflexions . . . 19
2.6.2.4 Bidiagonal forms . . . 24
2.6.2.5 Tridiagonal forms . . . 28
2.6.2.6 Orthonormal matrices . . . 29
2.6.2.7 Unitary matrices . . . 46
2.6.2.8 Eigensolvers . . . 63
2.6.3 LAPACK Functions . . . 64
2.6.3.1 Special Matrix Factorizations . . . 65
2.6.3.2 General Matrix Factorizations . . . 71
2.6.3.3 General Matrix Diagonalizations . . . 106
2.6.3.4 Symmetric Matrix Diagonalizations . . . 117
2.6.3.5 General Matrix Inversion . . . 134
2.6.3.6 General Systems Solvers . . . 138
2.6.3.7 General Matrix Singular Value Decomposition . . . 142
2.6.4 Lapack-like Functions . . . 150

2.6.4.1 General Matrix Factorizations . . . 150
2.6.5 Deprecated . . . 157
2.6.5.1 Types . . . 157
2.6.5.2 Auxiliary Functions . . . 159

3 Library Design Documentation **163**
3.1 Introduction . . . 163
3.2 Batch rocSOLVER . . . 163
3.3 Clients . . . 163
3.3.1 Testing rocSOLVER . . . 163
3.3.2 Benchmarking rocSOLVER . . . 164

Index **165**

CHAPTER

ONE

LEGAL DISCLAIMER

The information contained herein is for informational purposes only, and is subject to change without notice. In addition, any stated support is planned and is also subject to change. While every precaution has been taken in the preparation of this document, it may contain technical inaccuracies, omissions and typographical errors, and AMD is under no obligation to update or otherwise correct this information. Advanced Micro Devices, Inc. makes no representations or warranties with respect to the accuracy or completeness of the contents of this document, and assumes no liability of any kind, including the implied warranties of noninfringement, merchantability or fitness for particular purposes, with respect to the operation or use of AMD hardware, software or other products described herein. No license, including implied or arising by estoppel, to any intellectual property rights is granted by this document. Terms and limitations applicable to the purchase or use of AMD's products are as set forth in a signed agreement between the parties or in AMD's Standard Terms and Conditions of Sale.

CHAPTER

TWO

USER GUIDE

2.1 Introduction

2.1.1 Overview

rocSOLVER is an implementation of LAPACK routines on top of AMD ROCm. rocSOLVER is implemented in the HIP programming language and optimized for AMD's latest discrete GPUs.

2.1.2 Brief description and functionality

The rocSOLVER library is in the early stages of active development. New features are being continuously added, with new functionality documented at each release of the ROCm platform.

The following table summarizes the LAPACK functionality implemented in rocSOLVER's latest release.

LAPACK Auxiliary Function	single	double	single complex	double complex
rocsolver_lacgv			x	x
rocsolver_laswp	x	x	x	x
rocsolver_larfg	x	x	x	x
rocsolver_larft	x	x	x	x
rocsolver_larf	x	x	x	x
rocsolver_larfb	x	x	x	x
rocsolver_labrd	x	x	x	x
rocsolver_latrd	x	x	x	x
rocsolver_bdsqr	x	x	x	x
rocsolver_org2r	x	x		
rocsolver_orgqr	x	x		
rocsolver_orgl2	x	x		
rocsolver_orglq	x	x		
rocsolver_org2l	x	x		
rocsolver_orgql	x	x		
rocsolver_orgbr	x	x		
rocsolver_orgtr	x	x		
rocsolver_orm2r	x	x		
rocsolver_ormqr	x	x		
rocsolver_orml2	x	x		
rocsolver_ormlq	x	x		
rocsolver_orm2l	x	x		

continues on next page

Table 1 – continued from previous page

LAPACK Auxiliary Function	single	double	single complex	double complex
rocsolver_ormql	x	x		
rocsolver_ormbr	x	x		
rocsolver_ormtr	x	x		
rocsolver_ung2r			x	x
rocsolver_ungqr			x	x
rocsolver_ungl2			x	x
rocsolver_unglq			x	x
rocsolver_ung2l			x	x
rocsolver_ungql			x	x
rocsolver_ungbr			x	x
rocsolver_ungtr			x	x
rocsolver_unm2r			x	x
rocsolver_unmqr			x	x
rocsolver_unml2			x	x
rocsolver_unmlq			x	x
rocsolver_unm2l			x	x
rocsolver_unmql			x	x
rocsolver_unmbr			x	x
rocsolver_unmtr			x	x
rocsolver_sterf	x	x		
rocsolver_steqr	x	x	x	x

LAPACK Function	single	double	single complex	double complex
rocsolver_potf2	x	x	x	x
rocsolver_potf2_batched	x	x	x	x
rocsolver_potf2_strided_batched	x	x	x	x
rocsolver_potrf	x	x	x	x
rocsolver_potrf_batched	x	x	x	x
rocsolver_potrf_strided_batched	x	x	x	x
rocsolver_getf2	x	x	x	x
rocsolver_getf2_batched	x	x	x	x
rocsolver_getf2_strided_batched	x	x	x	x
rocsolver_getrf	x	x	x	x
rocsolver_getrf_batched	x	x	x	x
rocsolver_getrf_strided_batched	x	x	x	x
rocsolver_geqr2	x	x	x	x
rocsolver_geqr2_batched	x	x	x	x
rocsolver_geqr2_strided_batched	x	x	x	x
rocsolver_geqrf	x	x	x	x
rocsolver_geqrf_batched	x	x	x	x
rocsolver_geqrf_strided_batched	x	x	x	x
rocsolver_geql2	x	x	x	x
rocsolver_geql2_batched	x	x	x	x
rocsolver_geql2_strided_batched	x	x	x	x
rocsolver_geqlf	x	x	x	x
rocsolver_geqlf_batched	x	x	x	x
rocsolver_geqlf_strided_batched	x	x	x	x
rocsolver_gelq2	x	x	x	x

continues on next page

Table 2 – continued from previous page

LAPACK Function	single	double	single complex	double complex
rocsolver_gelq2_batched	x	x	x	x
rocsolver_gelq2_strided_batched	x	x	x	x
rocsolver_gelqf	x	x	x	x
rocsolver_gelqf_batched	x	x	x	x
rocsolver_gelqf_strided_batched	x	x	x	x
rocsolver_getrs	x	x	x	x
rocsolver_getrs_batched	x	x	x	x
rocsolver_getrs_strided_batched	x	x	x	x
rocsolver_getri	x	x	x	x
rocsolver_getri_batched	x	x	x	x
rocsolver_getri_strided_batched	x	x	x	x
rocsolver_gebd2	x	x	x	x
rocsolver_gebd2_batched	x	x	x	x
rocsolver_gebd2_strided_batched	x	x	x	x
rocsolver_gebrd	x	x	x	x
rocsolver_gebrd_batched	x	x	x	x
rocsolver_gebrd_strided_batched	x	x	x	x
rocsolver_gesvd	x	x	x	x
rocsolver_gesvd_batched	x	x	x	x
rocsolver_gesvd_strided_batched	x	x	x	x
rocsolver_sytd2	x	x		
rocsolver_sytd2_batched	x	x		
rocsolver_sytd2_strided_batched	x	x		
rocsolver_sytrd	x	x		
rocsolver_sytrd_batched	x	x		
rocsolver_sytrd_strided_batched	x	x		
rocsolver_hetd2			x	x
rocsolver_hetd2_batched			x	x
rocsolver_hetd2_strided_batched			x	x
rocsolver_hetrd			x	x
rocsolver_hetrd_batched			x	x
rocsolver_hetrd_strided_batched			x	x

Lapack-like Function	single	double	single complex	double complex
rocsolver_getf2_npvt	x	x	x	x
rocsolver_getf2_npvt_batched	x	x	x	x
rocsolver_getf2_npvt_strided_batched	x	x	x	x
rocsolver_getrf_npvt	x	x	x	x
rocsolver_getrf_npvt_batched	x	x	x	x
rocsolver_getrf_npvt_strided_batched	x	x	x	x
rocsolver_gels	x	x	x	x
rocsolver_gels_batched	x	x	x	x
rocsolver_gels_strided_batched	x	x	x	x

2.2 Building and installation

2.2.1 Prerequisites

rocSOLVER requires a ROCm-enabled platform. For more information, see the ROCm install guide.

rocSOLVER also requires a compatible version of rocBLAS installed on the system. For more information, see the rocBLAS install guide.

rocBLAS and rocSOLVER are both still under active development, and it is hard to define minimal compatibility versions. For now, a good rule of thumb is to always use rocSOLVER together with the matching rocBLAS version. For example, if you want to install rocSOLVER from ROCm 3.3 release, then be sure that ROCm 3.3 rocBLAS is also installed; if you are building the rocSOLVER branch tip, then you will need to build and install rocBLAS branch tip as well.

2.2.2 Installing from pre-built packages

If you have added the ROCm repositories to your Linux distribution, the latest release version of rocSOLVER can be installed using a package manager. On Ubuntu, for example, use the commands:

```
sudo apt-get update
sudo apt-get install rocsolver
```

2.2.3 Building & installing from source

The rocSOLVER source code is hosted on GitHub. Download the code and checkout the desired branch using:

```
git clone -b <desired_branch_name> https://github.com/ROCmSoftwarePlatform/rocSOLVER.git
cd rocSOLVER
```

To build from source, some external dependencies such as CMake and Python are required. Additionally, if the library clients are to be built (by default they are not), then LAPACK, Boost and GoogleTest will be also required. (The library clients: rocsolver-test and rocsolver-bench, provide the infrastructure for testing and benchmarking rocSOLVER. For more details on the library clients see the Design Documentation here: *Clients*).

2.2.3.1 Using the install.sh script

It is recommended that the provided install.sh script be used to build and install rocSOLVER. The command

```
./install.sh --help
```

gives detailed information on how to use this installation script.

Next, some common use cases are listed:

```
./install.sh
```

This command builds rocSOLVER and puts the generated library files, such as headers and `librocsolver.so`, in the output directory: `rocSOLVER/build/release/rocsolver-install`. Other output files from the configuration and building process can also be found at `rocSOLVER/build` and `rocSOLVER/build/release` directories. It is assumed that all external library dependencies have been installed. It also assumes that rocBLAS library is located at: `/opt/rocm/rocblas`.

```
./install.sh -g
```

Use the -g flag to build in debug mode. In this case the generated library files will be located at `rocSOLVER/build/debug/rocsolver-install`. Other output files from the configuration and building process can also be found at `rocSOLVER/build` and `rocSOLVER/build/debug` directories

```
./install.sh --lib_dir /home/user/rocsolverlib --build_dir buildoutput
```

Use `--lib_dir` and `--build_dir` to change output directories. In this case, for example, the installer will put the headers and library files at `/home/user/rocsolverlib`, while the outputs of the configure and building process will be at `rocSOLVER/buildoutput` and `rocSOLVER/buildoutput/release`. The selected output directories must be local, otherwise the user may require sudo privileges. To install rocSOLVER system-wide, we recommend the use of the -i flag as showed below.

```
./install.sh --rocblas_dir /alternative/rocblas/location
```

Use `--rocblas_dir` to change where the rocBLAS library will be looked for. In this case, for example, the installer will look for the rocBLAS library at `/alternative/rocblas/location`.

```
./install.sh -s
```

With the -s flag, the installer will generate a static library (`librocsolver.a`) instead.

```
./install.sh -h
```

With the -h flag, the installer will build rocSOLVER using the hip-clang compiler.

```
./install.sh -d
```

With the -d flag, the installer will first install all the external dependencies required by rocSOLVER library in `/usr/local`. This flag only needs to be used once. For subsequent invocations of install.sh it is not necessary to rebuild the dependencies.

```
./install.sh -c
```

With the -c flag, the installer will additionally build the library clients `rocsolver-bench` and `rocsolver-test`. The binaries will be located at `rocSOLVER/build/release/clients/staging`. It is assumed that all the client external dependencies have been installed.

```
./install.sh -dc
```

By combining c and d flags, the installer will also install all the external dependencies required by rocSOLVER clients. The -d flag only needs to be used once. For subsequent invocations of install.sh it is not necessary to rebuild the dependencies.

```
./install.sh -i
```

With the -i flag, the installer will additionally generate a pre-built rocSOLVER package and install it, using a suitable package manager, at the standard location `/opt/rocm/rocsolver`. This is the preferred approach to install rocSOLVER in a system. This way the library could be also safely removed using the package manager.

```
./install.sh -p
```

With the -p flag, the installer will also generate the rocSOLVER package, but it will not be installed.

```
./install.sh -i --install_dir /package/install/path
```

When generating a package, use `--install_dir` to change the directory where it will be installed. In this case, for example, rocSOLVER package will be installed at `/package/install/path`

2.2.3.2 Manual building and installation

Manual installation of all the external dependencies is not an easy task. Get more information on how to install each dependency at their corresponding documentation sources:

- CMake (version >3.5 is required).
- Python (version >2.7 is required. Python is installed by default in some systems like Ubuntu).
- Boost
- LAPACK (which internally depends on a Fortran compiler), and
- GoogleTest

Once all dependencies are installed (including ROCm and rocBLAS), rocSOLVER can be manually built using a combination of CMake and Make commands. Using CMake options could provide more flexibility to tailor the building and installation process. Here we just provide a list of examples of common use cases (see the CMake documentation for more information on CMake options).

```
mkdir -p build/release && cd build/release
CXX=/opt/rocm/bin/hcc cmake -DCMAKE_INSTALL_PREFIX=rocsolver-install ../..
make install
```

This is equivalent to `./install.sh`.

```
mkdir -p buildoutput/release && cd buildoutput/release
CXX=/opt/rocm/bin/hcc cmake -DCMAKE_INSTALL_PREFIX=/home/user/rocsolverlib ../..
make install
```

This is equivalent to `./install.sh --lib_dir /home/user/rocsolverlib --build_dir buildoutput`.

```
mkdir -p build/release && cd build/release
CXX=/opt/rocm/bin/hcc cmake -DCMAKE_INSTALL_PREFIX=rocsolver-install -Drocblas_DIR=/
↪alternative/rocblas/location ../..
make install
```

This is equivalent to `./install.sh --rocblas_dir /alternative/rocblas/location`.

```
mkdir -p build/debug && cd build/debug
CXX=/opt/rocm/bin/hcc cmake -DCMAKE_INSTALL_PREFIX=rocsolver-install -DCMAKE_BUILD_
↪TYPE=Debug ../..
make install
```

This is equivalent to `./install.sh -g`.

```
mkdir -p build/release && cd build/release
CXX=/opt/rocm/bin/hcc cmake -DCMAKE_INSTALL_PREFIX=rocsolver-install -DBUILD_SHARED_
↪LIBS=OFF ../..
make install
```

This is equivalent to `./install.sh -s`.

```
mkdir -p build/release && cd build/release
CXX=/opt/rocm/bin/hipcc cmake -DCMAKE_INSTALL_PREFIX=rocsolver-install ../..
make install
```

This is equivalent to `./install.sh -h`.

```
mkdir -p build/release && cd build/release
CXX=/opt/rocm/bin/hcc cmake -DCMAKE_INSTALL_PREFIX=rocsolver-install -DBUILD_CLIENTS_
↪TESTS=ON -DBUILD_CLIENTS_BENCHMARKS=ON ../..
make install
```

This is equivalent to `./install.sh -c`.

```
mkdir -p build/release && cd build/release
CXX=/opt/rocm/bin/hcc cmake -DCMAKE_INSTALL_PREFIX=rocsolver-install -DCPACK_SET_
↪DESTDIR=OFF -DCPACK_PACKAGING_INSTALL_PREFIX=/opt/rocm ../..
make install
make package
```

This is equivalent to `./install.sh -p`.

```
mkdir -p build/release && cd build/release
CXX=/opt/rocm/bin/hcc cmake -DCMAKE_INSTALL_PREFIX=rocsolver-install -DCPACK_SET_
↪DESTDIR=OFF -DCPACK_PACKAGING_INSTALL_PREFIX=/package/install/path ../..
make install
make package
sudo dpkg -i rocsolver[-\_]*.deb
```

On an Ubuntu system, for example, this would be equivalent to `./install.sh -i --install_dir /package/install/path`.

2.3 Using rocSOLVER

Once installed, rocSOLVER can be used just like any other library with a C API. The header file will need to be included in the user code, and both the rocBLAS and rocSOLVER shared libraries will become link-time and run-time dependencies for the user application.

2.3.1 QR factorization of a single matrix

The following code snippet uses rocSOLVER to compute the QR factorization of a general m-by-n real matrix in double precision. For a full description of the used rocSOLVER routine, see the API documentation here: *rocsolver_<type>geqrf()*.

```
////////////////////////////
// example.cpp source code //
////////////////////////////

#include <algorithm> // for std::min
#include <stdio.h>   // for size_t, printf
#include <vector>
#include <hip/hip_runtime_api.h> // for hip functions
```

(continues on next page)

(continued from previous page)

```
#include <rocsolver.h> // for all the rocsolver C interfaces and type declarations

// Example: Compute the QR Factorization of a matrix on the GPU

void get_example_matrix(std::vector<double>& hA,
                        rocblas_int& M,
                        rocblas_int& N,
                        rocblas_int& lda) {
  // a *very* small example input; not a very efficient use of the API
  const double A[3][3] = { {  12, -51,   4},
                           {   6, 167, -68},
                           {  -4,  24, -41} };
  M = 3;
  N = 3;
  lda = 3;
  // note: rocsolver matrices must be stored in column major format,
  //       i.e. entry (i,j) should be accessed by hA[i + j*lda]
  hA.resize(size_t(lda) * N);
  for (size_t i = 0; i < M; ++i) {
    for (size_t j = 0; j < N; ++j) {
      // copy A (2D array) into hA (1D array, column-major)
      hA[i + j*lda] = A[i][j];
    }
  }
}

// We use rocsolver_dgeqrf to factor a real M-by-N matrix, A.
// See https://rocsolver.readthedocs.io/en/latest/userguide_api.html#_CPPv416rocsolver_
↪dgeqrf14rocblas_handleK11rocblas_intK11rocblas_intPdK11rocblas_intPd
// and https://www.netlib.org/lapack/explore-html/df/dc5/group__variants_g_
↪ecomputational_ga3766ea903391b5cf9008132f7440ec7b.html
int main() {
  rocblas_int M;          // rows
  rocblas_int N;          // cols
  rocblas_int lda;        // leading dimension
  std::vector<double> hA; // input matrix on CPU
  get_example_matrix(hA, M, N, lda);

  // let's print the input matrix, just to see it
  printf("A = [\n");
  for (size_t i = 0; i < M; ++i) {
    printf("  ");
    for (size_t j = 0; j < N; ++j) {
      printf("% .3f ", hA[i + j*lda]);
    }
    printf(";\n");
  }
  printf("]\n");

  // initialization
  rocblas_handle handle;
  rocblas_create_handle(&handle);
```

(continues on next page)

(continued from previous page)

```
  // calculate the sizes of our arrays
  size_t size_A = size_t(lda) * N;          // count of elements in matrix A
  size_t size_piv = size_t(std::min(M, N)); // count of Householder scalars

  // allocate memory on GPU
  double *dA, *dIpiv;
  hipMalloc(&dA, sizeof(double)*size_A);
  hipMalloc(&dIpiv, sizeof(double)*size_piv);

  // copy data to GPU
  hipMemcpy(dA, hA.data(), sizeof(double)*size_A, hipMemcpyHostToDevice);

  // compute the QR factorization on the GPU
  rocsolver_dgeqrf(handle, M, N, dA, lda, dIpiv);

  // copy the results back to CPU
  std::vector<double> hIpiv(size_piv); // array for householder scalars on CPU
  hipMemcpy(hA.data(), dA, sizeof(double)*size_A, hipMemcpyDeviceToHost);
  hipMemcpy(hIpiv.data(), dIpiv, sizeof(double)*size_piv, hipMemcpyDeviceToHost);

  // the results are now in hA and hIpiv
  // we can print some of the results if we want to see them
  printf("R = [\n");
  for (size_t i = 0; i < M; ++i) {
    printf("  ");
    for (size_t j = 0; j < N; ++j) {
      printf("% .3f ", (i <= j) ? hA[i + j*lda] : 0);
    }
    printf(";\n");
  }
  printf("]\n");

  // clean up
  hipFree(dA);
  hipFree(dIpiv);
  rocblas_destroy_handle(handle);
}
```

The exact command used to compile the example above may vary depending on the system environment, but here is a typical example:

```
/opt/rocm/bin/hipcc -I/opt/rocm/include -c example.cpp
/opt/rocm/bin/hipcc -o example -L/opt/rocm/lib -lrocsolver -lrocblas example.o
```

2.3.2 QR factorization of a batch of matrices

One of the advantages of using GPUs is the ability to execute in parallel many operations of the same type but on different data sets. Based on this idea, rocSOLVER and rocBLAS provide a *batch* version of most of their routines. These batch versions allow the user to execute the same operation on a set of different matrices and/or vectors with a single library call. For more details on the approach to batch functionality followed in rocSOLVER, see *Batch rocSOLVER*.

2.3.2.1 Strided_batched version

The following code snippet uses rocSOLVER to compute the QR factorization of a series of general m-by-n real matrices in double precision. The matrices must be stored in contiguous memory locations on the GPU, and are accessed by a pointer to the first matrix and a stride value that gives the separation between one matrix and the next one. For a full description of the used rocSOLVER routine, see the API documentation here: *rocsolver_<type>geqrf_strided_batched()*.

```
////////////////////////////////////
// example_strided.cpp source code //
////////////////////////////////////

#include <iostream>
#include <stdlib.h>
#include <vector>
#include <rocsolver.h>      //  this includes all the rocsolver C interfaces and type␣
↪declarations

using namespace std;

int main() {
    rocblas_int M;
    rocblas_int N;
    rocblas_int lda;
    rocblas_int batch_count;  // number of matrices to factorize in the batch

    // initialize batch_count, M, N and lda with desired values
    // here===>>

    rocblas_handle handle;
    rocblas_create_handle(&handle); // this creates the rocblas handle

    rocblas_int strideA = lda*N;                    // separation between two matrices in␣
↪memory
    size_t size_A = size_t(strideA)*batch_count;    // size of the array that holds the␣
↪matrices
    rocblas_int strideP = min(M,N);                 // separation between two sets of␣
↪Householder scalars
    size_t size_piv = size_t(strideP)*batch_count; // size of the array that will have␣
↪the Householder scalars

    vector<double> hA(size_A);        // creates array for matrices in CPU
    vector<double> hIpiv(size_piv);   // creates array for householder scalars in CPU

    double *dA, *dIpiv;
    hipMalloc(&dA,sizeof(double)*size_A);       // allocates memory for matrices in GPU
```

(continues on next page)

(continued from previous page)

```
    hipMalloc(&dIpiv,sizeof(double)*size_piv);  // allocates memory for scalars in GPU

    // initialize all matrices (array hA) with input data
    // here===>>
    // ( matrices must be stored in column major format, i.e. entry (i,j)
    //   of the k-th matrix in the batch should be accessed by hA[i + j*lda + k*strideA]␣
↪)

    hipMemcpy(dA,hA.data(),sizeof(double)*size_A,hipMemcpyHostToDevice);          //␣
↪copy data to GPU

    rocsolver_dgeqrf_strided_batched(handle, M, N, dA,
                                     lda, strideA, dIpiv, strideP, batch_count);  //␣
↪compute the QR factorization on the GPU

    hipMemcpy(hA.data(),dA,sizeof(double)*size_A,hipMemcpyDeviceToHost);          //␣
↪copy the results back to CPU
    hipMemcpy(hIpiv.data(),dIpiv,sizeof(double)*size_piv,hipMemcpyDeviceToHost);

    // do something with the results in hA and hIpiv
    // here===>>

    hipFree(dA);                     // de-allocate GPU memory
    hipFree(dIpiv);
    rocblas_destroy_handle(handle);  // destroy handle

    return 0;
}
```

2.3.2.2 Batched version

The following code snippet uses rocSOLVER to compute the QR factorization of a series of general m-by-n real matrices in double precision. The matrices do not need to be in contiguous memory locations on the GPU, and will be accessed by an array of pointers. For a full description of the used rocSOLVER routine, see the API documentation here: *rocsolver_<type>geqrf_batched()*.

```
////////////////////////////////////
// example_batched.cpp source code //
////////////////////////////////////

#include <iostream>
#include <stdlib.h>
#include <vector>
#include <rocsolver.h>      //  this includes all the rocsolver C interfaces and type␣
↪declarations

using namespace std;

int main() {
    rocblas_int M;
    rocblas_int N;
```

(continues on next page)

(continued from previous page)

```
    rocblas_int lda;
    rocblas_int batch_count;  // number of matrices to factorize in the batch

    // initialize batch_count, M, N and lda with desired values
    // here===>>

    rocblas_handle handle;
    rocblas_create_handle(&handle); // this creates the rocblas handle

    size_t size_A = size_t(lda)*N;                 // size of the array that holds one␣
↪matrix
    rocblas_int strideP = min(M,N);                // separation between two sets of␣
↪Householder scalars
    size_t size_piv = size_t(strideP)*batch_count; // size of the array that will have␣
↪the Householder scalars

    vector<double> hIpiv(size_piv);          // creates array for householder scalars in␣
↪CPU
    vector<double> hA[batch_count];          // creates array on the CPU of pointers to␣
↪the CPU
    for(int b=0; b < batch_count; ++b) {
        hA[b] = vector<double>(size_A);      // each pointer will point to a matrix of␣
↪the batch on the CPU
    }

    double* A[batch_count];                      // creates array on the CPU of pointers␣
↪to the GPU
    for (int b = 0; b < batch_count; ++b)
        hipMalloc(&A[b], sizeof(double)*size_A); // each pointer will point to a matrix␣
↪of the batch on the GPU

    double **dA, *dIpiv;
    hipMalloc(&dA,sizeof(double*) * size_A);    // array on the GPU of pointers on the␣
↪GPU
    hipMalloc(&dIpiv,sizeof(double)*size_piv);  // allocates memory for scalars in GPU

    // initialize all matrices (arrays hA[k]) with input data
    // here===>>
    // ( matrices must be stored in column major format, i.e. entry (i,j)
    //   of the k-th matrix in the batch should be accessed by hA[k][i + j*lda] )

    for(int b=0; b < batch_count; ++b)
        hipMemcpy(A[b], hA[b].data(), sizeof(double)*size_A, hipMemcpyHostToDevice);    /
↪/ copy data to GPU
    hipMemcpy(dA, A, sizeof(double*) * batch_count, hipMemcpyHostToDevice);             /
↪/ copy pointers to GPU

    rocsolver_dgeqrf_batched(handle, M, N, dA,
                             lda, dIpiv, strideP, batch_count);  // compute the QR␣
↪factorization on the GPU

    for(int b=0;b<batch_count;b++)
```

(continues on next page)

(continued from previous page)

```
        hipMemcpy(hA[b].data(), A[b], sizeof(double) * size_A, hipMemcpyDeviceToHost); //
↪ copy the results back
    hipMemcpy(hIpiv.data(),dIpiv,sizeof(double)*size_piv,hipMemcpyDeviceToHost);

    // do something with the results in hA and hIpiv
    // here===>>

    for(int b=0;b<batch_count;++b)
        hipFree(A[b]);
    hipFree(dA);                        // de-allocate GPU memory
    hipFree(dIpiv);
    rocblas_destroy_handle(handle);    // destroy handle

    return 0;
}
```

2.4 Tuning rocSOLVER Performance

More to come later…

2.5 Memory model

More to come later…

2.6 rocSOLVER API

This section provides details of the rocSOLVER library API.

2.6.1 Types

rocSOLVER uses types and enumerations defined by the rocBLAS API. For more information, see the rocBLAS types.

2.6.1.1 Additional Types

These are types that extend the rocBLAS API.

rocblas_direct

enum **rocblas_direct**
Used to specify the order in which multiple elementary matrices are applied together.

Values:

enumerator **rocblas_forward_direction**
Elementary matrices applied from the right.

enumerator **rocblas_backward_direction**
Elementary matrices applied from the left.

rocblas_storev

enum **rocblas_storev**
Used to specify how householder vectors are stored in a matrix of vectors.

Values:

enumerator **rocblas_column_wise**
Householder vectors are stored in the columns of a matrix.

enumerator **rocblas_row_wise**
Householder vectors are stored in the rows of a matrix.

rocblas_svect

enum **rocblas_svect**
Used to specify how the singular vectors are to be computed and stored.

Values:

enumerator **rocblas_svect_all**
The entire associated orthogonal/unitary matrix is computed.

enumerator **rocblas_svect_singular**
Only the singular vectors are computed and stored in output array.

enumerator **rocblas_svect_overwrite**
Only the singular vectors are computed and overwrite the input matrix.

enumerator **rocblas_svect_none**
No singular vectors are computed.

rocblas_evect

enum **rocblas_evect**
Used to specify how the eigenvectors are to be computed.

Values:

enumerator **rocblas_evect_original**
Compute eigenvectors for the original symmetric/Hermitian matrix.

enumerator **rocblas_evect_tridiagonal**
Compute eigenvectors for the symmetric tridiagonal matrix.

enumerator **rocblas_evect_none**
No eigenvectors are computed.

rocblas_workmode

enum **rocblas_workmode**
Used to enable the use of fast algorithms (with out-of-place computations) in some of the routines.

Values:

enumerator **rocblas_outofplace**
Out-of-place computations are allowed; this requires enough free memory.

enumerator **rocblas_inplace**
When not enough memory, this forces in-place computations

2.6.2 LAPACK Auxiliary Functions

These are functions that support more advanced LAPACK routines.

2.6.2.1 Complex vector manipulations

rocsolver_<type>lacgv()

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

rocblas_status **rocsolver_clacgv**(rocblas_handle handle, const rocblas_int n, rocblas_float_complex *x, const rocblas_int incx)
LACGV conjugates the complex vector x.

It conjugates the n entries of a complex vector x with increment incx.

Parameters

- **handle** – **[in]** rocblas_handle

- **n** – **[in]** rocblas_int. n >= 0.

  The number of entries of the vector x.

- **x** – **[inout]** pointer to type. Array on the GPU of size at least n.

  On input it is the vector x, on output it is overwritten with vector conjg(x).

- **incx** – **[in]** rocblas_int. incx != 0.

  The increment between consecutive elements of x. If incx is negative, the elements of x are indexed in reverse order.

2.6.2.2 Matrix permutations and manipulations

rocsolver_<type>laswp()

rocblas_status **rocsolver_zlaswp**(rocblas_handle handle, const rocblas_int n, rocblas_double_complex *A, const rocblas_int lda, const rocblas_int k1, const rocblas_int k2, const rocblas_int *ipiv, const rocblas_int incx)

rocblas_status **rocsolver_claswp**(rocblas_handle handle, const rocblas_int n, rocblas_float_complex *A, const rocblas_int lda, const rocblas_int k1, const rocblas_int k2, const rocblas_int *ipiv, const rocblas_int incx)

rocblas_status **rocsolver_dlaswp**(rocblas_handle handle, const rocblas_int n, double *A, const rocblas_int lda, const rocblas_int k1, const rocblas_int k2, const rocblas_int *ipiv, const rocblas_int incx)

rocblas_status **rocsolver_slaswp**(rocblas_handle handle, const rocblas_int n, float *A, const rocblas_int lda, const rocblas_int k1, const rocblas_int k2, const rocblas_int *ipiv, const rocblas_int incx)

LASWP performs a series of row interchanges on the matrix A.

It interchanges row I with row IPIV[k1 + (I - k1) * abs(inx)], for each of rows K1 through K2 of A. k1 and k2 are 1-based indices.

Parameters

- **handle** – **[in]** rocblas_handle
- **n** – **[in]** rocblas_int. n >= 0.

  The number of columns of the matrix A.

- **A** – **[inout]** pointer to type. Array on the GPU of dimension lda*n.

  On entry, the matrix of column dimension n to which the row interchanges will be applied. On exit, the permuted matrix.

- **lda** – **[in]** rocblas_int. lda > 0.

  The leading dimension of the array A.

- **k1** – **[in]** rocblas_int. k1 > 0.

  The first element of IPIV for which a row interchange will be done. This is a 1-based index.

- **k2** – **[in]** rocblas_int. k2 > k1 > 0.

  (K2-K1+1) is the number of elements of IPIV for which a row interchange will be done. This is a 1-based index.

- **ipiv** – **[in]** pointer to rocblas_int. Array on the GPU of dimension at least k1 + (k2 - k1) * abs(incx).

  The vector of pivot indices. Only the elements in positions k1 through (k1 + (k2 - k1) * abs(incx)) of IPIV are accessed. Elements of ipiv are considered 1-based.

- **incx** – **[in]** rocblas_int. incx != 0.

  The increment between successive values of IPIV. If IPIV is negative, the pivots are applied in reverse order.

2.6.2.3 Householder reflexions

rocsolver_<type>larfg()

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

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

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

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

LARFG generates an orthogonal Householder reflector H of order n.

Householder reflector H is such that

```
H * [alpha] = [beta]
    [  x  ]   [  0 ]
```

where x is an n-1 vector and alpha and beta are scalars. Matrix H can be generated as

```
H = I - tau * [1] * [1 v']
              [v]
```

with v an n-1 vector and tau a scalar.

Parameters

- **handle** – **[in]** rocblas_handle
- **n** – **[in]** rocblas_int. n >= 0.

  The order (size) of reflector H.

- **alpha** – **[inout]** pointer to type. A scalar on the GPU.

  On input the scalar alpha, on output it is overwritten with beta.

- **x** – **[inout]** pointer to type. Array on the GPU of size at least n-1.

  On input it is the vector x, on output it is overwritten with vector v.

- **incx** – **[in]** rocblas_int. incx > 0.

  The increment between consecutive elements of x.

- **tau** – **[out]** pointer to type. A scalar on the GPU.

  The scalar tau.

rocsolver_<type>larft()

rocblas_status **rocsolver_zlarft**(rocblas_handle handle, const *rocblas_direct* direct, const *rocblas_storev* storev, const rocblas_int n, const rocblas_int k, rocblas_double_complex *V, const rocblas_int ldv, rocblas_double_complex *tau, rocblas_double_complex *T, const rocblas_int ldt)

rocblas_status **rocsolver_clarft**(rocblas_handle handle, const *rocblas_direct* direct, const *rocblas_storev* storev, const rocblas_int n, const rocblas_int k, rocblas_float_complex *V, const rocblas_int ldv, rocblas_float_complex *tau, rocblas_float_complex *T, const rocblas_int ldt)

rocblas_status **rocsolver_dlarft**(rocblas_handle handle, const *rocblas_direct* direct, const *rocblas_storev* storev, const rocblas_int n, const rocblas_int k, double *V, const rocblas_int ldv, double *tau, double *T, const rocblas_int ldt)

rocblas_status **rocsolver_slarft**(rocblas_handle handle, const *rocblas_direct* direct, const *rocblas_storev* storev, const rocblas_int n, const rocblas_int k, float *V, const rocblas_int ldv, float *tau, float *T, const rocblas_int ldt)

LARFT Generates the triangular factor T of a block reflector H of order n.

The block reflector H is defined as the product of k Householder matrices as

```
H = H(1) * H(2) * ... * H(k)  (forward direction), or
H = H(k) * ... * H(2) * H(1)  (backward direction)
```

depending on the value of direct.

The triangular matrix T is upper triangular in forward direction and lower triangular in backward direction. If storev is column-wise, then

```
H = I - V * T * V'
```

where the i-th column of matrix V contains the Householder vector associated to H(i). If storev is row-wise, then

```
H = I - V' * T * V
```

where the i-th row of matrix V contains the Householder vector associated to H(i).

Parameters

- **handle** – **[in]** rocblas_handle.
- **direct** – **[in]** *rocblas_direct* .

  Specifies the direction in which the Householder matrices are applied.

- **storev** – **[in]** *rocblas_storev* .

  Specifies how the Householder vectors are stored in matrix V.

- **n** – **[in]** rocblas_int. n >= 0.

  The order (size) of the block reflector.

- **k** – **[in]** rocblas_int. k >= 1.

  The number of Householder matrices.

- **V** – **[in]** pointer to type. Array on the GPU of size ldv*k if column-wise, or ldv*n if row-wise.

  The matrix of Householder vectors.

- **ldv** – **[in]** rocblas_int. ldv >= n if column-wise, or ldv >= k if row-wise.

  Leading dimension of V.

- **tau** – **[in]** pointer to type. Array of k scalars on the GPU.

  The vector of all the scalars associated to the Householder matrices.

- **T** – **[out]** pointer to type. Array on the GPU of dimension ldt*k.

  The triangular factor. T is upper triangular is forward operation, otherwise it is lower triangular. The rest of the array is not used.

- **ldt** – **[in]** rocblas_int. ldt >= k.

  The leading dimension of T.

rocsolver_<type>larf()

rocblas_status **rocsolver_zlarf**(rocblas_handle handle, const rocblas_side side, const rocblas_int m, const rocblas_int n, rocblas_double_complex *x, const rocblas_int incx, const rocblas_double_complex *alpha, rocblas_double_complex *A, const rocblas_int lda)

rocblas_status **rocsolver_clarf**(rocblas_handle handle, const rocblas_side side, const rocblas_int m, const rocblas_int n, rocblas_float_complex *x, const rocblas_int incx, const rocblas_float_complex *alpha, rocblas_float_complex *A, const rocblas_int lda)

rocblas_status **rocsolver_dlarf**(rocblas_handle handle, const rocblas_side side, const rocblas_int m, const rocblas_int n, double *x, const rocblas_int incx, const double *alpha, double *A, const rocblas_int lda)

rocblas_status **rocsolver_slarf**(rocblas_handle handle, const rocblas_side side, const rocblas_int m, const rocblas_int n, float *x, const rocblas_int incx, const float *alpha, float *A, const rocblas_int lda)

LARF applies a Householder reflector H to a general matrix A.

The Householder reflector H, of order m (or n), is to be applied to a m-by-n matrix A from the left (or the right). H is given by

```
H = I - alpha * x * x'
```

where alpha is a scalar and x a Householder vector. H is never actually computed.

Parameters

- **handle** – **[in]** rocblas_handle.
- **side** – **[in]** rocblas_side.

  If side = rocblas_side_left, then compute H*A If side = rocblas_side_right, then compute A*H
- **m** – **[in]** rocblas_int. m >= 0.

  Number of rows of A.
- **n** – **[in]** rocblas_int. n >= 0.

  Number of columns of A.
- **x** – **[in]** pointer to type. Array on the GPU of size at least (1 + (m-1)*abs(incx)) if left side, or at least (1 + (n-1)*abs(incx)) if right side.

  The Householder vector x.
- **incx** – **[in]** rocblas_int. incx != 0.

  Increment between to consecutive elements of x. If incx < 0, the elements of x are used in reverse order.
- **alpha** – **[in]** pointer to type. A scalar on the GPU.

  If alpha = 0, then H = I (A will remain the same, x is never used)
- **A** – **[inout]** pointer to type. Array on the GPU of size lda*n.

  On input, the matrix A. On output it is overwritten with H*A (or A*H).
- **lda** – **[in]** rocblas_int. lda >= m.

  Leading dimension of A.

rocsolver_<type>larfb()

rocblas_status **rocsolver_zlarfb**(rocblas_handle handle, const rocblas_side side, const rocblas_operation trans, const *rocblas_direct* direct, const *rocblas_storev* storev, const rocblas_int m, const rocblas_int n, const rocblas_int k, rocblas_double_complex *V, const rocblas_int ldv, rocblas_double_complex *T, const rocblas_int ldt, rocblas_double_complex *A, const rocblas_int lda)

rocblas_status **rocsolver_clarfb**(rocblas_handle handle, const rocblas_side side, const rocblas_operation trans, const *rocblas_direct* direct, const *rocblas_storev* storev, const rocblas_int m, const rocblas_int n, const rocblas_int k, rocblas_float_complex *V, const rocblas_int ldv, rocblas_float_complex *T, const rocblas_int ldt, rocblas_float_complex *A, const rocblas_int lda)

rocblas_status **rocsolver_dlarfb**(rocblas_handle handle, const rocblas_side side, const rocblas_operation trans, const *rocblas_direct* direct, const *rocblas_storev* storev, const rocblas_int m, const rocblas_int n, const rocblas_int k, double *V, const rocblas_int ldv, double *T, const rocblas_int ldt, double *A, const rocblas_int lda)

rocblas_status **rocsolver_slarfb**(rocblas_handle handle, const rocblas_side side, const rocblas_operation trans, const *rocblas_direct* direct, const *rocblas_storev* storev, const rocblas_int m, const rocblas_int n, const rocblas_int k, float *V, const rocblas_int ldv, float *T, const rocblas_int ldt, float *A, const rocblas_int lda)

LARFB applies a block reflector H to a general m-by-n matrix A.

The block reflector H is applied in one of the following forms, depending on the values of side and trans:

```
H  * A  (No transpose from the left)
H' * A  (Transpose or conjugate transpose from the left)
A * H   (No transpose from the right), and
A * H'  (Transpose or conjugate transpose from the right)
```

The block reflector H is defined as the product of k Householder matrices as

```
H = H(1) * H(2) * ... * H(k)  (forward direction), or
H = H(k) * ... * H(2) * H(1)  (backward direction)
```

depending on the value of direct. H is never stored. It is calculated as

```
H = I - V * T * V'
```

where the i-th column of matrix V contains the Householder vector associated with H(i), if storev is column-wise; or

```
H = I - V' * T * V
```

where the i-th row of matrix V contains the Householder vector associated with H(i), if storev is row-wise. T is the associated triangular factor as computed by LARFT.

Parameters

- **handle** – **[in]** rocblas_handle.
- **side** – **[in]** rocblas_side.

  Specifies from which side to apply H.
- **trans** – **[in]** rocblas_operation.

  Specifies whether the block reflector or its transpose/conjugate transpose is to be applied.
- **direct** – **[in]** *rocblas_direct* .

  Specifies the direction in which the Householder matrices were to be applied to generate H.
- **storev** – **[in]** *rocblas_storev* .

  Specifies how the Householder vectors are stored in matrix V.
- **m** – **[in]** rocblas_int. m >= 0.

  Number of rows of matrix A.
- **n** – **[in]** rocblas_int. n >= 0.

  Number of columns of matrix A.
- **k** – **[in]** rocblas_int. k >= 1.

  The number of Householder matrices.

- **V** – **[in]** pointer to type. Array on the GPU of size ldv*k if column-wise, ldv*n if row-wise and applying from the right, or ldv*m if row-wise and applying from the left.

  The matrix of Householder vectors.

- **ldv** – **[in]** rocblas_int. ldv >= k if row-wise, ldv >= m if column-wise and applying from the left, or ldv >= n if column-wise and applying from the right.

  Leading dimension of V.

- **T** – **[in]** pointer to type. Array on the GPU of dimension ldt*k.

  The triangular factor of the block reflector.

- **ldt** – **[in]** rocblas_int. ldt >= k.

  The leading dimension of T.

- **A** – **[inout]** pointer to type. Array on the GPU of size lda*n.

  On input, the matrix A. On output it is overwritten with H*A, A*H, H'*A, or A*H'.

- **lda** – **[in]** rocblas_int. lda >= m.

  Leading dimension of A.

2.6.2.4 Bidiagonal forms

rocsolver_<type>labrd()

rocblas_status **rocsolver_zlabrd**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, const rocblas_int k, rocblas_double_complex *A, const rocblas_int lda, double *D, double *E, rocblas_double_complex *tauq, rocblas_double_complex *taup, rocblas_double_complex *X, const rocblas_int ldx, rocblas_double_complex *Y, const rocblas_int ldy)

rocblas_status **rocsolver_clabrd**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, const rocblas_int k, rocblas_float_complex *A, const rocblas_int lda, float *D, float *E, rocblas_float_complex *tauq, rocblas_float_complex *taup, rocblas_float_complex *X, const rocblas_int ldx, rocblas_float_complex *Y, const rocblas_int ldy)

rocblas_status **rocsolver_dlabrd**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, const rocblas_int k, double *A, const rocblas_int lda, double *D, double *E, double *tauq, double *taup, double *X, const rocblas_int ldx, double *Y, const rocblas_int ldy)

rocblas_status **rocsolver_slabrd**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, const rocblas_int k, float *A, const rocblas_int lda, float *D, float *E, float *tauq, float *taup, float *X, const rocblas_int ldx, float *Y, const rocblas_int ldy)

LABRD computes the bidiagonal form of the first k rows and columns of a general m-by-n matrix A, as well as the matrices X and Y needed to reduce the remaining part of A.

The bidiagonal form is given by:

```
B = Q' * A * P
```

where B is upper bidiagonal if m >= n and lower bidiagonal if m < n, and Q and P are orthogonal/unitary matrices represented as the product of Householder matrices

```
Q = H(1) * H(2) * ... *  H(k)  and P = G(1) * G(2) * ... * G(k-1), if m >= n, or
Q = H(1) * H(2) * ... * H(k-1) and P = G(1) * G(2) * ... *  G(k),  if m < n
```

Each Householder matrix H(i) and G(i) is given by

```
H(i) = I - tauq[i-1] * v(i) * v(i)', and
G(i) = I - taup[i-1] * u(i) * u(i)'
```

If m >= n, the first i-1 elements of the Householder vector v(i) are zero, and v(i)[i] = 1; while the first i elements of the Householder vector u(i) are zero, and u(i)[i+1] = 1. If m < n, the first i elements of the Householder vector v(i) are zero, and v(i)[i+1] = 1; while the first i-1 elements of the Householder vector u(i) are zero, and u(i)[i] = 1.

The unreduced part of the matrix A can be updated using a block update:

```
A = A - V * Y' - X * U'
```

where V is an m-by-k matrix and U is an n-by-k formed using the vectors v and u.

Parameters

- **handle** – **[in]** rocblas_handle.
- **m** – **[in]** rocblas_int. m >= 0.

  The number of rows of the matrix A.
- **n** – **[in]** rocblas_int. n >= 0.

  The number of columns of the matrix A.
- **k** – **[in]** rocblas_int. min(m,n) >= k >= 0.

  The number of leading rows and columns of the matrix A to be reduced.
- **A** – **[inout]** pointer to type. Array on the GPU of dimension lda*n.

  On entry, the m-by-n matrix to be factored. On exit, the elements on the diagonal and superdiagonal (if m >= n), or subdiagonal (if m < n) contain the bidiagonal form B. If m >= n, the elements below the diagonal are the m - i elements of vector v(i) for i = 1,2,...,n, and the elements above the superdiagonal are the n - i - 1 elements of vector u(i) for i = 1,2,...,n-1. If m < n, the elements below the subdiagonal are the m - i - 1 elements of vector v(i) for i = 1,2,...,m-1, and the elements above the diagonal are the n - i elements of vector u(i) for i = 1,2,...,m.
- **lda** – **[in]** rocblas_int. lda >= m.

  specifies the leading dimension of A.
- **D** – **[out]** pointer to real type. Array on the GPU of dimension k.

  The diagonal elements of B.
- **E** – **[out]** pointer to real type. Array on the GPU of dimension k.

  The off-diagonal elements of B.
- **tauq** – **[out]** pointer to type. Array on the GPU of dimension k.

  The scalar factors of the Householder matrices H(i).

- **taup** – **[out]** pointer to type. Array on the GPU of dimension k.

  The scalar factors of the Householder matrices G(i).

- **X** – **[out]** pointer to type. Array on the GPU of dimension ldx*k.

  The m-by-k matrix needed to reduce the unreduced part of A.

- **ldx** – **[in]** rocblas_int. ldx >= m.

  specifies the leading dimension of X.

- **Y** – **[out]** pointer to type. Array on the GPU of dimension ldy*k.

  The n-by-k matrix needed to reduce the unreduced part of A.

- **ldy** – **[in]** rocblas_int. ldy >= n.

  specifies the leading dimension of Y.

rocsolver_<type>bdsqr()

rocblas_status **rocsolver_zbdsqr**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, const rocblas_int nv, const rocblas_int nu, const rocblas_int nc, double *D, double *E, rocblas_double_complex *V, const rocblas_int ldv, rocblas_double_complex *U, const rocblas_int ldu, rocblas_double_complex *C, const rocblas_int ldc, rocblas_int *info)

rocblas_status **rocsolver_cbdsqr**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, const rocblas_int nv, const rocblas_int nu, const rocblas_int nc, float *D, float *E, rocblas_float_complex *V, const rocblas_int ldv, rocblas_float_complex *U, const rocblas_int ldu, rocblas_float_complex *C, const rocblas_int ldc, rocblas_int *info)

rocblas_status **rocsolver_dbdsqr**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, const rocblas_int nv, const rocblas_int nu, const rocblas_int nc, double *D, double *E, double *V, const rocblas_int ldv, double *U, const rocblas_int ldu, double *C, const rocblas_int ldc, rocblas_int *info)

rocblas_status **rocsolver_sbdsqr**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, const rocblas_int nv, const rocblas_int nu, const rocblas_int nc, float *D, float *E, float *V, const rocblas_int ldv, float *U, const rocblas_int ldu, float *C, const rocblas_int ldc, rocblas_int *info)

BDSQR computes the singular value decomposition (SVD) of a n-by-n bidiagonal matrix B.

The SVD of B has the form:

```
B = Ub * S * Vb'
```

where S is the n-by-n diagonal matrix of singular values of B, the columns of Ub are the left singular vectors of B, and the columns of Vb are its right singular vectors.

The computation of the singular vectors is optional; this function accepts input matrices U (of size nu-by-n) and V (of size n-by-nv) that are overwritten with U*Ub and Vb'*V. If nu = 0 no left vectors are computed; if nv = 0 no right vectors are computed.

Optionally, this function can also compute Ub'*C for a given n-by-nc input matrix C.

Parameters

- **handle** – **[in]** rocblas_handle.
- **uplo** – **[in]** rocblas_fill.

  Specifies whether B is upper or lower bidiagonal.
- **n** – **[in]** rocblas_int. n >= 0.

  The number of rows and columns of matrix B.
- **nv** – **[in]** rocblas_int. nv >= 0.

  The number of columns of matrix V.
- **nu** – **[in]** rocblas_int. nu >= 0.

  The number of rows of matrix U.
- **nc** – **[in]** rocblas_int. nu >= 0.

  The number of columns of matrix C.
- **D** – **[inout]** pointer to real type. Array on the GPU of dimension n.

  On entry, the diagonal elements of B. On exit, if info = 0, the singular values of B in decreasing order; if info > 0, the diagonal elements of a bidiagonal matrix orthogonally equivalent to B.
- **E** – **[inout]** pointer to real type. Array on the GPU of dimension n-1.

  On entry, the off-diagonal elements of B. On exit, if info > 0, the off-diagonal elements of a bidiagonal matrix orthogonally equivalent to B (if info = 0 this matrix converges to zero).
- **V** – **[inout]** pointer to type. Array on the GPU of dimension ldv*nv.

  On entry, the matrix V. On exit, it is overwritten with Vb'*V. (Not referenced if nv = 0).
- **ldv** – **[in]** rocblas_int. ldv >= n if nv > 0, or ldv >=1 if nv = 0.

  Specifies the leading dimension of V.
- **U** – **[inout]** pointer to type. Array on the GPU of dimension ldu*n.

  On entry, the matrix U. On exit, it is overwritten with U*Ub. (Not referenced if nu = 0).
- **ldu** – **[in]** rocblas_int. ldu >= nu.

  Specifies the leading dimension of U.
- **C** – **[inout]** pointer to type. Array on the GPU of dimension ldc*nc.

  On entry, the matrix C. On exit, it is overwritten with Ub'*C. (Not referenced if nc = 0).
- **ldc** – **[in]** rocblas_int. ldc >= n if nc > 0, or ldc >=1 if nc = 0.

  Specifies the leading dimension of C.
- **info** – **[out]** pointer to a rocblas_int on the GPU.

  If info = 0, successful exit. If info = i > 0, i elements of E have not converged to zero.

2.6.2.5 Tridiagonal forms

rocsolver_<type>latrd()

rocblas_status **rocsolver_zlatrd**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, const rocblas_int k, rocblas_double_complex *A, const rocblas_int lda, double *E, rocblas_double_complex *tau, rocblas_double_complex *W, const rocblas_int ldw)

rocblas_status **rocsolver_clatrd**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, const rocblas_int k, rocblas_float_complex *A, const rocblas_int lda, float *E, rocblas_float_complex *tau, rocblas_float_complex *W, const rocblas_int ldw)

rocblas_status **rocsolver_dlatrd**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, const rocblas_int k, double *A, const rocblas_int lda, double *E, double *tau, double *W, const rocblas_int ldw)

rocblas_status **rocsolver_slatrd**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, const rocblas_int k, float *A, const rocblas_int lda, float *E, float *tau, float *W, const rocblas_int ldw)

LATRD computes the tridiagonal form of k rows and columns of a symmetric/hermitian matrix A, as well as the matrix W needed to update the remaining part of A.

The reduced form is given by:

```
T = Q' * A * Q
```

If uplo is lower, the first k rows and columns of T form a tridiagonal block, if uplo is upper, then the last k rows and columns of T form the tridiagonal block. Q is an orthogonal/unitary matrix represented as the product of Householder matrices

```
Q = H(1) * H(2) * ... *  H(k)  if uplo indicates lower, or
Q = H(n-1) * H(n-2) * ... * H(n-k) if uplo is upper.
```

Each Householder matrix H(i) is given by

```
H(i) = I - tau[i] * v(i) * v(i)'
```

where tau[i] is the corresponding Householder scalar. When uplo indicates lower, the first i elements of the Householder vector v(i) are zero, and v(i)[i+1] = 1. If uplo is upper, the last n-i elements of the Householder vector v(i) are zero, and v(i)[i] = 1.

The unreduced part of the matrix A can be updated using a rank update of the form:

```
A = A - V * W' - W * V'
```

where V is an n-by-k matrix formed by the vectors v(i).

Parameters

- **handle** – **[in]** rocblas_handle.
- **uplo** – **[in]** rocblas_fill.

  Specifies whether the upper or lower part of the matrix A is stored. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.

- **n** – **[in]** rocblas_int. n >= 0.

  The number of rows and columns of the matrix A.

- **k** – **[in]** rocblas_int. 0 <= k <= n.

  The number of rows and columns of the matrix A to be reduced.

- **A** – **[inout]** pointer to type. Array on the GPU of dimension lda*n.

  On entry, the n-by-n matrix to be reduced. On exit, if uplo is lower, the first k columns have been reduced to tridiagonal form (given in the diagonal elements of A and the array E), the elements below the diagonal contain the vectors v(i) stored as columns. If uplo is upper, the last k columns have been reduced to tridiagonal form (given in the diagonal elements of A and the array E), the elements above the diagonal contain the vectors v(i) stored as columns.

- **lda** – **[in]** rocblas_int. lda >= n.

  specifies the leading dimension of A.

- **E** – **[out]** pointer to real type. Array on the GPU of dimension n-1.

  If upper (lower), the last (first) k elements of E are the off-diagonal elements of the computed tridiagonal block.

- **tau** – **[out]** pointer to type. Array on the GPU of dimension n-1.

  If upper (lower), the last (first) k elements of tau are the scalar factors of the Householder matrices H(i).

- **W** – **[out]** pointer to type. Array on the GPU of dimension ldw*k.

  The n-by-k matrix needed to update the unreduced part of A.

- **ldw** – **[in]** rocblas_int. ldw >= n.

  specifies the leading dimension of W.

2.6.2.6 Orthonormal matrices

rocsolver_<type>org2r()

rocblas_status **rocsolver_dorg2r**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, const rocblas_int k, double *A, const rocblas_int lda, double *ipiv)

rocblas_status **rocsolver_sorg2r**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, const rocblas_int k, float *A, const rocblas_int lda, float *ipiv)

ORG2R generates a m-by-n Matrix Q with orthonormal columns.

(This is the unblocked version of the algorithm).

The matrix Q is defined as the first n columns of the product of k Householder reflectors of order m

```
Q = H(1) * H(2) * ... * H(k)
```

Householder matrices H(i) are never stored, they are computed from its corresponding Householder vector v(i) and scalar ipiv_i as returned by GEQRF.

Parameters

- **handle** – **[in]** rocblas_handle.

- **m** – **[in]** rocblas_int. m >= 0.

  The number of rows of the matrix Q.

- **n** – **[in]** rocblas_int. 0 <= n <= m.

  The number of columns of the matrix Q.

- **k** – **[in]** rocblas_int. 0 <= k <= n.

  The number of Householder reflectors.

- **A** – **[inout]** pointer to type. Array on the GPU of dimension lda*n.

  On entry, the i-th column has Householder vector v(i), for i = 1,2,...,k as returned in the first k columns of matrix A of GEQRF. On exit, the computed matrix Q.

- **lda** – **[in]** rocblas_int. lda >= m.

  Specifies the leading dimension of A.

- **ipiv** – **[in]** pointer to type. Array on the GPU of dimension at least k.

  The scalar factors of the Householder matrices H(i) as returned by GEQRF.

rocsolver_<type>orgqr()

rocblas_status **rocsolver_dorgqr**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, const rocblas_int k, double *A, const rocblas_int lda, double *ipiv)

rocblas_status **rocsolver_sorgqr**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, const rocblas_int k, float *A, const rocblas_int lda, float *ipiv)

ORGQR generates a m-by-n Matrix Q with orthonormal columns.

(This is the blocked version of the algorithm).

The matrix Q is defined as the first n columns of the product of k Householder reflectors of order m

```
Q = H(1) * H(2) * ... * H(k)
```

Householder matrices H(i) are never stored, they are computed from its corresponding Householder vector v(i) and scalar ipiv_i as returned by GEQRF.

Parameters

- **handle** – **[in]** rocblas_handle.

- **m** – **[in]** rocblas_int. m >= 0.

  The number of rows of the matrix Q.

- **n** – **[in]** rocblas_int. 0 <= n <= m.

  The number of columns of the matrix Q.

- **k** – **[in]** rocblas_int. 0 <= k <= n.

  The number of Householder reflectors.

- **A** – **[inout]** pointer to type. Array on the GPU of dimension lda*n.

  On entry, the i-th column has Householder vector v(i), for i = 1,2,...,k as returned in the first k columns of matrix A of GEQRF. On exit, the computed matrix Q.

- **lda** – **[in]** rocblas_int. lda >= m.

  Specifies the leading dimension of A.

- **ipiv** – **[in]** pointer to type. Array on the GPU of dimension at least k.

  The scalar factors of the Householder matrices H(i) as returned by GEQRF.

rocsolver_<type>orgl2()

rocblas_status **rocsolver_dorgl2**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, const rocblas_int k, double *A, const rocblas_int lda, double *ipiv)

rocblas_status **rocsolver_sorgl2**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, const rocblas_int k, float *A, const rocblas_int lda, float *ipiv)

ORGL2 generates a m-by-n Matrix Q with orthonormal rows.

(This is the unblocked version of the algorithm).

The matrix Q is defined as the first m rows of the product of k Householder reflectors of order n

```
Q = H(k) * H(k-1) * ... * H(1)
```

Householder matrices H(i) are never stored, they are computed from its corresponding Householder vector v(i) and scalar ipiv_i as returned by GELQF.

Parameters

- **handle** – **[in]** rocblas_handle.
- **m** – **[in]** rocblas_int. 0 <= m <= n.

  The number of rows of the matrix Q.

- **n** – **[in]** rocblas_int. n >= 0.

  The number of columns of the matrix Q.

- **k** – **[in]** rocblas_int. 0 <= k <= m.

  The number of Householder reflectors.

- **A** – **[inout]** pointer to type. Array on the GPU of dimension lda*n.

  On entry, the i-th row has Householder vector v(i), for i = 1,2,...,k as returned in the first k rows of matrix A of GELQF. On exit, the computed matrix Q.

- **lda** – **[in]** rocblas_int. lda >= m.

  Specifies the leading dimension of A.

- **ipiv** – **[in]** pointer to type. Array on the GPU of dimension at least k.

  The scalar factors of the Householder matrices H(i) as returned by GELQF.

rocsolver_<type>orglq()

rocblas_status **rocsolver_dorglq**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, const rocblas_int k, double *A, const rocblas_int lda, double *ipiv)

rocblas_status **rocsolver_sorglq**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, const rocblas_int k, float *A, const rocblas_int lda, float *ipiv)

ORGLQ generates a m-by-n Matrix Q with orthonormal rows.

(This is the blocked version of the algorithm).

The matrix Q is defined as the first m rows of the product of k Householder reflectors of order n

```
Q = H(k) * H(k-1) * ... * H(1)
```

Householder matrices H(i) are never stored, they are computed from its corresponding Householder vector v(i) and scalar ipiv_i as returned by GELQF.

Parameters

- **handle** – **[in]** rocblas_handle.
- **m** – **[in]** rocblas_int. 0 <= m <= n.

  The number of rows of the matrix Q.
- **n** – **[in]** rocblas_int. n >= 0.

  The number of columns of the matrix Q.
- **k** – **[in]** rocblas_int. 0 <= k <= m.

  The number of Householder reflectors.
- **A** – **[inout]** pointer to type. Array on the GPU of dimension lda*n.

  On entry, the i-th row has Householder vector v(i), for i = 1,2,...,k as returned in the first k rows of matrix A of GELQF. On exit, the computed matrix Q.
- **lda** – **[in]** rocblas_int. lda >= m.

  Specifies the leading dimension of A.
- **ipiv** – **[in]** pointer to type. Array on the GPU of dimension at least k.

  The scalar factors of the Householder matrices H(i) as returned by GELQF.

rocsolver_<type>org2l()

rocblas_status **rocsolver_dorg2l**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, const rocblas_int k, double *A, const rocblas_int lda, double *ipiv)

rocblas_status **rocsolver_sorg2l**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, const rocblas_int k, float *A, const rocblas_int lda, float *ipiv)

ORG2L generates a m-by-n Matrix Q with orthonormal columns.

(This is the unblocked version of the algorithm).

The matrix Q is defined as the last n columns of the product of k Householder reflectors of order m

```
Q = H(k) * H(k-1) * ... * H(1)
```

Householder matrices H(i) are never stored, they are computed from its corresponding Householder vector v(i) and scalar ipiv_i as returned by GEQLF.

Parameters

- **handle** – **[in]** rocblas_handle.
- **m** – **[in]** rocblas_int. m >= 0.

  The number of rows of the matrix Q.
- **n** – **[in]** rocblas_int. 0 <= n <= m.

  The number of columns of the matrix Q.
- **k** – **[in]** rocblas_int. 0 <= k <= n.

  The number of Householder reflectors.
- **A** – **[inout]** pointer to type. Array on the GPU of dimension lda*n.

  On entry, the (n-k+i)-th column has Householder vector v(i), for i = 1,2,...,k as returned in the last k columns of matrix A of GEQLF. On exit, the computed matrix Q.
- **lda** – **[in]** rocblas_int. lda >= m.

  Specifies the leading dimension of A.
- **ipiv** – **[in]** pointer to type. Array on the GPU of dimension at least k.

  The scalar factors of the Householder matrices H(i) as returned by GEQLF.

rocsolver_<type>orgql()

rocblas_status **rocsolver_dorgql**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, const rocblas_int k, double *A, const rocblas_int lda, double *ipiv)

rocblas_status **rocsolver_sorgql**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, const rocblas_int k, float *A, const rocblas_int lda, float *ipiv)

ORGQL generates a m-by-n Matrix Q with orthonormal columns.

(This is the blocked version of the algorithm).

The matrix Q is defined as the last n column of the product of k Householder reflectors of order m

```
Q = H(k) * H(k-1) * ... * H(1)
```

Householder matrices H(i) are never stored, they are computed from its corresponding Householder vector v(i) and scalar ipiv_i as returned by GEQLF.

Parameters

- **handle** – **[in]** rocblas_handle.
- **m** – **[in]** rocblas_int. m >= 0.

  The number of rows of the matrix Q.
- **n** – **[in]** rocblas_int. 0 <= n <= m.

  The number of columns of the matrix Q.

- **k** – **[in]** rocblas_int. 0 <= k <= n.

  The number of Householder reflectors.

- **A** – **[inout]** pointer to type. Array on the GPU of dimension lda*n.

  On entry, the (n-k+i)-th column has Householder vector v(i), for i = 1,2,...,k as returned in the last k columns of matrix A of GEQLF. On exit, the computed matrix Q.

- **lda** – **[in]** rocblas_int. lda >= m.

  Specifies the leading dimension of A.

- **ipiv** – **[in]** pointer to type. Array on the GPU of dimension at least k.

  The scalar factors of the Householder matrices H(i) as returned by GEQLF.

rocsolver_<type>orgbr()

rocblas_status **rocsolver_dorgbr**(rocblas_handle handle, const *rocblas_storev* storev, const rocblas_int m, const rocblas_int n, const rocblas_int k, double *A, const rocblas_int lda, double *ipiv)

rocblas_status **rocsolver_sorgbr**(rocblas_handle handle, const *rocblas_storev* storev, const rocblas_int m, const rocblas_int n, const rocblas_int k, float *A, const rocblas_int lda, float *ipiv)

ORGBR generates a m-by-n Matrix Q with orthonormal rows or columns.

If storev is column-wise, then the matrix Q has orthonormal columns. If m >= k, Q is defined as the first n columns of the product of k Householder reflectors of order m

```
Q = H(1) * H(2) * ... * H(k)
```

If m < k, Q is defined as the product of Householder reflectors of order m

```
Q = H(1) * H(2) * ... * H(m-1)
```

On the other hand, if storev is row-wise, then the matrix Q has orthonormal rows. If n > k, Q is defined as the first m rows of the product of k Householder reflectors of order n

```
Q = H(k) * H(k-1) * ... * H(1)
```

If n <= k, Q is defined as the product of Householder reflectors of order n

```
Q = H(n-1) * H(n-2) * ... * H(1)
```

The Householder matrices H(i) are never stored, they are computed from its corresponding Householder vectors v(i) and scalars ipiv_i as returned by GEBRD in its arguments A and tauq or taup.

Parameters

- **handle** – **[in]** rocblas_handle.
- **storev** – **[in]** *rocblas_storev* .

  Specifies whether to work column-wise or row-wise.

- **m** – **[in]** rocblas_int. m >= 0.

  The number of rows of the matrix Q. If row-wise, then min(n,k) <= m <= n.

- **n** – **[in]** rocblas_int. n >= 0.

  The number of columns of the matrix Q. If column-wise, then min(m,k) <= n <= m.

- **k** – **[in]** rocblas_int. k >= 0.

  The number of columns (if storev is colum-wise) or rows (if row-wise) of the original matrix reduced by GEBRD.

- **A** – **[inout]** pointer to type. Array on the GPU of dimension lda*n.

  On entry, the i-th column (or row) has the Householder vector v(i) as returned by GEBRD. On exit, the computed matrix Q.

- **lda** – **[in]** rocblas_int. lda >= m.

  Specifies the leading dimension of A.

- **ipiv** – **[in]** pointer to type. Array on the GPU of dimension min(m,k) if column-wise, or min(n,k) if row-wise.

  The scalar factors of the Householder matrices H(i) as returned by GEBRD.

rocsolver_<type>orgtr()

rocblas_status **rocsolver_dorgtr**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, double *A, const rocblas_int lda, double *ipiv)

rocblas_status **rocsolver_sorgtr**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, float *A, const rocblas_int lda, float *ipiv)

ORGTR generates a n-by-n orthogonal Matrix Q.

Q is defined as the product of n-1 Householder reflectors of order n. If uplo indicates upper, then Q has the form

```
Q = H(n-1) * H(n-2) * ... * H(1)
```

On the other hand, if uplo indicates lower, then Q has the form

```
Q = H(1) * H(2) * ... * H(n-1)
```

The Householder matrices H(i) are never stored, they are computed from its corresponding Householder vectors v(i) and scalars ipiv_i as returned by SYTRD in its arguments A and tau.

Parameters

- **handle** – **[in]** rocblas_handle.
- **uplo** – **[in]** rocblas_fill.

  Specifies whether the SYTRD factorization was upper or lower triangular. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.

- **n** – **[in]** rocblas_int. n >= 0.

  The number of rows and columns of the matrix Q.

- **A** – **[inout]** pointer to type. Array on the GPU of dimension lda*n.

  On entry, the (i+1)-th column (if uplo indicates upper) or i-th column (if uplo indicates lower) has the Householder vector v(i) as returned by SYTRD. On exit, the computed matrix Q.

- **lda** – **[in]** rocblas_int. lda >= m.

  Specifies the leading dimension of A.

- **ipiv** – **[in]** pointer to type. Array on the GPU of dimension n-1.

  The scalar factors of the Householder matrices H(i) as returned by SYTRD.

rocsolver_<type>orm2r()

rocblas_status **rocsolver_dorm2r**(rocblas_handle handle, const rocblas_side side, const rocblas_operation trans, const rocblas_int m, const rocblas_int n, const rocblas_int k, double *A, const rocblas_int lda, double *ipiv, double *C, const rocblas_int ldc)

rocblas_status **rocsolver_sorm2r**(rocblas_handle handle, const rocblas_side side, const rocblas_operation trans, const rocblas_int m, const rocblas_int n, const rocblas_int k, float *A, const rocblas_int lda, float *ipiv, float *C, const rocblas_int ldc)

ORM2R applies a matrix Q with orthonormal columns to a general m-by-n matrix C.

(This is the unblocked version of the algorithm).

The matrix Q is applied in one of the following forms, depending on the values of side and trans:

```
Q  * C  (No transpose from the left)
Q' * C  (Transpose from the left)
C * Q   (No transpose from the right), and
C * Q'  (Transpose from the right)
```

Q is an orthogonal matrix defined as the product of k Householder reflectors as

```
Q = H(1) * H(2) * ... * H(k)
```

of order m if applying from the left, or n if applying from the right. Q is never stored, it is calculated from the Householder vectors and scalars returned by the QR factorization GEQRF.

Parameters

- **handle** – **[in]** rocblas_handle.
- **side** – **[in]** rocblas_side.

  Specifies from which side to apply Q.

- **trans** – **[in]** rocblas_operation.

  Specifies whether the matrix Q or its transpose is to be applied.

- **m** – **[in]** rocblas_int. m >= 0.

  Number of rows of matrix C.

- **n** – **[in]** rocblas_int. n >= 0.

  Number of columns of matrix C.

- **k** – **[in]** rocblas_int. k >= 0; k <= m if side is left, k <= n if side is right.

  The number of Householder reflectors that form Q.

- **A** – **[in]** pointer to type. Array on the GPU of size lda*k.

The i-th column has the Householder vector v(i) associated with H(i) as returned by GEQRF in the first k columns of its argument A.

- **lda** – **[in]** rocblas_int. lda >= m if side is left, or lda >= n if side is right.

  Leading dimension of A.

- **ipiv** – **[in]** pointer to type. Array on the GPU of dimension at least k.

  The scalar factors of the Householder matrices H(i) as returned by GEQRF.

- **C** – **[inout]** pointer to type. Array on the GPU of size ldc*n.

  On input, the matrix C. On output it is overwritten with Q*C, C*Q, Q'*C, or C*Q'.

- **ldc** – **[in]** rocblas_int. ldc >= m.

  Leading dimension of C.

rocsolver_<type>ormqr()

rocblas_status **rocsolver_dormqr**(rocblas_handle handle, const rocblas_side side, const rocblas_operation trans, const rocblas_int m, const rocblas_int n, const rocblas_int k, double *A, const rocblas_int lda, double *ipiv, double *C, const rocblas_int ldc)

rocblas_status **rocsolver_sormqr**(rocblas_handle handle, const rocblas_side side, const rocblas_operation trans, const rocblas_int m, const rocblas_int n, const rocblas_int k, float *A, const rocblas_int lda, float *ipiv, float *C, const rocblas_int ldc)

ORMQR applies a matrix Q with orthonormal columns to a general m-by-n matrix C.

(This is the blocked version of the algorithm).

The matrix Q is applied in one of the following forms, depending on the values of side and trans:

```
Q  * C  (No transpose from the left)
Q' * C  (Transpose from the left)
C * Q   (No transpose from the right), and
C * Q'  (Transpose from the right)
```

Q is an orthogonal matrix defined as the product of k Householder reflectors as

```
Q = H(1) * H(2) * ... * H(k)
```

of order m if applying from the left, or n if applying from the right. Q is never stored, it is calculated from the Householder vectors and scalars returned by the QR factorization GEQRF.

Parameters

- **handle** – **[in]** rocblas_handle.
- **side** – **[in]** rocblas_side.

  Specifies from which side to apply Q.

- **trans** – **[in]** rocblas_operation.

  Specifies whether the matrix Q or its transpose is to be applied.

- **m** – **[in]** rocblas_int. m >= 0.

  Number of rows of matrix C.

- **n** – **[in]** rocblas_int. n >= 0.

  Number of columns of matrix C.

- **k** – **[in]** rocblas_int. k >= 0; k <= m if side is left, k <= n if side is right.

  The number of Householder reflectors that form Q.

- **A** – **[in]** pointer to type. Array on the GPU of size lda*k.

  The i-th column has the Householder vector v(i) associated with H(i) as returned by GEQRF in the first k columns of its argument A.

- **lda** – **[in]** rocblas_int. lda >= m if side is left, or lda >= n if side is right.

  Leading dimension of A.

- **ipiv** – **[in]** pointer to type. Array on the GPU of dimension at least k.

  The scalar factors of the Householder matrices H(i) as returned by GEQRF.

- **C** – **[inout]** pointer to type. Array on the GPU of size ldc*n.

  On input, the matrix C. On output it is overwritten with Q*C, C*Q, Q'*C, or C*Q'.

- **ldc** – **[in]** rocblas_int. ldc >= m.

  Leading dimension of C.

rocsolver_<type>orml2()

rocblas_status **rocsolver_dorml2**(rocblas_handle handle, const rocblas_side side, const rocblas_operation trans, const rocblas_int m, const rocblas_int n, const rocblas_int k, double *A, const rocblas_int lda, double *ipiv, double *C, const rocblas_int ldc)

rocblas_status **rocsolver_sorml2**(rocblas_handle handle, const rocblas_side side, const rocblas_operation trans, const rocblas_int m, const rocblas_int n, const rocblas_int k, float *A, const rocblas_int lda, float *ipiv, float *C, const rocblas_int ldc)

ORML2 applies a matrix Q with orthonormal rows to a general m-by-n matrix C.

(This is the unblocked version of the algorithm).

The matrix Q is applied in one of the following forms, depending on the values of side and trans:

```
Q  * C  (No transpose from the left)
Q' * C  (Transpose from the left)
C * Q   (No transpose from the right), and
C * Q'  (Transpose from the right)
```

Q is an orthogonal matrix defined as the product of k Householder reflectors as

```
Q = H(k) * H(k-1) * ... * H(1)
```

of order m if applying from the left, or n if applying from the right. Q is never stored, it is calculated from the Householder vectors and scalars returned by the LQ factorization GELQF.

Parameters

- **handle** – **[in]** rocblas_handle.

- **side** – **[in]** rocblas_side.

  Specifies from which side to apply Q.

- **trans** – **[in]** rocblas_operation.

  Specifies whether the matrix Q or its transpose is to be applied.

- **m** – **[in]** rocblas_int. m >= 0.

  Number of rows of matrix C.

- **n** – **[in]** rocblas_int. n >= 0.

  Number of columns of matrix C.

- **k** – **[in]** rocblas_int. k >= 0; k <= m if side is left, k <= n if side is right.

  The number of Householder reflectors that form Q.

- **A** – **[in]** pointer to type. Array on the GPU of size lda*m if side is left, or lda*n if side is right.

  The i-th row has the Householder vector v(i) associated with H(i) as returned by GELQF in the first k rows of its argument A.

- **lda** – **[in]** rocblas_int. lda >= k.

  Leading dimension of A.

- **ipiv** – **[in]** pointer to type. Array on the GPU of dimension at least k.

  The scalar factors of the Householder matrices H(i) as returned by GELQF.

- **C** – **[inout]** pointer to type. Array on the GPU of size ldc*n.

  On input, the matrix C. On output it is overwritten with Q*C, C*Q, Q'*C, or C*Q'.

- **ldc** – **[in]** rocblas_int. ldc >= m.

  Leading dimension of C.

rocsolver_<type>ormlq()

rocblas_status **rocsolver_dormlq**(rocblas_handle handle, const rocblas_side side, const rocblas_operation trans, const rocblas_int m, const rocblas_int n, const rocblas_int k, double *A, const rocblas_int lda, double *ipiv, double *C, const rocblas_int ldc)

rocblas_status **rocsolver_sormlq**(rocblas_handle handle, const rocblas_side side, const rocblas_operation trans, const rocblas_int m, const rocblas_int n, const rocblas_int k, float *A, const rocblas_int lda, float *ipiv, float *C, const rocblas_int ldc)

ORMLQ applies a matrix Q with orthonormal rows to a general m-by-n matrix C.

(This is the blocked version of the algorithm).

The matrix Q is applied in one of the following forms, depending on the values of side and trans:

```
Q  * C  (No transpose from the left)
Q' * C  (Transpose from the left)
C * Q   (No transpose from the right), and
C * Q'  (Transpose from the right)
```

Q is an orthogonal matrix defined as the product of k Householder reflectors as

```
Q = H(k) * H(k-1) * ... * H(1)
```

of order m if applying from the left, or n if applying from the right. Q is never stored, it is calculated from the Householder vectors and scalars returned by the LQ factorization GELQF.

Parameters

- **handle** – **[in]** rocblas_handle.
- **side** – **[in]** rocblas_side.

  Specifies from which side to apply Q.
- **trans** – **[in]** rocblas_operation.

  Specifies whether the matrix Q or its transpose is to be applied.
- **m** – **[in]** rocblas_int. m >= 0.

  Number of rows of matrix C.
- **n** – **[in]** rocblas_int. n >= 0.

  Number of columns of matrix C.
- **k** – **[in]** rocblas_int. k >= 0; k <= m if side is left, k <= n if side is right.

  The number of Householder reflectors that form Q.
- **A** – **[in]** pointer to type. Array on the GPU of size lda*m if side is left, or lda*n if side is right.

  The i-th row has the Householder vector v(i) associated with H(i) as returned by GELQF in the first k rows of its argument A.
- **lda** – **[in]** rocblas_int. lda >= k.

  Leading dimension of A.
- **ipiv** – **[in]** pointer to type. Array on the GPU of dimension at least k.

  The scalar factors of the Householder matrices H(i) as returned by GELQF.
- **C** – **[inout]** pointer to type. Array on the GPU of size ldc*n.

  On input, the matrix C. On output it is overwritten with Q*C, C*Q, Q'*C, or C*Q'.
- **ldc** – **[in]** rocblas_int. ldc >= m.

  Leading dimension of C.

rocsolver_<type>orm2l()

rocblas_status **rocsolver_dorm2l**(rocblas_handle handle, const rocblas_side side, const rocblas_operation trans, const rocblas_int m, const rocblas_int n, const rocblas_int k, double *A, const rocblas_int lda, double *ipiv, double *C, const rocblas_int ldc)

rocblas_status **rocsolver_sorm2l**(rocblas_handle handle, const rocblas_side side, const rocblas_operation trans, const rocblas_int m, const rocblas_int n, const rocblas_int k, float *A, const rocblas_int lda, float *ipiv, float *C, const rocblas_int ldc)

ORM2L applies a matrix Q with orthonormal columns to a general m-by-n matrix C.

(This is the unblocked version of the algorithm).

The matrix Q is applied in one of the following forms, depending on the values of side and trans:

```
Q  * C  (No transpose from the left)
Q' * C  (Transpose from the left)
C * Q   (No transpose from the right), and
C * Q'  (Transpose from the right)
```

Q is an orthogonal matrix defined as the product of k Householder reflectors as

```
Q = H(k) * H(k-1) * ... * H(1)
```

of order m if applying from the left, or n if applying from the right. Q is never stored, it is calculated from the Householder vectors and scalars returned by the QL factorization GEQLF.

Parameters

- **handle** – **[in]** rocblas_handle.
- **side** – **[in]** rocblas_side.

  Specifies from which side to apply Q.
- **trans** – **[in]** rocblas_operation.

  Specifies whether the matrix Q or its transpose is to be applied.
- **m** – **[in]** rocblas_int. m >= 0.

  Number of rows of matrix C.
- **n** – **[in]** rocblas_int. n >= 0.

  Number of columns of matrix C.
- **k** – **[in]** rocblas_int. k >= 0; k <= m if side is left, k <= n if side is right.

  The number of Householder reflectors that form Q.
- **A** – **[in]** pointer to type. Array on the GPU of size lda*k.

  The i-th column has the Householder vector v(i) associated with H(i) as returned by GEQLF in the last k columns of its argument A.
- **lda** – **[in]** rocblas_int. lda >= m if side is left, lda >= n if side is right.

  Leading dimension of A.
- **ipiv** – **[in]** pointer to type. Array on the GPU of dimension at least k.

  The scalar factors of the Householder matrices H(i) as returned by GEQLF.
- **C** – **[inout]** pointer to type. Array on the GPU of size ldc*n.

  On input, the matrix C. On output it is overwritten with Q*C, C*Q, Q'*C, or C*Q'.
- **ldc** – **[in]** rocblas_int. ldc >= m.

  Leading dimension of C.

rocsolver_<type>ormql()

rocblas_status **rocsolver_dormql**(rocblas_handle handle, const rocblas_side side, const rocblas_operation trans, const rocblas_int m, const rocblas_int n, const rocblas_int k, double *A, const rocblas_int lda, double *ipiv, double *C, const rocblas_int ldc)

rocblas_status **rocsolver_sormql**(rocblas_handle handle, const rocblas_side side, const rocblas_operation trans, const rocblas_int m, const rocblas_int n, const rocblas_int k, float *A, const rocblas_int lda, float *ipiv, float *C, const rocblas_int ldc)

ORMQL applies a matrix Q with orthonormal columns to a general m-by-n matrix C.

(This is the blocked version of the algorithm).

The matrix Q is applied in one of the following forms, depending on the values of side and trans:

```
Q  * C  (No transpose from the left)
Q' * C  (Transpose from the left)
C * Q   (No transpose from the right), and
C * Q'  (Transpose from the right)
```

Q is an orthogonal matrix defined as the product of k Householder reflectors as

```
Q = H(k) * H(k-1) * ... * H(1)
```

of order m if applying from the left, or n if applying from the right. Q is never stored, it is calculated from the Householder vectors and scalars returned by the QL factorization GEQLF.

Parameters

- **handle** – **[in]** rocblas_handle.
- **side** – **[in]** rocblas_side.

  Specifies from which side to apply Q.
- **trans** – **[in]** rocblas_operation.

  Specifies whether the matrix Q or its transpose is to be applied.
- **m** – **[in]** rocblas_int. m >= 0.

  Number of rows of matrix C.
- **n** – **[in]** rocblas_int. n >= 0.

  Number of columns of matrix C.
- **k** – **[in]** rocblas_int. k >= 0; k <= m if side is left, k <= n if side is right.

  The number of Householder reflectors that form Q.
- **A** – **[in]** pointer to type. Array on the GPU of size lda*k.

  The i-th column has the Householder vector v(i) associated with H(i) as returned by GEQLF in the last k columns of its argument A.
- **lda** – **[in]** rocblas_int. lda >= m if side is left, lda >= n if side is right.

  Leading dimension of A.
- **ipiv** – **[in]** pointer to type. Array on the GPU of dimension at least k.

  The scalar factors of the Householder matrices H(i) as returned by GEQLF.

- **C** – **[inout]** pointer to type. Array on the GPU of size ldc*n.

  On input, the matrix C. On output it is overwritten with Q*C, C*Q, Q'*C, or C*Q'.

- **ldc** – **[in]** rocblas_int. ldc >= m.

  Leading dimension of C.

rocsolver_<type>ormbr()

rocblas_status **rocsolver_dormbr**(rocblas_handle handle, const *rocblas_storev* storev, const rocblas_side side, const rocblas_operation trans, const rocblas_int m, const rocblas_int n, const rocblas_int k, double *A, const rocblas_int lda, double *ipiv, double *C, const rocblas_int ldc)

rocblas_status **rocsolver_sormbr**(rocblas_handle handle, const *rocblas_storev* storev, const rocblas_side side, const rocblas_operation trans, const rocblas_int m, const rocblas_int n, const rocblas_int k, float *A, const rocblas_int lda, float *ipiv, float *C, const rocblas_int ldc)

ORMBR applies a matrix Q with orthonormal rows or columns to a general m-by-n matrix C.

If storev is column-wise, then the matrix Q has orthonormal columns. If storev is row-wise, then the matrix Q has orthonormal rows. The matrix Q is applied in one of the following forms, depending on the values of side and trans:

```
Q  * C  (No transpose from the left)
Q' * C  (Transpose from the left)
C * Q   (No transpose from the right), and
C * Q'  (Transpose from the right)
```

The order nq of orthogonal matrix Q is nq = m if applying from the left, or nq = n if applying from the right.

When storev is column-wise, if nq >= k, then Q is defined as the product of k Householder reflectors of order nq

```
Q = H(1) * H(2) * ... * H(k),
```

and if nq < k, then Q is defined as the product

```
Q = H(1) * H(2) * ... * H(nq-1).
```

When storev is row-wise, if nq > k, then Q is defined as the product of k Householder reflectors of order nq

```
Q = H(1) * H(2) * ... * H(k),
```

and if n <= k, Q is defined as the product

```
Q = H(1) * H(2) * ... * H(nq-1)
```

The Householder matrices H(i) are never stored, they are computed from its corresponding Householder vectors v(i) and scalars ipiv_i as returned by GEBRD in its arguments A and tauq or taup.

Parameters

- **handle** – **[in]** rocblas_handle.
- **storev** – **[in]** *rocblas_storev* .

  Specifies whether to work column-wise or row-wise.

- **side** – **[in]** rocblas_side.

  Specifies from which side to apply Q.

- **trans** – **[in]** rocblas_operation.

  Specifies whether the matrix Q or its transpose is to be applied.

- **m** – **[in]** rocblas_int. m >= 0.

  Number of rows of matrix C.

- **n** – **[in]** rocblas_int. n >= 0.

  Number of columns of matrix C.

- **k** – **[in]** rocblas_int. k >= 0.

  The number of columns (if storev is colum-wise) or rows (if row-wise) of the original matrix reduced by GEBRD.

- **A** – **[in]** pointer to type. Array on the GPU of size lda*min(nq,k) if column-wise, or lda*nq if row-wise.

  The i-th column (or row) has the Householder vector v(i) associated with H(i) as returned by GEBRD.

- **lda** – **[in]** rocblas_int. lda >= nq if column-wise, or lda >= min(nq,k) if row-wise.

  Leading dimension of A.

- **ipiv** – **[in]** pointer to type. Array on the GPU of dimension at least min(nq,k).

  The scalar factors of the Householder matrices H(i) as returned by GEBRD.

- **C** – **[inout]** pointer to type. Array on the GPU of size ldc*n.

  On input, the matrix C. On output it is overwritten with Q*C, C*Q, Q'*C, or C*Q'.

- **ldc** – **[in]** rocblas_int. ldc >= m.

  Leading dimension of C.

rocsolver_<type>ormtr()

rocblas_status **rocsolver_dormtr**(rocblas_handle handle, const rocblas_side side, const rocblas_fill uplo, const rocblas_operation trans, const rocblas_int m, const rocblas_int n, double *A, const rocblas_int lda, double *ipiv, double *C, const rocblas_int ldc)

rocblas_status **rocsolver_sormtr**(rocblas_handle handle, const rocblas_side side, const rocblas_fill uplo, const rocblas_operation trans, const rocblas_int m, const rocblas_int n, float *A, const rocblas_int lda, float *ipiv, float *C, const rocblas_int ldc)

ORMTR applies an orthogonal matrix Q to a general m-by-n matrix C.

The matrix Q is applied in one of the following forms, depending on the values of side and trans:

```
Q  * C  (No transpose from the left)
Q' * C  (Transpose from the left)
C * Q   (No transpose from the right), and
C * Q'  (Transpose from the right)
```

The order nq of orthogonal matrix Q is nq = m if applying from the left, or nq = n if applying from the right.

Q is defined as the product of nq-1 Householder reflectors of order nq. If uplo indicates upper, then Q has the form

```
Q = H(nq-1) * H(nq-2) * ... * H(1).
```

On the other hand, if uplo indicates lower, then Q has the form

```
Q = H(1) * H(2) * ... * H(nq-1)
```

The Householder matrices H(i) are never stored, they are computed from its corresponding Householder vectors v(i) and scalars ipiv_i as returned by SYTRD in its arguments A and tau.

Parameters

- **handle** – **[in]** rocblas_handle.
- **side** – **[in]** rocblas_side.

  Specifies from which side to apply Q.
- **uplo** – **[in]** rocblas_fill.

  Specifies whether the SYTRD factorization was upper or lower triangular. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.
- **trans** – **[in]** rocblas_operation.

  Specifies whether the matrix Q or its transpose is to be applied.
- **m** – **[in]** rocblas_int. m >= 0.

  Number of rows of matrix C.
- **n** – **[in]** rocblas_int. n >= 0.

  Number of columns of matrix C.
- **A** – **[in]** pointer to type. Array on the GPU of size lda*nq.

  On entry, the (i+1)-th column (if uplo indicates upper) or i-th column (if uplo indicates lower) has the Householder vector v(i) as returned by SYTRD.
- **lda** – **[in]** rocblas_int. lda >= nq.

  Leading dimension of A.
- **ipiv** – **[in]** pointer to type. Array on the GPU of dimension at least nq-1.

  The scalar factors of the Householder matrices H(i) as returned by SYTRD.
- **C** – **[inout]** pointer to type. Array on the GPU of size ldc*n.

  On input, the matrix C. On output it is overwritten with Q*C, C*Q, Q'*C, or C*Q'.
- **ldc** – **[in]** rocblas_int. ldc >= m.

  Leading dimension of C.

2.6.2.7 Unitary matrices

rocsolver_<type>ung2r()

rocblas_status **rocsolver_zung2r**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, const rocblas_int k, rocblas_double_complex *A, const rocblas_int lda, rocblas_double_complex *ipiv)

rocblas_status **rocsolver_cung2r**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, const rocblas_int k, rocblas_float_complex *A, const rocblas_int lda, rocblas_float_complex *ipiv)

UNG2R generates a m-by-n complex Matrix Q with orthonormal columns.

(This is the unblocked version of the algorithm).

The matrix Q is defined as the first n columns of the product of k Householder reflectors of order m

```
Q = H(1) * H(2) * ... * H(k)
```

Householder matrices H(i) are never stored, they are computed from its corresponding Householder vector v(i) and scalar ipiv_i as returned by GEQRF.

Parameters

- **handle** – **[in]** rocblas_handle.
- **m** – **[in]** rocblas_int. m >= 0.

  The number of rows of the matrix Q.
- **n** – **[in]** rocblas_int. 0 <= n <= m.

  The number of columns of the matrix Q.
- **k** – **[in]** rocblas_int. 0 <= k <= n.

  The number of Householder reflectors.
- **A** – **[inout]** pointer to type. Array on the GPU of dimension lda*n.

  On entry, the i-th column has Householder vector v(i), for i = 1,2,...,k as returned in the first k columns of matrix A of GEQRF. On exit, the computed matrix Q.
- **lda** – **[in]** rocblas_int. lda >= m.

  Specifies the leading dimension of A.
- **ipiv** – **[in]** pointer to type. Array on the GPU of dimension at least k.

  The scalar factors of the Householder matrices H(i) as returned by GEQRF.

rocsolver_<type>ungqr()

rocblas_status **rocsolver_zungqr**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, const rocblas_int k, rocblas_double_complex *A, const rocblas_int lda, rocblas_double_complex *ipiv)

rocblas_status **rocsolver_cungqr**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, const rocblas_int k, rocblas_float_complex *A, const rocblas_int lda, rocblas_float_complex *ipiv)

UNGQR generates a m-by-n complex Matrix Q with orthonormal columns.

(This is the blocked version of the algorithm).

The matrix Q is defined as the first n columns of the product of k Householder reflectors of order m

```
Q = H(1) * H(2) * ... * H(k)
```

Householder matrices H(i) are never stored, they are computed from its corresponding Householder vector v(i) and scalar ipiv_i as returned by GEQRF.

Parameters

- **handle** – **[in]** rocblas_handle.
- **m** – **[in]** rocblas_int. m >= 0.

  The number of rows of the matrix Q.
- **n** – **[in]** rocblas_int. 0 <= n <= m.

  The number of columns of the matrix Q.
- **k** – **[in]** rocblas_int. 0 <= k <= n.

  The number of Householder reflectors.
- **A** – **[inout]** pointer to type. Array on the GPU of dimension lda*n.

  On entry, the i-th column has Householder vector v(i), for i = 1,2,...,k as returned in the first k columns of matrix A of GEQRF. On exit, the computed matrix Q.
- **lda** – **[in]** rocblas_int. lda >= m.

  Specifies the leading dimension of A.
- **ipiv** – **[in]** pointer to type. Array on the GPU of dimension at least k.

  The scalar factors of the Householder matrices H(i) as returned by GEQRF.

rocsolver_<type>ungl2()

rocblas_status **rocsolver_zungl2**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, const rocblas_int k, rocblas_double_complex *A, const rocblas_int lda, rocblas_double_complex *ipiv)

rocblas_status **rocsolver_cungl2**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, const rocblas_int k, rocblas_float_complex *A, const rocblas_int lda, rocblas_float_complex *ipiv)

UNGL2 generates a m-by-n complex Matrix Q with orthonormal rows.

(This is the unblocked version of the algorithm).

The matrix Q is defined as the first m rows of the product of k Householder reflectors of order n

```
Q = H(k)**H * H(k-1)**H * ... * H(1)**H
```

Householder matrices H(i) are never stored, they are computed from its corresponding Householder vector v(i) and scalar ipiv_i as returned by GELQF.

Parameters

- **handle** – **[in]** rocblas_handle.
- **m** – **[in]** rocblas_int. 0 <= m <= n.

  The number of rows of the matrix Q.
- **n** – **[in]** rocblas_int. n >= 0.

  The number of columns of the matrix Q.
- **k** – **[in]** rocblas_int. 0 <= k <= m.

  The number of Householder reflectors.
- **A** – **[inout]** pointer to type. Array on the GPU of dimension lda*n.

  On entry, the i-th row has Householder vector v(i), for i = 1,2,...,k as returned in the first k rows of matrix A of GELQF. On exit, the computed matrix Q.
- **lda** – **[in]** rocblas_int. lda >= m.

  Specifies the leading dimension of A.
- **ipiv** – **[in]** pointer to type. Array on the GPU of dimension at least k.

  The scalar factors of the Householder matrices H(i) as returned by GELQF.

rocsolver_<type>unglq()

rocblas_status **rocsolver_zunglq**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, const rocblas_int k, rocblas_double_complex *A, const rocblas_int lda, rocblas_double_complex *ipiv)

rocblas_status **rocsolver_cunglq**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, const rocblas_int k, rocblas_float_complex *A, const rocblas_int lda, rocblas_float_complex *ipiv)

UNGLQ generates a m-by-n complex Matrix Q with orthonormal rows.

(This is the blocked version of the algorithm).

The matrix Q is defined as the first m rows of the product of k Householder reflectors of order n

```
Q = H(k)**H * H(k-1)**H * ... * H(1)**H
```

Householder matrices H(i) are never stored, they are computed from its corresponding Householder vector v(i) and scalar ipiv_i as returned by GELQF.

Parameters

- **handle** – **[in]** rocblas_handle.

- **m** – **[in]** rocblas_int. 0 <= m <= n.

  The number of rows of the matrix Q.

- **n** – **[in]** rocblas_int. n >= 0.

  The number of columns of the matrix Q.

- **k** – **[in]** rocblas_int. 0 <= k <= m.

  The number of Householder reflectors.

- **A** – **[inout]** pointer to type. Array on the GPU of dimension lda*n.

  On entry, the i-th row has Householder vector v(i), for i = 1,2,...,k as returned in the first k rows of matrix A of GELQF. On exit, the computed matrix Q.

- **lda** – **[in]** rocblas_int. lda >= m.

  Specifies the leading dimension of A.

- **ipiv** – **[in]** pointer to type. Array on the GPU of dimension at least k.

  The scalar factors of the Householder matrices H(i) as returned by GELQF.

rocsolver_<type>ung2l()

rocblas_status **rocsolver_zung2l**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, const rocblas_int k, rocblas_double_complex *A, const rocblas_int lda, rocblas_double_complex *ipiv)

rocblas_status **rocsolver_cung2l**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, const rocblas_int k, rocblas_float_complex *A, const rocblas_int lda, rocblas_float_complex *ipiv)

UNG2L generates a m-by-n complex Matrix Q with orthonormal columns.

(This is the unblocked version of the algorithm).

The matrix Q is defined as the last n columns of the product of k Householder reflectors of order m

```
Q = H(k) * H(k-1) * ... * H(1)
```

Householder matrices H(i) are never stored, they are computed from its corresponding Householder vector v(i) and scalar ipiv_i as returned by GEQLF.

Parameters

- **handle** – **[in]** rocblas_handle.
- **m** – **[in]** rocblas_int. m >= 0.

  The number of rows of the matrix Q.

- **n** – **[in]** rocblas_int. 0 <= n <= m.

  The number of columns of the matrix Q.

- **k** – **[in]** rocblas_int. 0 <= k <= n.

  The number of Householder reflectors.

- **A** – **[inout]** pointer to type. Array on the GPU of dimension lda*n.

  On entry, the (n-k+i)-th column has Householder vector v(i), for i = 1,2,...,k as returned in the last k columns of matrix A of GEQLF. On exit, the computed matrix Q.

- **lda** – **[in]** rocblas_int. lda >= m.

  Specifies the leading dimension of A.

- **ipiv** – **[in]** pointer to type. Array on the GPU of dimension at least k.

  The scalar factors of the Householder matrices H(i) as returned by GEQLF.

rocsolver_<type>ungql()

rocblas_status **rocsolver_zungql**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, const rocblas_int k, rocblas_double_complex *A, const rocblas_int lda, rocblas_double_complex *ipiv)

rocblas_status **rocsolver_cungql**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, const rocblas_int k, rocblas_float_complex *A, const rocblas_int lda, rocblas_float_complex *ipiv)

UNGQL generates a m-by-n complex Matrix Q with orthonormal columns.

(This is the blocked version of the algorithm).

The matrix Q is defined as the last n columns of the product of k Householder reflectors of order m

```
Q = H(k) * H(k-1) * ... * H(1)
```

Householder matrices H(i) are never stored, they are computed from its corresponding Householder vector v(i) and scalar ipiv_i as returned by GEQLF.

Parameters

- **handle** – **[in]** rocblas_handle.
- **m** – **[in]** rocblas_int. m >= 0.

  The number of rows of the matrix Q.

- **n** – **[in]** rocblas_int. 0 <= n <= m.

  The number of columns of the matrix Q.

- **k** – **[in]** rocblas_int. 0 <= k <= n.

  The number of Householder reflectors.

- **A** – **[inout]** pointer to type. Array on the GPU of dimension lda*n.

  On entry, the (n-k+i)-th column has Householder vector v(i), for i = 1,2,...,k as returned in the last k columns of matrix A of GEQLF. On exit, the computed matrix Q.

- **lda** – **[in]** rocblas_int. lda >= m.

  Specifies the leading dimension of A.

- **ipiv** – **[in]** pointer to type. Array on the GPU of dimension at least k.

  The scalar factors of the Householder matrices H(i) as returned by GEQLF.

rocsolver_<type>ungbr()

rocblas_status **rocsolver_zungbr**(rocblas_handle handle, const *rocblas_storev* storev, const rocblas_int m, const rocblas_int n, const rocblas_int k, rocblas_double_complex *A, const rocblas_int lda, rocblas_double_complex *ipiv)

rocblas_status **rocsolver_cungbr**(rocblas_handle handle, const *rocblas_storev* storev, const rocblas_int m, const rocblas_int n, const rocblas_int k, rocblas_float_complex *A, const rocblas_int lda, rocblas_float_complex *ipiv)

UNGBR generates a m-by-n complex Matrix Q with orthonormal rows or columns.

If storev is column-wise, then the matrix Q has orthonormal columns. If m >= k, Q is defined as the first n columns of the product of k Householder reflectors of order m

```
Q = H(1) * H(2) * ... * H(k)
```

If m < k, Q is defined as the product of Householder reflectors of order m

```
Q = H(1) * H(2) * ... * H(m-1)
```

On the other hand, if storev is row-wise, then the matrix Q has orthonormal rows. If n > k, Q is defined as the first m rows of the product of k Householder reflectors of order n

```
Q = H(k) * H(k-1) * ... * H(1)
```

If n <= k, Q is defined as the product of Householder reflectors of order n

```
Q = H(n-1) * H(n-2) * ... * H(1)
```

The Householder matrices H(i) are never stored, they are computed from its corresponding Householder vectors v(i) and scalars ipiv_i as returned by GEBRD in its arguments A and tauq or taup.

Parameters

- **handle** – **[in]** rocblas_handle.
- **storev** – **[in]** *rocblas_storev* .

  Specifies whether to work column-wise or row-wise.
- **m** – **[in]** rocblas_int. m >= 0.

  The number of rows of the matrix Q. If row-wise, then min(n,k) <= m <= n.
- **n** – **[in]** rocblas_int. n >= 0.

  The number of columns of the matrix Q. If column-wise, then min(m,k) <= n <= m.
- **k** – **[in]** rocblas_int. k >= 0.

  The number of columns (if storev is colum-wise) or rows (if row-wise) of the original matrix reduced by GEBRD.
- **A** – **[inout]** pointer to type. Array on the GPU of dimension lda*n.

  On entry, the i-th column (or row) has the Householder vector v(i) as returned by GEBRD. On exit, the computed matrix Q.
- **lda** – **[in]** rocblas_int. lda >= m.

  Specifies the leading dimension of A.

- **ipiv** – **[in]** pointer to type. Array on the GPU of dimension min(m,k) if column-wise, or min(n,k) if row-wise.

  The scalar factors of the Householder matrices H(i) as returned by GEBRD.

rocsolver_<type>ungtr()

rocblas_status **rocsolver_zungtr**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, rocblas_double_complex *A, const rocblas_int lda, rocblas_double_complex *ipiv)

rocblas_status **rocsolver_cungtr**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, rocblas_float_complex *A, const rocblas_int lda, rocblas_float_complex *ipiv)

UNGTR generates a n-by-n unitary Matrix Q.

Q is defined as the product of n-1 Householder reflectors of order n. If uplo indicates upper, then Q has the form

```
Q = H(n-1) * H(n-2) * ... * H(1)
```

On the other hand, if uplo indicates lower, then Q has the form

```
Q = H(1) * H(2) * ... * H(n-1)
```

The Householder matrices H(i) are never stored, they are computed from its corresponding Householder vectors v(i) and scalars ipiv_i as returned by HETRD in its arguments A and tau.

Parameters

- **handle** – **[in]** rocblas_handle.
- **uplo** – **[in]** rocblas_fill.

  Specifies whether the HETRD factorization was upper or lower triangular. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.

- **n** – **[in]** rocblas_int. n >= 0.

  The number of rows and columns of the matrix Q.

- **A** – **[inout]** pointer to type. Array on the GPU of dimension lda*n.

  On entry, the (i+1)-th column (if uplo indicates upper) or i-th column (if uplo indicates lower) has the Householder vector v(i) as returned by HETRD. On exit, the computed matrix Q.

- **lda** – **[in]** rocblas_int. lda >= m.

  Specifies the leading dimension of A.

- **ipiv** – **[in]** pointer to type. Array on the GPU of dimension n-1.

  The scalar factors of the Householder matrices H(i) as returned by HETRD.

rocsolver_<type>unm2r()

rocblas_status **rocsolver_zunm2r**(rocblas_handle handle, const rocblas_side side, const rocblas_operation trans, const rocblas_int m, const rocblas_int n, const rocblas_int k, rocblas_double_complex *A, const rocblas_int lda, rocblas_double_complex *ipiv, rocblas_double_complex *C, const rocblas_int ldc)

rocblas_status **rocsolver_cunm2r**(rocblas_handle handle, const rocblas_side side, const rocblas_operation trans, const rocblas_int m, const rocblas_int n, const rocblas_int k, rocblas_float_complex *A, const rocblas_int lda, rocblas_float_complex *ipiv, rocblas_float_complex *C, const rocblas_int ldc)

UNM2R applies a complex matrix Q with orthonormal columns to a general m-by-n matrix C.

(This is the unblocked version of the algorithm).

The matrix Q is applied in one of the following forms, depending on the values of side and trans:

```
Q  * C  (No transpose from the left)
Q' * C  (Conjugate transpose from the left)
C * Q   (No transpose from the right), and
C * Q'  (Conjugate transpose from the right)
```

Q is a unitary matrix defined as the product of k Householder reflectors as

```
Q = H(1) * H(2) * ... * H(k)
```

of order m if applying from the left, or n if applying from the right. Q is never stored, it is calculated from the Householder vectors and scalars returned by the QR factorization GEQRF.

Parameters

- **handle** – **[in]** rocblas_handle.
- **side** – **[in]** rocblas_side.

  Specifies from which side to apply Q.
- **trans** – **[in]** rocblas_operation.

  Specifies whether the matrix Q or its conjugate transpose is to be applied.
- **m** – **[in]** rocblas_int. m >= 0.

  Number of rows of matrix C.
- **n** – **[in]** rocblas_int. n >= 0.

  Number of columns of matrix C.
- **k** – **[in]** rocblas_int. k >= 0; k <= m if side is left, k <= n if side is right.

  The number of Householder reflectors that form Q.
- **A** – **[in]** pointer to type. Array on the GPU of size lda*k.

  The i-th column has the Householder vector v(i) associated with H(i) as returned by GEQRF in the first k columns of its argument A.
- **lda** – **[in]** rocblas_int. lda >= m if side is left, or lda >= n if side is right.

  Leading dimension of A.

- **ipiv** – **[in]** pointer to type. Array on the GPU of dimension at least k.

  The scalar factors of the Householder matrices H(i) as returned by GEQRF.

- **C** – **[inout]** pointer to type. Array on the GPU of size ldc*n.

  On input, the matrix C. On output it is overwritten with Q*C, C*Q, Q'*C, or C*Q'.

- **ldc** – **[in]** rocblas_int. ldc >= m.

  Leading dimension of C.

rocsolver_<type>unmqr()

rocblas_status **rocsolver_zunmqr**(rocblas_handle handle, const rocblas_side side, const rocblas_operation trans, const rocblas_int m, const rocblas_int n, const rocblas_int k, rocblas_double_complex *A, const rocblas_int lda, rocblas_double_complex *ipiv, rocblas_double_complex *C, const rocblas_int ldc)

rocblas_status **rocsolver_cunmqr**(rocblas_handle handle, const rocblas_side side, const rocblas_operation trans, const rocblas_int m, const rocblas_int n, const rocblas_int k, rocblas_float_complex *A, const rocblas_int lda, rocblas_float_complex *ipiv, rocblas_float_complex *C, const rocblas_int ldc)

UNMQR applies a complex matrix Q with orthonormal columns to a general m-by-n matrix C.

(This is the blocked version of the algorithm).

The matrix Q is applied in one of the following forms, depending on the values of side and trans:

```
Q  * C  (No transpose from the left)
Q' * C  (Conjugate transpose from the left)
C * Q   (No transpose from the right), and
C * Q'  (Conjugate transpose from the right)
```

Q is a unitary matrix defined as the product of k Householder reflectors as

```
Q = H(1) * H(2) * ... * H(k)
```

of order m if applying from the left, or n if applying from the right. Q is never stored, it is calculated from the Householder vectors and scalars returned by the QR factorization GEQRF.

Parameters

- **handle** – **[in]** rocblas_handle.
- **side** – **[in]** rocblas_side.

  Specifies from which side to apply Q.

- **trans** – **[in]** rocblas_operation.

  Specifies whether the matrix Q or its conjugate transpose is to be applied.

- **m** – **[in]** rocblas_int. m >= 0.

  Number of rows of matrix C.

- **n** – **[in]** rocblas_int. n >= 0.

  Number of columns of matrix C.

- **k** – **[in]** rocblas_int. k >= 0; k <= m if side is left, k <= n if side is right.

  The number of Householder reflectors that form Q.

- **A** – **[in]** pointer to type. Array on the GPU of size lda*k.

  The i-th column has the Householder vector v(i) associated with H(i) as returned by GEQRF in the first k columns of its argument A.

- **lda** – **[in]** rocblas_int. lda >= m if side is left, or lda >= n if side is right.

  Leading dimension of A.

- **ipiv** – **[in]** pointer to type. Array on the GPU of dimension at least k.

  The scalar factors of the Householder matrices H(i) as returned by GEQRF.

- **C** – **[inout]** pointer to type. Array on the GPU of size ldc*n.

  On input, the matrix C. On output it is overwritten with Q*C, C*Q, Q'*C, or C*Q'.

- **ldc** – **[in]** rocblas_int. ldc >= m.

  Leading dimension of C.

rocsolver_<type>unml2()

rocblas_status **rocsolver_zunml2**(rocblas_handle handle, const rocblas_side side, const rocblas_operation trans, const rocblas_int m, const rocblas_int n, const rocblas_int k, rocblas_double_complex *A, const rocblas_int lda, rocblas_double_complex *ipiv, rocblas_double_complex *C, const rocblas_int ldc)

rocblas_status **rocsolver_cunml2**(rocblas_handle handle, const rocblas_side side, const rocblas_operation trans, const rocblas_int m, const rocblas_int n, const rocblas_int k, rocblas_float_complex *A, const rocblas_int lda, rocblas_float_complex *ipiv, rocblas_float_complex *C, const rocblas_int ldc)

UNML2 applies a complex matrix Q with orthonormal rows to a general m-by-n matrix C.

(This is the unblocked version of the algorithm).

The matrix Q is applied in one of the following forms, depending on the values of side and trans:

```
Q  * C  (No transpose from the left)
Q' * C  (Conjugate transpose from the left)
C * Q   (No transpose from the right), and
C * Q'  (Conjugate transpose from the right)
```

Q is a unitary matrix defined as the product of k Householder reflectors as

```
Q = H(k)**H * H(k-1)**H * ... * H(1)**H
```

of order m if applying from the left, or n if applying from the right. Q is never stored, it is calculated from the Householder vectors and scalars returned by the LQ factorization GELQF.

Parameters

- **handle** – **[in]** rocblas_handle.
- **side** – **[in]** rocblas_side.

  Specifies from which side to apply Q.

- **trans** – **[in]** rocblas_operation.

  Specifies whether the matrix Q or its conjugate transpose is to be applied.

- **m** – **[in]** rocblas_int. m >= 0.

  Number of rows of matrix C.

- **n** – **[in]** rocblas_int. n >= 0.

  Number of columns of matrix C.

- **k** – **[in]** rocblas_int. k >= 0; k <= m if side is left, k <= n if side is right.

  The number of Householder reflectors that form Q.

- **A** – **[in]** pointer to type. Array on the GPU of size lda*m if side is left, or lda*n if side is right.

  The i-th row has the Householder vector v(i) associated with H(i) as returned by GELQF in the first k rows of its argument A.

- **lda** – **[in]** rocblas_int. lda >= k.

  Leading dimension of A.

- **ipiv** – **[in]** pointer to type. Array on the GPU of dimension at least k.

  The scalar factors of the Householder matrices H(i) as returned by GELQF.

- **C** – **[inout]** pointer to type. Array on the GPU of size ldc*n.

  On input, the matrix C. On output it is overwritten with Q*C, C*Q, Q'*C, or C*Q'.

- **ldc** – **[in]** rocblas_int. ldc >= m.

  Leading dimension of C.

rocsolver_<type>unmlq()

rocblas_status **rocsolver_zunmlq**(rocblas_handle handle, const rocblas_side side, const rocblas_operation trans, const rocblas_int m, const rocblas_int n, const rocblas_int k, rocblas_double_complex *A, const rocblas_int lda, rocblas_double_complex *ipiv, rocblas_double_complex *C, const rocblas_int ldc)

rocblas_status **rocsolver_cunmlq**(rocblas_handle handle, const rocblas_side side, const rocblas_operation trans, const rocblas_int m, const rocblas_int n, const rocblas_int k, rocblas_float_complex *A, const rocblas_int lda, rocblas_float_complex *ipiv, rocblas_float_complex *C, const rocblas_int ldc)

UNMLQ applies a complex matrix Q with orthonormal rows to a general m-by-n matrix C.

(This is the blocked version of the algorithm).

The matrix Q is applied in one of the following forms, depending on the values of side and trans:

```
Q  * C  (No transpose from the left)
Q' * C  (Conjugate transpose from the left)
C * Q   (No transpose from the right), and
C * Q'  (Conjugate transpose from the right)
```

Q is a unitary matrix defined as the product of k Householder reflectors as

```
Q = H(k)**H * H(k-1)**H * ... * H(1)**H
```

of order m if applying from the left, or n if applying from the right. Q is never stored, it is calculated from the Householder vectors and scalars returned by the LQ factorization GELQF.

Parameters

- **handle** – **[in]** rocblas_handle.
- **side** – **[in]** rocblas_side.

  Specifies from which side to apply Q.
- **trans** – **[in]** rocblas_operation.

  Specifies whether the matrix Q or its conjugate transpose is to be applied.
- **m** – **[in]** rocblas_int. m >= 0.

  Number of rows of matrix C.
- **n** – **[in]** rocblas_int. n >= 0.

  Number of columns of matrix C.
- **k** – **[in]** rocblas_int. k >= 0; k <= m if side is left, k <= n if side is right.

  The number of Householder reflectors that form Q.
- **A** – **[in]** pointer to type. Array on the GPU of size lda*m if side is left, or lda*n if side is right.

  The i-th row has the Householder vector v(i) associated with H(i) as returned by GELQF in the first k rows of its argument A.
- **lda** – **[in]** rocblas_int. lda >= k.

  Leading dimension of A.
- **ipiv** – **[in]** pointer to type. Array on the GPU of dimension at least k.

  The scalar factors of the Householder matrices H(i) as returned by GELQF.
- **C** – **[inout]** pointer to type. Array on the GPU of size ldc*n.

  On input, the matrix C. On output it is overwritten with Q*C, C*Q, Q'*C, or C*Q'.
- **ldc** – **[in]** rocblas_int. ldc >= m.

  Leading dimension of C.

rocsolver_<type>unm2l()

rocblas_status **rocsolver_zunm2l**(rocblas_handle handle, const rocblas_side side, const rocblas_operation trans, const rocblas_int m, const rocblas_int n, const rocblas_int k, rocblas_double_complex *A, const rocblas_int lda, rocblas_double_complex *ipiv, rocblas_double_complex *C, const rocblas_int ldc)

rocblas_status **rocsolver_cunm2l**(rocblas_handle handle, const rocblas_side side, const rocblas_operation trans, const rocblas_int m, const rocblas_int n, const rocblas_int k, rocblas_float_complex *A, const rocblas_int lda, rocblas_float_complex *ipiv, rocblas_float_complex *C, const rocblas_int ldc)

UNM2L applies a complex matrix Q with orthonormal columns to a general m-by-n matrix C.

(This is the unblocked version of the algorithm).

The matrix Q is applied in one of the following forms, depending on the values of side and trans:

```
Q  * C  (No transpose from the left)
Q' * C  (Conjugate transpose from the left)
C * Q   (No transpose from the right), and
C * Q'  (Conjugate transpose from the right)
```

Q is a unitary matrix defined as the product of k Householder reflectors as

```
Q = H(k) * H(k-1) * ... * H(1)
```

of order m if applying from the left, or n if applying from the right. Q is never stored, it is calculated from the Householder vectors and scalars returned by the QL factorization GEQLF.

Parameters

- **handle** – **[in]** rocblas_handle.
- **side** – **[in]** rocblas_side.

  Specifies from which side to apply Q.
- **trans** – **[in]** rocblas_operation.

  Specifies whether the matrix Q or its conjugate transpose is to be applied.
- **m** – **[in]** rocblas_int. m >= 0.

  Number of rows of matrix C.
- **n** – **[in]** rocblas_int. n >= 0.

  Number of columns of matrix C.
- **k** – **[in]** rocblas_int. k >= 0; k <= m if side is left, k <= n if side is right.

  The number of Householder reflectors that form Q.
- **A** – **[in]** pointer to type. Array on the GPU of size lda*k.

  The i-th column has the Householder vector v(i) associated with H(i) as returned by GEQLF in the last k columns of its argument A.
- **lda** – **[in]** rocblas_int. lda >= m if side is left, lda >= n if side is right.

  Leading dimension of A.
- **ipiv** – **[in]** pointer to type. Array on the GPU of dimension at least k.

  The scalar factors of the Householder matrices H(i) as returned by GEQLF.
- **C** – **[inout]** pointer to type. Array on the GPU of size ldc*n.

  On input, the matrix C. On output it is overwritten with Q*C, C*Q, Q'*C, or C*Q'.
- **ldc** – **[in]** rocblas_int. ldc >= m.

  Leading dimension of C.

rocsolver_<type>unmql()

rocblas_status **rocsolver_zunmql**(rocblas_handle handle, const rocblas_side side, const rocblas_operation trans, const rocblas_int m, const rocblas_int n, const rocblas_int k, rocblas_double_complex *A, const rocblas_int lda, rocblas_double_complex *ipiv, rocblas_double_complex *C, const rocblas_int ldc)

rocblas_status **rocsolver_cunmql**(rocblas_handle handle, const rocblas_side side, const rocblas_operation trans, const rocblas_int m, const rocblas_int n, const rocblas_int k, rocblas_float_complex *A, const rocblas_int lda, rocblas_float_complex *ipiv, rocblas_float_complex *C, const rocblas_int ldc)

UNMQL applies a complex matrix Q with orthonormal columns to a general m-by-n matrix C.

(This is the blocked version of the algorithm).

The matrix Q is applied in one of the following forms, depending on the values of side and trans:

```
Q  * C  (No transpose from the left)
Q' * C  (Conjugate transpose from the left)
C * Q   (No transpose from the right), and
C * Q'  (Conjugate transpose from the right)
```

Q is a unitary matrix defined as the product of k Householder reflectors as

```
Q = H(k) * H(k-1) * ... * H(1)
```

of order m if applying from the left, or n if applying from the right. Q is never stored, it is calculated from the Householder vectors and scalars returned by the QL factorization GEQLF.

Parameters

- **handle** – **[in]** rocblas_handle.
- **side** – **[in]** rocblas_side.

  Specifies from which side to apply Q.
- **trans** – **[in]** rocblas_operation.

  Specifies whether the matrix Q or its conjugate transpose is to be applied.
- **m** – **[in]** rocblas_int. m >= 0.

  Number of rows of matrix C.
- **n** – **[in]** rocblas_int. n >= 0.

  Number of columns of matrix C.
- **k** – **[in]** rocblas_int. k >= 0; k <= m if side is left, k <= n if side is right.

  The number of Householder reflectors that form Q.
- **A** – **[in]** pointer to type. Array on the GPU of size lda*k.

  The i-th column has the Householder vector v(i) associated with H(i) as returned by GEQLF in the last k columns of its argument A.
- **lda** – **[in]** rocblas_int. lda >= m if side is left, lda >= n if side is right.

  Leading dimension of A.

- **ipiv** – **[in]** pointer to type. Array on the GPU of dimension at least k.

  The scalar factors of the Householder matrices H(i) as returned by GEQLF.

- **C** – **[inout]** pointer to type. Array on the GPU of size ldc*n.

  On input, the matrix C. On output it is overwritten with Q*C, C*Q, Q'*C, or C*Q'.

- **ldc** – **[in]** rocblas_int. ldc >= m.

  Leading dimension of C.

rocsolver_<type>unmbr()

rocblas_status **rocsolver_zunmbr**(rocblas_handle handle, const *rocblas_storev* storev, const rocblas_side side, const rocblas_operation trans, const rocblas_int m, const rocblas_int n, const rocblas_int k, rocblas_double_complex *A, const rocblas_int lda, rocblas_double_complex *ipiv, rocblas_double_complex *C, const rocblas_int ldc)

rocblas_status **rocsolver_cunmbr**(rocblas_handle handle, const *rocblas_storev* storev, const rocblas_side side, const rocblas_operation trans, const rocblas_int m, const rocblas_int n, const rocblas_int k, rocblas_float_complex *A, const rocblas_int lda, rocblas_float_complex *ipiv, rocblas_float_complex *C, const rocblas_int ldc)

UNMBR applies a complex matrix Q with orthonormal rows or columns to a general m-by-n matrix C.

If storev is column-wise, then the matrix Q has orthonormal columns. If storev is row-wise, then the matrix Q has orthonormal rows. The matrix Q is applied in one of the following forms, depending on the values of side and trans:

```
Q  * C  (No transpose from the left)
Q' * C  (Conjugate transpose from the left)
C * Q   (No transpose from the right), and
C * Q'  (Conjugate transpose from the right)
```

The order nq of unitary matrix Q is nq = m if applying from the left, or nq = n if applying from the right.

When storev is column-wise, if nq >= k, then Q is defined as the product of k Householder reflectors of order nq

```
Q = H(1) * H(2) * ... * H(k),
```

and if nq < k, then Q is defined as the product

```
Q = H(1) * H(2) * ... * H(nq-1).
```

When storev is row-wise, if nq > k, then Q is defined as the product of k Householder reflectors of order nq

```
Q = H(1) * H(2) * ... * H(k),
```

and if n <= k, Q is defined as the product

```
Q = H(1) * H(2) * ... * H(nq-1)
```

The Householder matrices H(i) are never stored, they are computed from its corresponding Householder vectors v(i) and scalars ipiv_i as returned by GEBRD in its arguments A and tauq or taup.

Parameters

- **handle** – **[in]** rocblas_handle.
- **storev** – **[in]** *rocblas_storev* .

  Specifies whether to work column-wise or row-wise.
- **side** – **[in]** rocblas_side.

  Specifies from which side to apply Q.
- **trans** – **[in]** rocblas_operation.

  Specifies whether the matrix Q or its conjugate transpose is to be applied.
- **m** – **[in]** rocblas_int. m >= 0.

  Number of rows of matrix C.
- **n** – **[in]** rocblas_int. n >= 0.

  Number of columns of matrix C.
- **k** – **[in]** rocblas_int. k >= 0.

  The number of columns (if storev is colum-wise) or rows (if row-wise) of the original matrix reduced by GEBRD.
- **A** – **[in]** pointer to type. Array on the GPU of size lda*min(nq,k) if column-wise, or lda*nq if row-wise.

  The i-th column (or row) has the Householder vector v(i) associated with H(i) as returned by GEBRD.
- **lda** – **[in]** rocblas_int. lda >= nq if column-wise, or lda >= min(nq,k) if row-wise.

  Leading dimension of A.
- **ipiv** – **[in]** pointer to type. Array on the GPU of dimension at least min(nq,k).

  The scalar factors of the Householder matrices H(i) as returned by GEBRD.
- **C** – **[inout]** pointer to type. Array on the GPU of size ldc*n.

  On input, the matrix C. On output it is overwritten with Q*C, C*Q, Q'*C, or C*Q'.
- **ldc** – **[in]** rocblas_int. ldc >= m.

  Leading dimension of C.

rocsolver_<type>unmtr()

rocblas_status **rocsolver_zunmtr**(rocblas_handle handle, const rocblas_side side, const rocblas_fill uplo, const rocblas_operation trans, const rocblas_int m, const rocblas_int n, rocblas_double_complex *A, const rocblas_int lda, rocblas_double_complex *ipiv, rocblas_double_complex *C, const rocblas_int ldc)

rocblas_status **rocsolver_cunmtr**(rocblas_handle handle, const rocblas_side side, const rocblas_fill uplo, const rocblas_operation trans, const rocblas_int m, const rocblas_int n, rocblas_float_complex *A, const rocblas_int lda, rocblas_float_complex *ipiv, rocblas_float_complex *C, const rocblas_int ldc)

UNMTR applies a unitary matrix Q to a general m-by-n matrix C.

The matrix Q is applied in one of the following forms, depending on the values of side and trans:

```
Q  * C  (No transpose from the left)
Q' * C  (Conjugate transpose from the left)
C * Q   (No transpose from the right), and
C * Q'  (Conjugate transpose from the right)
```

The order nq of unitary matrix Q is nq = m if applying from the left, or nq = n if applying from the right.

Q is defined as the product of nq-1 Householder reflectors of order nq. If uplo indicates upper, then Q has the form

```
Q = H(nq-1) * H(nq-2) * ... * H(1).
```

On the other hand, if uplo indicates lower, then Q has the form

```
Q = H(1) * H(2) * ... * H(nq-1)
```

The Householder matrices H(i) are never stored, they are computed from its corresponding Householder vectors v(i) and scalars ipiv_i as returned by HETRD in its arguments A and tau.

Parameters

- **handle** – **[in]** rocblas_handle.
- **side** – **[in]** rocblas_side.

  Specifies from which side to apply Q.
- **uplo** – **[in]** rocblas_fill.

  Specifies whether the SYTRD factorization was upper or lower triangular. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.
- **trans** – **[in]** rocblas_operation.

  Specifies whether the matrix Q or its conjugate transpose is to be applied.
- **m** – **[in]** rocblas_int. m >= 0.

  Number of rows of matrix C.
- **n** – **[in]** rocblas_int. n >= 0.

  Number of columns of matrix C.
- **A** – **[in]** pointer to type. Array on the GPU of size lda*nq.

  On entry, the (i+1)-th column (if uplo indicates upper) or i-th column (if uplo indicates lower) has the Householder vector v(i) as returned by HETRD.
- **lda** – **[in]** rocblas_int. lda >= nq.

  Leading dimension of A.
- **ipiv** – **[in]** pointer to type. Array on the GPU of dimension at least nq-1.

  The scalar factors of the Householder matrices H(i) as returned by HETRD.
- **C** – **[inout]** pointer to type. Array on the GPU of size ldc*n.

  On input, the matrix C. On output it is overwritten with Q*C, C*Q, Q'*C, or C*Q'.
- **ldc** – **[in]** rocblas_int. ldc >= m.

  Leading dimension of C.

2.6.2.8 Eigensolvers

rocsolver_<type>sterf()

rocblas_status **rocsolver_dsterf**(rocblas_handle handle, const rocblas_int n, double *D, double *E, rocblas_int *info)

rocblas_status **rocsolver_ssterf**(rocblas_handle handle, const rocblas_int n, float *D, float *E, rocblas_int *info)

STERF computes the eigenvalues of a symmetric tridiagonal matrix.

The eigenvalues of the symmetric tridiagonal matrix are computed by the Pal-Walker-Kahan variant of the QL/QR algorithm, and returned in increasing order.

The matrix is not represented explicitly, but rather as the array of diagonal elements D and the array of symmetric off-diagonal elements E as returned by, e.g., SYTRD or HETRD.

Parameters

- **handle** – **[in]** rocblas_handle.
- **n** – **[in]** rocblas_int. n >= 0.

  The number of rows and columns of the tridiagonal matrix.
- **D** – **[inout]** pointer to real type. Array on the GPU of dimension n.

  On entry, the diagonal elements of the matrix. On exit, if info = 0, the eigenvalues in increasing order. If info > 0, the diagonal elements of a tridiagonal matrix that is similar to the original matrix (i.e. has the same eigenvalues).
- **E** – **[inout]** pointer to real type. Array on the GPU of dimension n-1.

  On entry, the off-diagonal elements of the matrix. On exit, if info = 0, this array converges to zero. If info > 0, the off-diagonal elements of a tridiagonal matrix that is similar to the original matrix (i.e. has the same eigenvalues).
- **info** – **[out]** pointer to a rocblas_int on the GPU.

  If info = 0, successful exit. If info = i > 0, STERF did not converge. i elements of E did not converge to zero.

rocsolver_<type>steqr()

rocblas_status **rocsolver_zsteqr**(rocblas_handle handle, const *rocblas_evect* compC, const rocblas_int n, double *D, double *E, rocblas_double_complex *C, const rocblas_int ldc, rocblas_int *info)

rocblas_status **rocsolver_csteqr**(rocblas_handle handle, const *rocblas_evect* compC, const rocblas_int n, float *D, float *E, rocblas_float_complex *C, const rocblas_int ldc, rocblas_int *info)

rocblas_status **rocsolver_dsteqr**(rocblas_handle handle, const *rocblas_evect* compC, const rocblas_int n, double *D, double *E, double *C, const rocblas_int ldc, rocblas_int *info)

rocblas_status **rocsolver_ssteqr**(rocblas_handle handle, const *rocblas_evect* compC, const rocblas_int n, float *D, float *E, float *C, const rocblas_int ldc, rocblas_int *info)

STEQR computes the eigenvalues and (optionally) eigenvectors of a symmetric tridiagonal matrix.

The eigenvalues of the symmetric tridiagonal matrix are computed by the implicit QL/QR algorithm, and returned in increasing order.

The matrix is not represented explicitly, but rather as the array of diagonal elements D and the array of symmetric off-diagonal elements E as returned by, e.g., SYTRD or HETRD. If the tridiagonal matrix is the reduced form of a full symmetric/Hermitian matrix as returned by, e.g., SYTRD or HETRD, then the eigenvectors of the original matrix can also be computed, depending on the value of compC.

Parameters

- **handle** – **[in]** rocblas_handle.
- **compC** – **[in]** *rocblas_evect* .

  Specifies how the eigenvectors are computed.
- **n** – **[in]** rocblas_int. n >= 0.

  The number of rows and columns of the tridiagonal matrix.
- **D** – **[inout]** pointer to real type. Array on the GPU of dimension n.

  On entry, the diagonal elements of the matrix. On exit, if info = 0, the eigenvalues in increasing order. If info > 0, the diagonal elements of a tridiagonal matrix that is similar to the original matrix (i.e. has the same eigenvalues).
- **E** – **[inout]** pointer to real type. Array on the GPU of dimension n-1.

  On entry, the off-diagonal elements of the matrix. On exit, if info = 0, this array converges to zero. If info > 0, the off-diagonal elements of a tridiagonal matrix that is similar to the original matrix (i.e. has the same eigenvalues).
- **C** – **[inout]** pointer to type. Array on the GPU of dimension ldc*n.

  On entry, if compC is original, the orthogonal/unitary matrix used for the reduction to tridiagonal form as returned by, e.g., ORGTR or UNGTR. On exit, it is overwritten with the eigenvectors of the original symmetric/Hermitian matrix (if compC is original), or the eigenvectors of the tridiagonal matrix (if compC is tridiagonal). (Not referenced if compC is none).
- **ldc** – **[in]** rocblas_int. ldc >= n if compc is original or tridiagonal.

  Specifies the leading dimension of C. (Not referenced if compC is none).
- **info** – **[out]** pointer to a rocblas_int on the GPU.

  If info = 0, successful exit. If info = i > 0, STEQR did not converge. i elements of E did not converge to zero.

2.6.3 LAPACK Functions

LAPACK routines solve complex Numerical Linear Algebra problems.

2.6.3.1 Special Matrix Factorizations

rocsolver_<type>potf2()

rocblas_status **rocsolver_zpotf2**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, rocblas_double_complex *A, const rocblas_int lda, rocblas_int *info)

rocblas_status **rocsolver_cpotf2**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, rocblas_float_complex *A, const rocblas_int lda, rocblas_int *info)

rocblas_status **rocsolver_dpotf2**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, double *A, const rocblas_int lda, rocblas_int *info)

rocblas_status **rocsolver_spotf2**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, float *A, const rocblas_int lda, rocblas_int *info)

POTF2 computes the Cholesky factorization of a real symmetric/complex Hermitian positive definite matrix A.

(This is the unblocked version of the algorithm).

The factorization has the form:

```
A = U' * U, or
A = L  * L'
```

depending on the value of uplo. U is an upper triangular matrix and L is lower triangular.

Parameters

- **handle** – **[in]** rocblas_handle.
- **uplo** – **[in]** rocblas_fill.

  Specifies whether the factorization is upper or lower triangular. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.
- **n** – **[in]** rocblas_int. n >= 0.

  The matrix dimensions.
- **A** – **[inout]** pointer to type. Array on the GPU of dimension lda*n.

  On entry, the matrix A to be factored. On exit, the lower or upper triangular factor.
- **lda** – **[in]** rocblas_int. lda >= n.

  specifies the leading dimension of A.
- **info** – **[out]** pointer to a rocblas_int on the GPU.

  If info = 0, successful factorization of matrix A. If info = i > 0, the leading minor of order i of A is not positive definite. The factorization stopped at this point.

rocsolver_<type>potf2_batched()

rocblas_status **rocsolver_zpotf2_batched**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, rocblas_double_complex *const A[], const rocblas_int lda, rocblas_int *info, const rocblas_int batch_count)

rocblas_status **rocsolver_cpotf2_batched**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, rocblas_float_complex *const A[], const rocblas_int lda, rocblas_int *info, const rocblas_int batch_count)

rocblas_status **rocsolver_dpotf2_batched**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, double *const A[], const rocblas_int lda, rocblas_int *info, const rocblas_int batch_count)

rocblas_status **rocsolver_spotf2_batched**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, float *const A[], const rocblas_int lda, rocblas_int *info, const rocblas_int batch_count)

POTF2_BATCHED computes the Cholesky factorization of a batch of real symmetric/complex Hermitian positive definite matrices.

(This is the unblocked version of the algorithm).

The factorization of matrix A_i in the batch has the form:

```
A_i = U_i' * U_i, or
A_i = L_i  * L_i'
```

depending on the value of uplo. U_i is an upper triangular matrix and L_i is lower triangular.

Parameters

- **handle** – **[in]** rocblas_handle.
- **uplo** – **[in]** rocblas_fill.

  Specifies whether the factorization is upper or lower triangular. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.
- **n** – **[in]** rocblas_int. n >= 0.

  The dimension of matrix A_i.
- **A** – **[inout]** array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n.

  On entry, the matrices A_i to be factored. On exit, the upper or lower triangular factors.
- **lda** – **[in]** rocblas_int. lda >= n.

  specifies the leading dimension of A_i.
- **info** – **[out]** pointer to rocblas_int. Array of batch_count integers on the GPU.

  If info_i = 0, successful factorization of matrix A_i. If info_i = j > 0, the leading minor of order j of A_i is not positive definite. The i-th factorization stopped at this point.
- **batch_count** – **[in]** rocblas_int. batch_count >= 0.

  Number of matrices in the batch.

rocsolver_<type>potf2_strided_batched()

rocblas_status **rocsolver_zpotf2_strided_batched**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, rocblas_double_complex *A, const rocblas_int lda, const rocblas_stride strideA, rocblas_int *info, const rocblas_int batch_count)

rocblas_status **rocsolver_cpotf2_strided_batched**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, rocblas_float_complex *A, const rocblas_int lda, const rocblas_stride strideA, rocblas_int *info, const rocblas_int batch_count)

rocblas_status **rocsolver_dpotf2_strided_batched**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, double *A, const rocblas_int lda, const rocblas_stride strideA, rocblas_int *info, const rocblas_int batch_count)

rocblas_status **rocsolver_spotf2_strided_batched**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, float *A, const rocblas_int lda, const rocblas_stride strideA, rocblas_int *info, const rocblas_int batch_count)

POTF2_STRIDED_BATCHED computes the Cholesky factorization of a batch of real symmetric/complex Hermitian positive definite matrices.

(This is the unblocked version of the algorithm).

The factorization of matrix A_i in the batch has the form:

```
A_i = U_i' * U_i, or
A_i = L_i  * L_i'
```

depending on the value of uplo. U_i is an upper triangular matrix and L_i is lower triangular.

Parameters

- **handle** – **[in]** rocblas_handle.
- **uplo** – **[in]** rocblas_fill.

  Specifies whether the factorization is upper or lower triangular. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.
- **n** – **[in]** rocblas_int. n >= 0.

  The dimension of matrix A_i.
- **A** – **[inout]** pointer to type. Array on the GPU (the size depends on the value of strideA).

  On entry, the matrices A_i to be factored. On exit, the upper or lower triangular factors.
- **lda** – **[in]** rocblas_int. lda >= n.

  specifies the leading dimension of A_i.
- **strideA** – **[in]** rocblas_stride.

  Stride from the start of one matrix A_i and the next one A_(i+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n.

- **info** – **[out]** pointer to rocblas_int. Array of batch_count integers on the GPU.

  If info_i = 0, successful factorization of matrix A_i. If info_i = j > 0, the leading minor of order j of A_i is not positive definite. The i-th factorization stopped at this point.

- **batch_count** – **[in]** rocblas_int. batch_count >= 0.

  Number of matrices in the batch.

rocsolver_<type>potrf()

rocblas_status **rocsolver_zpotrf**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, rocblas_double_complex *A, const rocblas_int lda, rocblas_int *info)

rocblas_status **rocsolver_cpotrf**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, rocblas_float_complex *A, const rocblas_int lda, rocblas_int *info)

rocblas_status **rocsolver_dpotrf**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, double *A, const rocblas_int lda, rocblas_int *info)

rocblas_status **rocsolver_spotrf**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, float *A, const rocblas_int lda, rocblas_int *info)

POTRF computes the Cholesky factorization of a real symmetric/complex Hermitian positive definite matrix A.

(This is the blocked version of the algorithm).

The factorization has the form:

```
A = U' * U, or
A = L  * L'
```

depending on the value of uplo. U is an upper triangular matrix and L is lower triangular.

Parameters

- **handle** – **[in]** rocblas_handle.
- **uplo** – **[in]** rocblas_fill.

  Specifies whether the factorization is upper or lower triangular. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.

- **n** – **[in]** rocblas_int. n >= 0.

  The matrix dimensions.

- **A** – **[inout]** pointer to type. Array on the GPU of dimension lda*n.

  On entry, the matrix A to be factored. On exit, the lower or upper triangular factor.

- **lda** – **[in]** rocblas_int. lda >= n.

  specifies the leading dimension of A.

- **info** – **[out]** pointer to a rocblas_int on the GPU.

  If info = 0, successful factorization of matrix A. If info = i > 0, the leading minor of order i of A is not positive definite. The factorization stopped at this point.

rocsolver_<type>potrf_batched()

rocblas_status **rocsolver_zpotrf_batched**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, rocblas_double_complex *const A[], const rocblas_int lda, rocblas_int *info, const rocblas_int batch_count)

rocblas_status **rocsolver_cpotrf_batched**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, rocblas_float_complex *const A[], const rocblas_int lda, rocblas_int *info, const rocblas_int batch_count)

rocblas_status **rocsolver_dpotrf_batched**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, double *const A[], const rocblas_int lda, rocblas_int *info, const rocblas_int batch_count)

rocblas_status **rocsolver_spotrf_batched**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, float *const A[], const rocblas_int lda, rocblas_int *info, const rocblas_int batch_count)

POTRF_BATCHED computes the Cholesky factorization of a batch of real symmetric/complex Hermitian positive definite matrices.

(This is the blocked version of the algorithm).

The factorization of matrix A_i in the batch has the form:

```
A_i = U_i' * U_i, or
A_i = L_i  * L_i'
```

depending on the value of uplo. U_i is an upper triangular matrix and L_i is lower triangular.

Parameters

- **handle** – **[in]** rocblas_handle.
- **uplo** – **[in]** rocblas_fill.

  Specifies whether the factorization is upper or lower triangular. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.
- **n** – **[in]** rocblas_int. n >= 0.

  The dimension of matrix A_i.
- **A** – **[inout]** array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n.

  On entry, the matrices A_i to be factored. On exit, the upper or lower triangular factors.
- **lda** – **[in]** rocblas_int. lda >= n.

  specifies the leading dimension of A_i.
- **info** – **[out]** pointer to rocblas_int. Array of batch_count integers on the GPU.

  If info_i = 0, successful factorization of matrix A_i. If info_i = j > 0, the leading minor of order j of A_i is not positive definite. The i-th factorization stopped at this point.
- **batch_count** – **[in]** rocblas_int. batch_count >= 0.

  Number of matrices in the batch.

rocsolver_<type>potrf_strided_batched()

rocblas_status **rocsolver_zpotrf_strided_batched**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, rocblas_double_complex *A, const rocblas_int lda, const rocblas_stride strideA, rocblas_int *info, const rocblas_int batch_count)

rocblas_status **rocsolver_cpotrf_strided_batched**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, rocblas_float_complex *A, const rocblas_int lda, const rocblas_stride strideA, rocblas_int *info, const rocblas_int batch_count)

rocblas_status **rocsolver_dpotrf_strided_batched**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, double *A, const rocblas_int lda, const rocblas_stride strideA, rocblas_int *info, const rocblas_int batch_count)

rocblas_status **rocsolver_spotrf_strided_batched**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, float *A, const rocblas_int lda, const rocblas_stride strideA, rocblas_int *info, const rocblas_int batch_count)

POTRF_STRIDED_BATCHED computes the Cholesky factorization of a batch of real symmetric/complex Hermitian positive definite matrices.

(This is the blocked version of the algorithm).

The factorization of matrix A_i in the batch has the form:

```
A_i = U_i' * U_i, or
A_i = L_i  * L_i'
```

depending on the value of uplo. U_i is an upper triangular matrix and L_i is lower triangular.

Parameters

- **handle** – **[in]** rocblas_handle.
- **uplo** – **[in]** rocblas_fill.

  Specifies whether the factorization is upper or lower triangular. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.
- **n** – **[in]** rocblas_int. n >= 0.

  The dimension of matrix A_i.
- **A** – **[inout]** pointer to type. Array on the GPU (the size depends on the value of strideA).

  On entry, the matrices A_i to be factored. On exit, the upper or lower triangular factors.
- **lda** – **[in]** rocblas_int. lda >= n.

  specifies the leading dimension of A_i.
- **strideA** – **[in]** rocblas_stride.

  Stride from the start of one matrix A_i and the next one A_(i+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n.

- **info** – **[out]** pointer to rocblas_int. Array of batch_count integers on the GPU.

  If info_i = 0, successful factorization of matrix A_i. If info_i = j > 0, the leading minor of order j of A_i is not positive definite. The i-th factorization stopped at this point.

- **batch_count** – **[in]** rocblas_int. batch_count >= 0.

  Number of matrices in the batch.

2.6.3.2 General Matrix Factorizations

rocsolver_<type>getf2()

rocblas_status **rocsolver_zgetf2**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_double_complex *A, const rocblas_int lda, rocblas_int *ipiv, rocblas_int *info)

rocblas_status **rocsolver_cgetf2**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_float_complex *A, const rocblas_int lda, rocblas_int *ipiv, rocblas_int *info)

rocblas_status **rocsolver_dgetf2**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, double *A, const rocblas_int lda, rocblas_int *ipiv, rocblas_int *info)

rocblas_status **rocsolver_sgetf2**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, float *A, const rocblas_int lda, rocblas_int *ipiv, rocblas_int *info)

GETF2 computes the LU factorization of a general m-by-n matrix A using partial pivoting with row interchanges.

(This is the unblocked Level-2-BLAS version of the algorithm. An optimized internal implementation without rocBLAS calls could be executed with small and mid-size matrices if optimizations are enabled (default option). For more details see the section "tuning rocSOLVER performance" on the User's guide).

The factorization has the form

```
A = P * L * U
```

where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if m > n), and U is upper triangular (upper trapezoidal if m < n).

Parameters

- **handle** – **[in]** rocblas_handle.
- **m** – **[in]** rocblas_int. m >= 0.

  The number of rows of the matrix A.

- **n** – **[in]** rocblas_int. n >= 0.

  The number of columns of the matrix A.

- **A** – **[inout]** pointer to type. Array on the GPU of dimension lda*n.

  On entry, the m-by-n matrix A to be factored. On exit, the factors L and U from the factorization. The unit diagonal elements of L are not stored.

- **lda** – **[in]** rocblas_int. lda >= m.

  Specifies the leading dimension of A.

- **ipiv** – **[out]** pointer to rocblas_int. Array on the GPU of dimension min(m,n).

  The vector of pivot indices. Elements of ipiv are 1-based indices. For 1 <= i <= min(m,n), the row i of the matrix was interchanged with row ipiv[i]. Matrix P of the factorization can be derived from ipiv.

- **info** – **[out]** pointer to a rocblas_int on the GPU.

  If info = 0, successful exit. If info = i > 0, U is singular. U(i,i) is the first zero pivot.

rocsolver_<type>getf2_batched()

rocblas_status **rocsolver_zgetf2_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_double_complex *const A[], const rocblas_int lda, rocblas_int *ipiv, const rocblas_stride strideP, rocblas_int *info, const rocblas_int batch_count)

rocblas_status **rocsolver_cgetf2_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_float_complex *const A[], const rocblas_int lda, rocblas_int *ipiv, const rocblas_stride strideP, rocblas_int *info, const rocblas_int batch_count)

rocblas_status **rocsolver_dgetf2_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, double *const A[], const rocblas_int lda, rocblas_int *ipiv, const rocblas_stride strideP, rocblas_int *info, const rocblas_int batch_count)

rocblas_status **rocsolver_sgetf2_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, float *const A[], const rocblas_int lda, rocblas_int *ipiv, const rocblas_stride strideP, rocblas_int *info, const rocblas_int batch_count)

GETF2_BATCHED computes the LU factorization of a batch of general m-by-n matrices using partial pivoting with row interchanges.

(This is the unblocked Level-2-BLAS version of the algorithm. An optimized internal implementation without rocBLAS calls could be executed with small and mid-size matrices if optimizations are enabled (default option). For more details see the section "tuning rocSOLVER performance" on the User's guide).

The factorization of matrix A_i in the batch has the form

```
A_i = P_i * L_i * U_i
```

where P_i is a permutation matrix, L_i is lower triangular with unit diagonal elements (lower trapezoidal if m > n), and U_i is upper triangular (upper trapezoidal if m < n).

Parameters

- **handle** – **[in]** rocblas_handle.
- **m** – **[in]** rocblas_int. m >= 0.

  The number of rows of all matrices A_i in the batch.

- **n** – **[in]** rocblas_int. n >= 0.

  The number of columns of all matrices A_i in the batch.

- **A** – **[inout]** array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n.

  On entry, the m-by-n matrices A_i to be factored. On exit, the factors L_i and U_i from the factorizations. The unit diagonal elements of L_i are not stored.

- **lda** – **[in]** rocblas_int. lda >= m.

  Specifies the leading dimension of matrices A_i.

- **ipiv** – **[out]** pointer to rocblas_int. Array on the GPU (the size depends on the value of strideP).

  Contains the vectors of pivot indices ipiv_i (corresponding to A_i). Dimension of ipiv_i is min(m,n). Elements of ipiv_i are 1-based indices. For each instance A_i in the batch and for 1 <= j <= min(m,n), the row j of the matrix A_i was interchanged with row ipiv_i[j]. Matrix P_i of the factorization can be derived from ipiv_i.

- **strideP** – **[in]** rocblas_stride.

  Stride from the start of one vector ipiv_i to the next one ipiv_(i+1). There is no restriction for the value of strideP. Normal use case is strideP >= min(m,n).

- **info** – **[out]** pointer to rocblas_int. Array of batch_count integers on the GPU.

  If info_i = 0, successful exit for factorization of A_i. If info_i = j > 0, U_i is singular. U_i(j,j) is the first zero pivot.

- **batch_count** – **[in]** rocblas_int. batch_count >= 0.

  Number of matrices in the batch.

rocsolver_<type>getf2_strided_batched()

rocblas_status **rocsolver_zgetf2_strided_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_double_complex *A, const rocblas_int lda, const rocblas_stride strideA, rocblas_int *ipiv, const rocblas_stride strideP, rocblas_int *info, const rocblas_int batch_count)

rocblas_status **rocsolver_cgetf2_strided_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_float_complex *A, const rocblas_int lda, const rocblas_stride strideA, rocblas_int *ipiv, const rocblas_stride strideP, rocblas_int *info, const rocblas_int batch_count)

rocblas_status **rocsolver_dgetf2_strided_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, double *A, const rocblas_int lda, const rocblas_stride strideA, rocblas_int *ipiv, const rocblas_stride strideP, rocblas_int *info, const rocblas_int batch_count)

rocblas_status **rocsolver_sgetf2_strided_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, float *A, const rocblas_int lda, const rocblas_stride strideA, rocblas_int *ipiv, const rocblas_stride strideP, rocblas_int *info, const rocblas_int batch_count)

GETF2_STRIDED_BATCHED computes the LU factorization of a batch of general m-by-n matrices using partial pivoting with row interchanges.

(This is the unblocked Level-2-BLAS version of the algorithm. An optimized internal implementation without rocBLAS calls could be executed with small and mid-size matrices if optimizations are enabled (default option). For more details see the section "tuning rocSOLVER performance" on the User's guide).

The factorization of matrix A_i in the batch has the form

```
A_i = P_i * L_i * U_i
```

where P_i is a permutation matrix, L_i is lower triangular with unit diagonal elements (lower trapezoidal if m > n), and U_i is upper triangular (upper trapezoidal if m < n).

Parameters

- **handle** – **[in]** rocblas_handle.
- **m** – **[in]** rocblas_int. m >= 0.

  The number of rows of all matrices A_i in the batch.
- **n** – **[in]** rocblas_int. n >= 0.

  The number of columns of all matrices A_i in the batch.
- **A** – **[inout]** pointer to type. Array on the GPU (the size depends on the value of strideA).

  On entry, the m-by-n matrices A_i to be factored. On exit, the factors L_i and U_i from the factorization. The unit diagonal elements of L_i are not stored.
- **lda** – **[in]** rocblas_int. lda >= m.

  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). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n
- **ipiv** – **[out]** pointer to rocblas_int. Array on the GPU (the size depends on the value of strideP).

  Contains the vectors of pivots indices ipiv_i (corresponding to A_i). Dimension of ipiv_i is min(m,n). Elements of ipiv_i are 1-based indices. For each instance A_i in the batch and for 1 <= j <= min(m,n), the row j of the matrix A_i was interchanged with row ipiv_i[j]. Matrix P_i of the factorization can be derived from ipiv_i.
- **strideP** – **[in]** rocblas_stride.

  Stride from the start of one vector ipiv_i to the next one ipiv_(i+1). There is no restriction for the value of strideP. Normal use case is strideP >= min(m,n).
- **info** – **[out]** pointer to rocblas_int. Array of batch_count integers on the GPU.

  If info_i = 0, successful exit for factorization of A_i. If info_i = j > 0, U_i is singular. U_i(j,j) is the first zero pivot.

- **batch_count** – **[in]** rocblas_int. batch_count >= 0.

  Number of matrices in the batch.

rocsolver_<type>getrf()

rocblas_status **rocsolver_zgetrf**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_double_complex *A, const rocblas_int lda, rocblas_int *ipiv, rocblas_int *info)

rocblas_status **rocsolver_cgetrf**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_float_complex *A, const rocblas_int lda, rocblas_int *ipiv, rocblas_int *info)

rocblas_status **rocsolver_dgetrf**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, double *A, const rocblas_int lda, rocblas_int *ipiv, rocblas_int *info)

rocblas_status **rocsolver_sgetrf**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, float *A, const rocblas_int lda, rocblas_int *ipiv, rocblas_int *info)

GETRF computes the LU factorization of a general m-by-n matrix A using partial pivoting with row interchanges.

(This is the blocked Level-3-BLAS version of the algorithm. An optimized internal implementation without rocBLAS calls could be executed with mid-size matrices if optimizations are enabled (default option). For more details see the section "tuning rocSOLVER performance" on the User's guide).

The factorization has the form

```
A = P * L * U
```

where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if m > n), and U is upper triangular (upper trapezoidal if m < n).

Parameters

- **handle** – **[in]** rocblas_handle.
- **m** – **[in]** rocblas_int. m >= 0.

  The number of rows of the matrix A.

- **n** – **[in]** rocblas_int. n >= 0.

  The number of columns of the matrix A.

- **A** – **[inout]** pointer to type. Array on the GPU of dimension lda*n.

  On entry, the m-by-n matrix A to be factored. On exit, the factors L and U from the factorization. The unit diagonal elements of L are not stored.

- **lda** – **[in]** rocblas_int. lda >= m.

  Specifies the leading dimension of A.

- **ipiv** – **[out]** pointer to rocblas_int. Array on the GPU of dimension min(m,n).

  The vector of pivot indices. Elements of ipiv are 1-based indices. For 1 <= i <= min(m,n), the row i of the matrix was interchanged with row ipiv[i]. Matrix P of the factorization can be derived from ipiv.

- **info** – **[out]** pointer to a rocblas_int on the GPU.

  If info = 0, successful exit. If info = i > 0, U is singular. U(i,i) is the first zero pivot.

rocsolver_<type>getrf_batched()

rocblas_status **rocsolver_zgetrf_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_double_complex *const A[], const rocblas_int lda, rocblas_int *ipiv, const rocblas_stride strideP, rocblas_int *info, const rocblas_int batch_count)

rocblas_status **rocsolver_cgetrf_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_float_complex *const A[], const rocblas_int lda, rocblas_int *ipiv, const rocblas_stride strideP, rocblas_int *info, const rocblas_int batch_count)

rocblas_status **rocsolver_dgetrf_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, double *const A[], const rocblas_int lda, rocblas_int *ipiv, const rocblas_stride strideP, rocblas_int *info, const rocblas_int batch_count)

rocblas_status **rocsolver_sgetrf_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, float *const A[], const rocblas_int lda, rocblas_int *ipiv, const rocblas_stride strideP, rocblas_int *info, const rocblas_int batch_count)

GETRF_BATCHED computes the LU factorization of a batch of general m-by-n matrices using partial pivoting with row interchanges.

(This is the blocked Level-3-BLAS version of the algorithm. An optimized internal implementation without rocBLAS calls could be executed with mid-size matrices if optimizations are enabled (default option). For more details see the section "tuning rocSOLVER performance" on the User's guide).

The factorization of matrix A_i in the batch has the form

```
A_i = P_i * L_i * U_i
```

where P_i is a permutation matrix, L_i is lower triangular with unit diagonal elements (lower trapezoidal if m > n), and U_i is upper triangular (upper trapezoidal if m < n).

Parameters

- **handle** – **[in]** rocblas_handle.
- **m** – **[in]** rocblas_int. m >= 0.

  The number of rows of all matrices A_i in the batch.
- **n** – **[in]** rocblas_int. n >= 0.

  The number of columns of all matrices A_i in the batch.
- **A** – **[inout]** array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n.

  On entry, the m-by-n matrices A_i to be factored. On exit, the factors L_i and U_i from the factorizations. The unit diagonal elements of L_i are not stored.

- **lda** – **[in]** rocblas_int. lda >= m.

  Specifies the leading dimension of matrices A_i.

- **ipiv** – **[out]** pointer to rocblas_int. Array on the GPU (the size depends on the value of strideP).

  Contains the vectors of pivot indices ipiv_i (corresponding to A_i). Dimension of ipiv_i is min(m,n). Elements of ipiv_i are 1-based indices. For each instance A_i in the batch and for 1 <= j <= min(m,n), the row j of the matrix A_i was interchanged with row ipiv_i(j). Matrix P_i of the factorization can be derived from ipiv_i.

- **strideP** – **[in]** rocblas_stride.

  Stride from the start of one vector ipiv_i to the next one ipiv_(i+1). There is no restriction for the value of strideP. Normal use case is strideP >= min(m,n).

- **info** – **[out]** pointer to rocblas_int. Array of batch_count integers on the GPU.

  If info_i = 0, successful exit for factorization of A_i. If info_i = j > 0, U_i is singular. U_i(j,j) is the first zero pivot.

- **batch_count** – **[in]** rocblas_int. batch_count >= 0.

  Number of matrices in the batch.

rocsolver_<type>getrf_strided_batched()

rocblas_status **rocsolver_zgetrf_strided_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_double_complex *A, const rocblas_int lda, const rocblas_stride strideA, rocblas_int *ipiv, const rocblas_stride strideP, rocblas_int *info, const rocblas_int batch_count)

rocblas_status **rocsolver_cgetrf_strided_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_float_complex *A, const rocblas_int lda, const rocblas_stride strideA, rocblas_int *ipiv, const rocblas_stride strideP, rocblas_int *info, const rocblas_int batch_count)

rocblas_status **rocsolver_dgetrf_strided_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, double *A, const rocblas_int lda, const rocblas_stride strideA, rocblas_int *ipiv, const rocblas_stride strideP, rocblas_int *info, const rocblas_int batch_count)

rocblas_status **rocsolver_sgetrf_strided_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, float *A, const rocblas_int lda, const rocblas_stride strideA, rocblas_int *ipiv, const rocblas_stride strideP, rocblas_int *info, const rocblas_int batch_count)

GETRF_STRIDED_BATCHED computes the LU factorization of a batch of general m-by-n matrices using partial pivoting with row interchanges.

(This is the blocked Level-3-BLAS version of the algorithm. An optimized internal implementation without rocBLAS calls could be executed with mid-size matrices if optimizations are enabled (default option). For more

details see the section "tuning rocSOLVER performance" on the User's guide).

The factorization of matrix A_i in the batch has the form

```
A_i = P_i * L_i * U_i
```

where P_i is a permutation matrix, L_i is lower triangular with unit diagonal elements (lower trapezoidal if m > n), and U_i is upper triangular (upper trapezoidal if m < n).

Parameters

- **handle** – **[in]** rocblas_handle.
- **m** – **[in]** rocblas_int. m >= 0.

  The number of rows of all matrices A_i in the batch.
- **n** – **[in]** rocblas_int. n >= 0.

  The number of columns of all matrices A_i in the batch.
- **A** – **[inout]** pointer to type. Array on the GPU (the size depends on the value of strideA).

  On entry, the m-by-n matrices A_i to be factored. On exit, the factors L_i and U_i from the factorization. The unit diagonal elements of L_i are not stored.
- **lda** – **[in]** rocblas_int. lda >= m.

  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). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n
- **ipiv** – **[out]** pointer to rocblas_int. Array on the GPU (the size depends on the value of strideP).

  Contains the vectors of pivots indices ipiv_i (corresponding to A_i). Dimension of ipiv_i is min(m,n). Elements of ipiv_i are 1-based indices. For each instance A_i in the batch and for 1 <= j <= min(m,n), the row j of the matrix A_i was interchanged with row ipiv_i(j). Matrix P_i of the factorization can be derived from ipiv_i.
- **strideP** – **[in]** rocblas_stride.

  Stride from the start of one vector ipiv_i to the next one ipiv_(i+1). There is no restriction for the value of strideP. Normal use case is strideP >= min(m,n).
- **info** – **[out]** pointer to rocblas_int. Array of batch_count integers on the GPU.

  If info_i = 0, successful exit for factorization of A_i. If info_i = j > 0, U_i is singular. U_i(j,j) is the first zero pivot.
- **batch_count** – **[in]** rocblas_int. batch_count >= 0.

  Number of matrices in the batch.

rocsolver_<type>geqr2()

rocblas_status **rocsolver_zgeqr2**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_double_complex *A, const rocblas_int lda, rocblas_double_complex *ipiv)

rocblas_status **rocsolver_cgeqr2**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_float_complex *A, const rocblas_int lda, rocblas_float_complex *ipiv)

rocblas_status **rocsolver_dgeqr2**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, double *A, const rocblas_int lda, double *ipiv)

rocblas_status **rocsolver_sgeqr2**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, float *A, const rocblas_int lda, float *ipiv)

GEQR2 computes a QR factorization of a general m-by-n matrix A.

(This is the unblocked version of the algorithm).

The factorization has the form

```
A =  Q * [ R ]
         [ 0 ]
```

where R is upper triangular (upper trapezoidal if m < n), and Q is a m-by-m orthogonal/unitary matrix represented as the product of Householder matrices

```
Q = H(1) * H(2) * ... * H(k), with k = min(m,n)
```

Each Householder matrix H(i), for i = 1,2,...,k, is given by

```
H(i) = I - ipiv[i-1] * v(i) * v(i)'
```

where the first i-1 elements of the Householder vector v(i) are zero, and v(i)[i] = 1.

Parameters

- **handle** – **[in]** rocblas_handle.
- **m** – **[in]** rocblas_int. m >= 0.

  The number of rows of the matrix A.
- **n** – **[in]** rocblas_int. n >= 0.

  The number of columns of the matrix A.
- **A** – **[inout]** pointer to type. Array on the GPU of dimension lda*n.

  On entry, the m-by-n matrix to be factored. On exit, the elements on and above the diagonal contain the factor R; the elements below the diagonal are the m - i elements of vector v(i) for i = 1,2,...,min(m,n).
- **lda** – **[in]** rocblas_int. lda >= m.

  Specifies the leading dimension of A.
- **ipiv** – **[out]** pointer to type. Array on the GPU of dimension min(m,n).

  The scalar factors of the Householder matrices H(i).

rocsolver_<type>geqr2_batched()

rocblas_status **rocsolver_zgeqr2_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_double_complex *const A[], const rocblas_int lda, rocblas_double_complex *ipiv, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_cgeqr2_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_float_complex *const A[], const rocblas_int lda, rocblas_float_complex *ipiv, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_dgeqr2_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, double *const A[], const rocblas_int lda, double *ipiv, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_sgeqr2_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, float *const A[], const rocblas_int lda, float *ipiv, const rocblas_stride strideP, const rocblas_int batch_count)

GEQR2_BATCHED computes the QR factorization of a batch of general m-by-n matrices.

(This is the unblocked version of the algorithm).

The factorization of matrix A_j in the batch has the form

```
A_j =  Q_j * [ R_j ]
             [  0  ]
```

where R_j is upper triangular (upper trapezoidal if m < n), and Q_j is a m-by-m orthogonal/unitary matrix represented as the product of Householder matrices

```
Q_j = H_j(1) * H_j(2) * ... * H_j(k), with k = min(m,n)
```

Each Householder matrices H_j(i), for j = 1,2,...,batch_count, and i = 1,2,...,k, is given by

```
H_j(i) = I - ipiv_j[i-1] * v_j(i) * v_j(i)'
```

where the first i-1 elements of Householder vector v_j(i) are zero, and v_j(i)[i] = 1.

Parameters

- **handle** – **[in]** rocblas_handle.
- **m** – **[in]** rocblas_int. m >= 0.

  The number of rows of all the matrices A_j in the batch.
- **n** – **[in]** rocblas_int. n >= 0.

  The number of columns of all the matrices A_j in the batch.
- **A** – **[inout]** Array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n.

  On entry, the m-by-n matrices A_j to be factored. On exit, the elements on and above the diagonal contain the factor R_j. The elements below the diagonal are the m - i elements of vector v_j(i) for i=1,2,...,min(m,n).

- **lda** – **[in]** rocblas_int. lda >= m.

  Specifies the leading dimension of matrices A_j.

- **ipiv** – **[out]** pointer to type. Array on the GPU (the size depends on the value of strideP).

  Contains the vectors ipiv_j of scalar factors of the Householder matrices H_j(i).

- **strideP** – **[in]** rocblas_stride.

  Stride from the start of one vector ipiv_j to the next one ipiv_(j+1). There is no restriction for the value of strideP. Normal use is strideP >= min(m,n).

- **batch_count** – **[in]** rocblas_int. batch_count >= 0.

  Number of matrices in the batch.

rocsolver_<type>geqr2_strided_batched()

rocblas_status **rocsolver_zgeqr2_strided_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_double_complex *A, const rocblas_int lda, const rocblas_stride strideA, rocblas_double_complex *ipiv, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_cgeqr2_strided_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_float_complex *A, const rocblas_int lda, const rocblas_stride strideA, rocblas_float_complex *ipiv, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_dgeqr2_strided_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, double *A, const rocblas_int lda, const rocblas_stride strideA, double *ipiv, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_sgeqr2_strided_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, float *A, const rocblas_int lda, const rocblas_stride strideA, float *ipiv, const rocblas_stride strideP, const rocblas_int batch_count)

GEQR2_STRIDED_BATCHED computes the QR factorization of a batch of general m-by-n matrices.

(This is the unblocked version of the algorithm).

The factorization of matrix A_j in the batch has the form

```
A_j =  Q_j * [ R_j ]
             [  0  ]
```

where R_j is upper triangular (upper trapezoidal if m < n), and Q_j is a m-by-m orthogonal/unitary matrix represented as the product of Householder matrices

```
Q_j = H_j(1) * H_j(2) * ... * H_j(k), with k = min(m,n)
```

Each Householder matrices H_j(i), for j = 1,2,...,batch_count, and i = 1,2,...,k, is given by

```
H_j(i) = I - ipiv_j[i-1] * v_j(i) * v_j(i)'
```

where the first i-1 elements of Householder vector v_j(i) are zero, and v_j(i)[i] = 1.

Parameters

- **handle** – **[in]** rocblas_handle.
- **m** – **[in]** rocblas_int. m >= 0.

  The number of rows of all the matrices A_j in the batch.
- **n** – **[in]** rocblas_int. n >= 0.

  The number of columns of all the matrices A_j in the batch.
- **A** – **[inout]** pointer to type. Array on the GPU (the size depends on the value of strideA).

  On entry, the m-by-n matrices A_j to be factored. On exit, the elements on and above the diagonal contain the factor R_j. The elements below the diagonal are the m - i elements of vector v_j(i) for i = 1,2,...,min(m,n).
- **lda** – **[in]** rocblas_int. lda >= m.

  Specifies the leading dimension of matrices A_j.
- **strideA** – **[in]** rocblas_stride.

  Stride from the start of one matrix A_j and the next one A_(j+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n.
- **ipiv** – **[out]** pointer to type. Array on the GPU (the size depends on the value of strideP).

  Contains the vectors ipiv_j of scalar factors of the Householder matrices H_j(i).
- **strideP** – **[in]** rocblas_stride.

  Stride from the start of one vector ipiv_j to the next one ipiv_(j+1). There is no restriction for the value of strideP. Normal use is strideP >= min(m,n).
- **batch_count** – **[in]** rocblas_int. batch_count >= 0.

  Number of matrices in the batch.

rocsolver_<type>geqrf()

rocblas_status **rocsolver_zgeqrf**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_double_complex *A, const rocblas_int lda, rocblas_double_complex *ipiv)

rocblas_status **rocsolver_cgeqrf**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_float_complex *A, const rocblas_int lda, rocblas_float_complex *ipiv)

rocblas_status **rocsolver_dgeqrf**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, double *A, const rocblas_int lda, double *ipiv)

rocblas_status **rocsolver_sgeqrf**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, float *A, const rocblas_int lda, float *ipiv)

GEQRF computes a QR factorization of a general m-by-n matrix A.

(This is the blocked version of the algorithm).

The factorization has the form

```
A =  Q * [ R ]
         [ 0 ]
```

where R is upper triangular (upper trapezoidal if m < n), and Q is a m-by-m orthogonal/unitary matrix represented as the product of Householder matrices

```
Q = H(1) * H(2) * ... * H(k), with k = min(m,n)
```

Each Householder matrix H(i), for i = 1,2,...,k, is given by

```
H(i) = I - ipiv[i-1] * v(i) * v(i)'
```

where the first i-1 elements of the Householder vector v(i) are zero, and v(i)[i] = 1.

Parameters

- **handle** – **[in]** rocblas_handle.
- **m** – **[in]** rocblas_int. m >= 0.

  The number of rows of the matrix A.
- **n** – **[in]** rocblas_int. n >= 0.

  The number of columns of the matrix A.
- **A** – **[inout]** pointer to type. Array on the GPU of dimension lda*n.

  On entry, the m-by-n matrix to be factored. On exit, the elements on and above the diagonal contain the factor R; the elements below the diagonal are the m - i elements of vector v(i) for i = 1,2,...,min(m,n).
- **lda** – **[in]** rocblas_int. lda >= m.

  Specifies the leading dimension of A.
- **ipiv** – **[out]** pointer to type. Array on the GPU of dimension min(m,n).

  The scalar factors of the Householder matrices H(i).

rocsolver_<type>geqrf_batched()

rocblas_status **rocsolver_zgeqrf_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_double_complex *const A[], const rocblas_int lda, rocblas_double_complex *ipiv, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_cgeqrf_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_float_complex *const A[], const rocblas_int lda, rocblas_float_complex *ipiv, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_dgeqrf_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, double *const A[], const rocblas_int lda, double *ipiv, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_sgeqrf_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, float *const A[], const rocblas_int lda, float *ipiv, const rocblas_stride strideP, const rocblas_int batch_count)

GEQRF_BATCHED computes the QR factorization of a batch of general m-by-n matrices.

(This is the blocked version of the algorithm).

The factorization of matrix A_j in the batch has the form

```
A_j =  Q_j * [ R_j ]
             [  0  ]
```

where R_j is upper triangular (upper trapezoidal if m < n), and Q_j is a m-by-m orthogonal/unitary matrix represented as the product of Householder matrices

```
Q_j = H_j(1) * H_j(2) * ... * H_j(k), with k = min(m,n)
```

Each Householder matrices H_j(i), for j = 1,2,...,batch_count, and i = 1,2,...,k, is given by

```
H_j(i) = I - ipiv_j[i-1] * v_j(i) * v_j(i)'
```

where the first i-1 elements of vector Householder vector v_j(i) are zero, and v_j(i)[i] = 1.

Parameters

- **handle** – **[in]** rocblas_handle.
- **m** – **[in]** rocblas_int. m >= 0.

  The number of rows of all the matrices A_j in the batch.
- **n** – **[in]** rocblas_int. n >= 0.

  The number of columns of all the matrices A_j in the batch.
- **A** – **[inout]** Array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n.

  On entry, the m-by-n matrices A_j to be factored. On exit, the elements on and above the diagonal contain the factor R_j. The elements below the diagonal are the m - i elements of vector v_j(i) for i=1,2,...,min(m,n).
- **lda** – **[in]** rocblas_int. lda >= m.

  Specifies the leading dimension of matrices A_j.
- **ipiv** – **[out]** pointer to type. Array on the GPU (the size depends on the value of strideP).

  Contains the vectors ipiv_j of scalar factors of the Householder matrices H_j(i).
- **strideP** – **[in]** rocblas_stride.

  Stride from the start of one vector ipiv_j to the next one ipiv_(j+1). There is no restriction for the value of strideP. Normal use is strideP >= min(m,n).
- **batch_count** – **[in]** rocblas_int. batch_count >= 0.

  Number of matrices in the batch.

rocsolver_<type>geqrf_strided_batched()

rocblas_status **rocsolver_zgeqrf_strided_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_double_complex *A, const rocblas_int lda, const rocblas_stride strideA, rocblas_double_complex *ipiv, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_cgeqrf_strided_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_float_complex *A, const rocblas_int lda, const rocblas_stride strideA, rocblas_float_complex *ipiv, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_dgeqrf_strided_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, double *A, const rocblas_int lda, const rocblas_stride strideA, double *ipiv, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_sgeqrf_strided_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, float *A, const rocblas_int lda, const rocblas_stride strideA, float *ipiv, const rocblas_stride strideP, const rocblas_int batch_count)

GEQRF_STRIDED_BATCHED computes the QR factorization of a batch of general m-by-n matrices.

(This is the blocked version of the algorithm).

The factorization of matrix A_j in the batch has the form

```
A_j =  Q_j * [ R_j ]
             [  0  ]
```

where R_j is upper triangular (upper trapezoidal if m < n), and Q_j is a m-by-m orthogonal/unitary matrix represented as the product of Householder matrices

```
Q_j = H_j(1) * H_j(2) * ... * H_j(k), with k = min(m,n)
```

Each Householder matrices H_j(i), for j = 1,2,...,batch_count, and i = 1,2,...,k, is given by

```
H_j(i) = I - ipiv_j[i-1] * v_j(i) * v_j(i)'
```

where the first i-1 elements of vector Householder vector v_j(i) are zero, and v_j(i)[i] = 1.

Parameters

- **handle** – **[in]** rocblas_handle.
- **m** – **[in]** rocblas_int. m >= 0.

  The number of rows of all the matrices A_j in the batch.
- **n** – **[in]** rocblas_int. n >= 0.

  The number of columns of all the matrices A_j in the batch.
- **A** – **[inout]** pointer to type. Array on the GPU (the size depends on the value of strideA).

On entry, the m-by-n matrices A_j to be factored. On exit, the elements on and above the diagonal contain the factor R_j. The elements below the diagonal are the m - i elements of vector v_j(i) for i = 1,2,...,min(m,n).

- **lda** – **[in]** rocblas_int. lda >= m.

  Specifies the leading dimension of matrices A_j.

- **strideA** – **[in]** rocblas_stride.

  Stride from the start of one matrix A_j and the next one A_(j+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n.

- **ipiv** – **[out]** pointer to type. Array on the GPU (the size depends on the value of strideP).

  Contains the vectors ipiv_j of scalar factors of the Householder matrices H_j(i).

- **strideP** – **[in]** rocblas_stride.

  Stride from the start of one vector ipiv_j to the next one ipiv_(j+1). There is no restriction for the value of strideP. Normal use is strideP >= min(m,n).

- **batch_count** – **[in]** rocblas_int. batch_count >= 0.

  Number of matrices in the batch.

rocsolver_<type>geql2()

rocblas_status **rocsolver_zgeql2**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_double_complex *A, const rocblas_int lda, rocblas_double_complex *ipiv)

rocblas_status **rocsolver_cgeql2**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_float_complex *A, const rocblas_int lda, rocblas_float_complex *ipiv)

rocblas_status **rocsolver_dgeql2**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, double *A, const rocblas_int lda, double *ipiv)

rocblas_status **rocsolver_sgeql2**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, float *A, const rocblas_int lda, float *ipiv)

GEQL2 computes a QL factorization of a general m-by-n matrix A.

(This is the unblocked version of the algorithm).

The factorization has the form

```
A =  Q * [ 0 ]
         [ L ]
```

where L is lower triangular (lower trapezoidal if m < n), and Q is a m-by-m orthogonal/unitary matrix represented as the product of Householder matrices

```
Q = H(k) * ... * H(2) * H(1), with k = min(m,n)
```

Each Householder matrix H(i), for i = 1,2,...,k, is given by

```
H(i) = I - ipiv[i-1] * v(i) * v(i)'
```

where the last m-i elements of the Householder vector v(i) are zero, and v(i)[i] = 1.

Parameters

- **handle** – **[in]** rocblas_handle.
- **m** – **[in]** rocblas_int. m >= 0.

  The number of rows of the matrix A.
- **n** – **[in]** rocblas_int. n >= 0.

  The number of columns of the matrix A.
- **A** – **[inout]** pointer to type. Array on the GPU of dimension lda*n.

  On entry, the m-by-n matrix to be factored. On exit, the elements on and below the (m-n)th subdiagonal (when m >= n) or the (n-m)th superdiagonal (when n > m) contain the factor L; the elements above the sub/superdiagonal are the i - 1 elements of vector v(i) for i = 1,2,...,min(m,n).
- **lda** – **[in]** rocblas_int. lda >= m.

  Specifies the leading dimension of A.
- **ipiv** – **[out]** pointer to type. Array on the GPU of dimension min(m,n).

  The scalar factors of the Householder matrices H(i).

rocsolver_<type>geql2_batched()

rocblas_status **rocsolver_zgeql2_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_double_complex *const A[], const rocblas_int lda, rocblas_double_complex *ipiv, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_cgeql2_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_float_complex *const A[], const rocblas_int lda, rocblas_float_complex *ipiv, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_dgeql2_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, double *const A[], const rocblas_int lda, double *ipiv, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_sgeql2_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, float *const A[], const rocblas_int lda, float *ipiv, const rocblas_stride strideP, const rocblas_int batch_count)

GEQL2_BATCHED computes the QL factorization of a batch of general m-by-n matrices.

(This is the unblocked version of the algorithm).

The factorization of matrix A_j in the batch has the form

```
A_j =  Q_j * [  0  ]
             [ L_j ]
```

where L_j is lower triangular (lower trapezoidal if m < n), and Q_j is a m-by-m orthogonal/unitary matrix represented as the product of Householder matrices

```
Q_j = H_j(k) * ... * H_j(2) * H_j(1), with k = min(m,n)
```

Each Householder matrices H_j(i), for j = 1,2,...,batch_count, and i = 1,2,...,k, is given by

```
H_j(i) = I - ipiv_j[i-1] * v_j(i) * v_j(i)'
```

where the last m-i elements of Householder vector v_j(i) are zero, and v_j(i)[i] = 1.

Parameters

- **handle** – **[in]** rocblas_handle.
- **m** – **[in]** rocblas_int. m >= 0.

  The number of rows of all the matrices A_j in the batch.
- **n** – **[in]** rocblas_int. n >= 0.

  The number of columns of all the matrices A_j in the batch.
- **A** – **[inout]** Array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n.

  On entry, the m-by-n matrices A_j to be factored. On exit, the elements on and below the (m-n)th subdiagonal (when m >= n) or the (n-m)th superdiagonal (when n > m) contain the factor L_j; the elements above the sub/superdiagonal are the i - 1 elements of vector v_j(i) for i = 1,2,...,min(m,n).
- **lda** – **[in]** rocblas_int. lda >= m.

  Specifies the leading dimension of matrices A_j.
- **ipiv** – **[out]** pointer to type. Array on the GPU (the size depends on the value of strideP).

  Contains the vectors ipiv_j of scalar factors of the Householder matrices H_j(i).
- **strideP** – **[in]** rocblas_stride.

  Stride from the start of one vector ipiv_j to the next one ipiv_(j+1). There is no restriction for the value of strideP. Normal use is strideP >= min(m,n).
- **batch_count** – **[in]** rocblas_int. batch_count >= 0.

  Number of matrices in the batch.

rocsolver_<type>geql2_strided_batched()

rocblas_status **rocsolver_zgeql2_strided_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_double_complex *A, const rocblas_int lda, const rocblas_stride strideA, rocblas_double_complex *ipiv, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_cgeql2_strided_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_float_complex *A, const rocblas_int lda, const rocblas_stride strideA, rocblas_float_complex *ipiv, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_dgeql2_strided_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, double *A, const rocblas_int lda, const rocblas_stride strideA, double *ipiv, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_sgeql2_strided_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, float *A, const rocblas_int lda, const rocblas_stride strideA, float *ipiv, const rocblas_stride strideP, const rocblas_int batch_count)

GEQL2_STRIDED_BATCHED computes the QL factorization of a batch of general m-by-n matrices.

(This is the unblocked version of the algorithm).

The factorization of matrix A_j in the batch has the form

```
A_j =  Q_j * [  0  ]
             [ L_j ]
```

where L_j is lower triangular (lower trapezoidal if m < n), and Q_j is a m-by-m orthogonal/unitary matrix represented as the product of Householder matrices

```
Q_j = H_j(k) * ... * H_j(2) * H_j(1), with k = min(m,n)
```

Each Householder matrices H_j(i), for j = 1,2,...,batch_count, and i = 1,2,...,k, is given by

```
H_j(i) = I - ipiv_j[i-1] * v_j(i) * v_j(i)'
```

where the last m-i elements of Householder vector v_j(i) are zero, and v_j(i)[i] = 1.

Parameters

- **handle** – **[in]** rocblas_handle.
- **m** – **[in]** rocblas_int. m >= 0.

  The number of rows of all the matrices A_j in the batch.
- **n** – **[in]** rocblas_int. n >= 0.

  The number of columns of all the matrices A_j in the batch.
- **A** – **[inout]** pointer to type. Array on the GPU (the size depends on the value of strideA).

  On entry, the m-by-n matrices A_j to be factored. On exit, the elements on and below the (m-n)th subdiagonal (when m >= n) or the (n-m)th superdiagonal (when n > m) contain the factor L_j; the elements above the sub/superdiagonal are the i - 1 elements of vector v_j(i) for i = 1,2,...,min(m,n).
- **lda** – **[in]** rocblas_int. lda >= m.

  Specifies the leading dimension of matrices A_j.
- **strideA** – **[in]** rocblas_stride.

  Stride from the start of one matrix A_j and the next one A_(j+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n.
- **ipiv** – **[out]** pointer to type. Array on the GPU (the size depends on the value of strideP).

  Contains the vectors ipiv_j of scalar factors of the Householder matrices H_j(i).

- **strideP** – **[in]** rocblas_stride.

  Stride from the start of one vector ipiv_j to the next one ipiv_(j+1). There is no restriction for the value of strideP. Normal use is strideP >= min(m,n).

- **batch_count** – **[in]** rocblas_int. batch_count >= 0.

  Number of matrices in the batch.

rocsolver_<type>geqlf()

rocblas_status **rocsolver_zgeqlf**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_double_complex *A, const rocblas_int lda, rocblas_double_complex *ipiv)

rocblas_status **rocsolver_cgeqlf**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_float_complex *A, const rocblas_int lda, rocblas_float_complex *ipiv)

rocblas_status **rocsolver_dgeqlf**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, double *A, const rocblas_int lda, double *ipiv)

rocblas_status **rocsolver_sgeqlf**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, float *A, const rocblas_int lda, float *ipiv)

GEQLF computes a QL factorization of a general m-by-n matrix A.

(This is the blocked version of the algorithm).

The factorization has the form

```
A =  Q * [ 0 ]
         [ L ]
```

where L is lower triangular (lower trapezoidal if m < n), and Q is a m-by-m orthogonal/unitary matrix represented as the product of Householder matrices

```
Q = H(k) * ... * H(2) * H(1), with k = min(m,n)
```

Each Householder matrix H(i), for i = 1,2,...,k, is given by

```
H(i) = I - ipiv[i-1] * v(i) * v(i)'
```

where the last m-i elements of the Householder vector v(i) are zero, and v(i)[i] = 1.

Parameters

- **handle** – **[in]** rocblas_handle.
- **m** – **[in]** rocblas_int. m >= 0.

  The number of rows of the matrix A.

- **n** – **[in]** rocblas_int. n >= 0.

  The number of columns of the matrix A.

- **A** – **[inout]** pointer to type. Array on the GPU of dimension lda*n.

  On entry, the m-by-n matrix to be factored. On exit, the elements on and below the (m-n)th subdiagonal (when m >= n) or the (n-m)th superdiagonal (when n > m) contain the factor

L; the elements above the sub/superdiagonal are the i - 1 elements of vector v(i) for i = 1,2,...,min(m,n).

- **lda** – **[in]** rocblas_int. lda >= m.

  Specifies the leading dimension of A.

- **ipiv** – **[out]** pointer to type. Array on the GPU of dimension min(m,n).

  The scalar factors of the Householder matrices H(i).

rocsolver_<type>geqlf_batched()

rocblas_status **rocsolver_zgeqlf_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_double_complex *const A[], const rocblas_int lda, rocblas_double_complex *ipiv, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_cgeqlf_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_float_complex *const A[], const rocblas_int lda, rocblas_float_complex *ipiv, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_dgeqlf_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, double *const A[], const rocblas_int lda, double *ipiv, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_sgeqlf_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, float *const A[], const rocblas_int lda, float *ipiv, const rocblas_stride strideP, const rocblas_int batch_count)

GEQLF_BATCHED computes the QL factorization of a batch of general m-by-n matrices.

(This is the blocked version of the algorithm).

The factorization of matrix A_j in the batch has the form

```
A_j =  Q_j * [  0  ]
             [ L_j ]
```

where L_j is lower triangular (lower trapezoidal if m < n), and Q_j is a m-by-m orthogonal/unitary matrix represented as the product of Householder matrices

```
Q_j = H_j(k) * ... * H_j(2) * H_j(1), with k = min(m,n)
```

Each Householder matrices H_j(i), for j = 1,2,...,batch_count, and i = 1,2,...,k, is given by

```
H_j(i) = I - ipiv_j[i-1] * v_j(i) * v_j(i)'
```

where the last m-i elements of vector Householder vector v_j(i) are zero, and v_j(i)[i] = 1.

Parameters

- **handle** – **[in]** rocblas_handle.
- **m** – **[in]** rocblas_int. m >= 0.

  The number of rows of all the matrices A_j in the batch.

- **n** – **[in]** rocblas_int. n >= 0.

  The number of columns of all the matrices A_j in the batch.

- **A** – **[inout]** Array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n.

  On entry, the m-by-n matrices A_j to be factored. On exit, the elements on and below the (m-n)th subdiagonal (when m >= n) or the (n-m)th superdiagonal (when n > m) contain the factor L_j; the elements above the sub/superdiagonal are the i - 1 elements of vector v_j(i) for i = 1,2,...,min(m,n).

- **lda** – **[in]** rocblas_int. lda >= m.

  Specifies the leading dimension of matrices A_j.

- **ipiv** – **[out]** pointer to type. Array on the GPU (the size depends on the value of strideP).

  Contains the vectors ipiv_j of scalar factors of the Householder matrices H_j(i).

- **strideP** – **[in]** rocblas_stride.

  Stride from the start of one vector ipiv_j to the next one ipiv_(j+1). There is no restriction for the value of strideP. Normal use is strideP >= min(m,n).

- **batch_count** – **[in]** rocblas_int. batch_count >= 0.

  Number of matrices in the batch.

rocsolver_<type>geqlf_strided_batched()

rocblas_status **rocsolver_zgeqlf_strided_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_double_complex *A, const rocblas_int lda, const rocblas_stride strideA, rocblas_double_complex *ipiv, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_cgeqlf_strided_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_float_complex *A, const rocblas_int lda, const rocblas_stride strideA, rocblas_float_complex *ipiv, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_dgeqlf_strided_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, double *A, const rocblas_int lda, const rocblas_stride strideA, double *ipiv, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_sgeqlf_strided_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, float *A, const rocblas_int lda, const rocblas_stride strideA, float *ipiv, const rocblas_stride strideP, const rocblas_int batch_count)

GEQLF_STRIDED_BATCHED computes the QL factorization of a batch of general m-by-n matrices.

(This is the blocked version of the algorithm).

The factorization of matrix A_j in the batch has the form

```
A_j =  Q_j * [  0  ]
             [ L_j ]
```

where L_j is lower triangular (lower trapezoidal if m < n), and Q_j is a m-by-m orthogonal/unitary matrix represented as the product of Householder matrices

```
Q_j = H_j(k) * ... * H_j(2) * H_j(1), with k = min(m,n)
```

Each Householder matrices H_j(i), for j = 1,2,...,batch_count, and i = 1,2,...,k, is given by

```
H_j(i) = I - ipiv_j[i-1] * v_j(i) * v_j(i)'
```

where the last m-i elements of vector Householder vector v_j(i) are zero, and v_j(i)[i] = 1.

Parameters

- **handle** – **[in]** rocblas_handle.
- **m** – **[in]** rocblas_int. m >= 0.

  The number of rows of all the matrices A_j in the batch.
- **n** – **[in]** rocblas_int. n >= 0.

  The number of columns of all the matrices A_j in the batch.
- **A** – **[inout]** pointer to type. Array on the GPU (the size depends on the value of strideA).

  On entry, the m-by-n matrices A_j to be factored. On exit, the elements on and below the (m-n)th subdiagonal (when m >= n) or the (n-m)th superdiagonal (when n > m) contain the factor L_j; the elements above the sub/superdiagonal are the i - 1 elements of vector v_j(i) for i = 1,2,...,min(m,n).
- **lda** – **[in]** rocblas_int. lda >= m.

  Specifies the leading dimension of matrices A_j.
- **strideA** – **[in]** rocblas_stride.

  Stride from the start of one matrix A_j and the next one A_(j+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n.
- **ipiv** – **[out]** pointer to type. Array on the GPU (the size depends on the value of strideP).

  Contains the vectors ipiv_j of scalar factors of the Householder matrices H_j(i).
- **strideP** – **[in]** rocblas_stride.

  Stride from the start of one vector ipiv_j to the next one ipiv_(j+1). There is no restriction for the value of strideP. Normal use is strideP >= min(m,n).
- **batch_count** – **[in]** rocblas_int. batch_count >= 0.

  Number of matrices in the batch.

rocsolver_<type>gelq2()

rocblas_status **rocsolver_zgelq2**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_double_complex *A, const rocblas_int lda, rocblas_double_complex *ipiv)

rocblas_status **rocsolver_cgelq2**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_float_complex *A, const rocblas_int lda, rocblas_float_complex *ipiv)

rocblas_status **rocsolver_dgelq2**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, double *A, const rocblas_int lda, double *ipiv)

rocblas_status **rocsolver_sgelq2**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, float *A, const rocblas_int lda, float *ipiv)

GELQ2 computes a LQ factorization of a general m-by-n matrix A.

(This is the unblocked version of the algorithm).

The factorization has the form

```
A = [ L 0 ] * Q
```

where L is lower triangular (lower trapezoidal if m > n), and Q is a n-by-n orthogonal/unitary matrix represented as the product of Householder matrices

```
Q = H(k) * H(k-1) * ... * H(1), with k = min(m,n)
```

Each Householder matrix H(i), for i = 1,2,...,k, is given by

```
H(i) = I - ipiv[i-1] * v(i)' * v(i)
```

where the first i-1 elements of the Householder vector v(i) are zero, and v(i)[i] = 1.

Parameters

- **handle** – **[in]** rocblas_handle.
- **m** – **[in]** rocblas_int. m >= 0.

  The number of rows of the matrix A.
- **n** – **[in]** rocblas_int. n >= 0.

  The number of columns of the matrix A.
- **A** – **[inout]** pointer to type. Array on the GPU of dimension lda*n.

  On entry, the m-by-n matrix to be factored. On exit, the elements on and below the diagonal contain the factor L; the elements above the diagonal are the n - i elements of vector v(i) for i = 1,2,...,min(m,n).
- **lda** – **[in]** rocblas_int. lda >= m.

  Specifies the leading dimension of A.
- **ipiv** – **[out]** pointer to type. Array on the GPU of dimension min(m,n).

  The scalar factors of the Householder matrices H(i).

rocsolver_<type>gelq2_batched()

rocblas_status **rocsolver_zgelq2_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_double_complex *const A[], const rocblas_int lda, rocblas_double_complex *ipiv, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_cgelq2_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_float_complex *const A[], const rocblas_int lda, rocblas_float_complex *ipiv, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_dgelq2_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, double *const A[], const rocblas_int lda, double *ipiv, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_sgelq2_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, float *const A[], const rocblas_int lda, float *ipiv, const rocblas_stride strideP, const rocblas_int batch_count)

GELQ2_BATCHED computes the LQ factorization of a batch of general m-by-n matrices.

(This is the unblocked version of the algorithm).

The factorization of matrix A_j in the batch has the form

```
A_j = [ L_j 0 ] * Q_j
```

where L_j is lower triangular (lower trapezoidal if m > n), and Q_j is a n-by-n orthogonal/unitary matrix represented as the product of Householder matrices

```
Q_j = H_j(k) * H_j(k-1) * ... * H_j(1), with k = min(m,n)
```

Each Householder matrices H_j(i), for j = 1,2,...,batch_count, and i = 1,2,...,k, is given by

```
H_j(i) = I - ipiv_j[i-1] * v_j(i)' * v_j(i)
```

where the first i-1 elements of Householder vector v_j(i) are zero, and v_j(i)[i] = 1.

Parameters

- **handle** – **[in]** rocblas_handle.
- **m** – **[in]** rocblas_int. m >= 0.

  The number of rows of all the matrices A_j in the batch.
- **n** – **[in]** rocblas_int. n >= 0.

  The number of columns of all the matrices A_j in the batch.
- **A** – **[inout]** Array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n.

  On entry, the m-by-n matrices A_j to be factored. On exit, the elements on and below the diagonal contain the factor L_j. The elements above the diagonal are the n - i elements of vector v_j(i) for i=1,2,...,min(m,n).

- **lda** – **[in]** rocblas_int. lda >= m.

  Specifies the leading dimension of matrices A_j.

- **ipiv** – **[out]** pointer to type. Array on the GPU (the size depends on the value of strideP).

  Contains the vectors ipiv_j of scalar factors of the Householder matrices H_j(i).

- **strideP** – **[in]** rocblas_stride.

  Stride from the start of one vector ipiv_j to the next one ipiv_(j+1). There is no restriction for the value of strideP. Normal use is strideP >= min(m,n).

- **batch_count** – **[in]** rocblas_int. batch_count >= 0.

  Number of matrices in the batch.

rocsolver_<type>gelq2_strided_batched()

rocblas_status **rocsolver_zgelq2_strided_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_double_complex *A, const rocblas_int lda, const rocblas_stride strideA, rocblas_double_complex *ipiv, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_cgelq2_strided_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_float_complex *A, const rocblas_int lda, const rocblas_stride strideA, rocblas_float_complex *ipiv, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_dgelq2_strided_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, double *A, const rocblas_int lda, const rocblas_stride strideA, double *ipiv, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_sgelq2_strided_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, float *A, const rocblas_int lda, const rocblas_stride strideA, float *ipiv, const rocblas_stride strideP, const rocblas_int batch_count)

GELQ2_STRIDED_BATCHED computes the LQ factorization of a batch of general m-by-n matrices.

(This is the unblocked version of the algorithm).

The factorization of matrix A_j in the batch has the form

```
A_j = [ L_j 0 ] * Q_j
```

where L_j is lower triangular (lower trapezoidal if m > n), and Q_j is a n-by-n orthogonal/unitary matrix represented as the product of Householder matrices

```
Q_j = H_j(k) * H_j(k-1) * ... * H_j(1), with k = min(m,n)
```

Each Householder matrices H_j(i), for j = 1,2,...,batch_count, and i = 1,2,...,k, is given by

```
H_j(i) = I - ipiv_j[i-1] * v_j(i)' * v_j(i)
```

where the first i-1 elements of vector Householder vector v_j(i) are zero, and v_j(i)[i] = 1.

Parameters

- **handle** – **[in]** rocblas_handle.
- **m** – **[in]** rocblas_int. m >= 0.

  The number of rows of all the matrices A_j in the batch.
- **n** – **[in]** rocblas_int. n >= 0.

  The number of columns of all the matrices A_j in the batch.
- **A** – **[inout]** pointer to type. Array on the GPU (the size depends on the value of strideA).

  On entry, the m-by-n matrices A_j to be factored. On exit, the elements on and below the diagonal contain the factor L_j. The elements above the diagonal are the n - i elements of vector v_j(i) for i = 1,2,...,min(m,n).
- **lda** – **[in]** rocblas_int. lda >= m.

  Specifies the leading dimension of matrices A_j.
- **strideA** – **[in]** rocblas_stride.

  Stride from the start of one matrix A_j and the next one A_(j+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n.
- **ipiv** – **[out]** pointer to type. Array on the GPU (the size depends on the value of strideP).

  Contains the vectors ipiv_j of scalar factors of the Householder matrices H_j(i).
- **strideP** – **[in]** rocblas_stride.

  Stride from the start of one vector ipiv_j to the next one ipiv_(j+1). There is no restriction for the value of strideP. Normal use is strideP >= min(m,n).
- **batch_count** – **[in]** rocblas_int. batch_count >= 0.

  Number of matrices in the batch.

rocsolver_<type>gelqf()

rocblas_status **rocsolver_zgelqf**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_double_complex *A, const rocblas_int lda, rocblas_double_complex *ipiv)

rocblas_status **rocsolver_cgelqf**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_float_complex *A, const rocblas_int lda, rocblas_float_complex *ipiv)

rocblas_status **rocsolver_dgelqf**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, double *A, const rocblas_int lda, double *ipiv)

rocblas_status **rocsolver_sgelqf**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, float *A, const rocblas_int lda, float *ipiv)

GELQF computes a LQ factorization of a general m-by-n matrix A.

(This is the blocked version of the algorithm).

The factorization has the form

```
A = [ L 0 ] * Q
```

where L is lower triangular (lower trapezoidal if m > n), and Q is a n-by-n orthogonal/unitary matrix represented as the product of Householder matrices

```
Q = H(k) * H(k-1) * ... * H(1), with k = min(m,n)
```

Each Householder matrix H(i), for i = 1,2,...,k, is given by

```
H(i) = I - ipiv[i-1] * v(i)' * v(i)
```

where the first i-1 elements of the Householder vector v(i) are zero, and v(i)[i] = 1.

Parameters

- **handle** – **[in]** rocblas_handle.
- **m** – **[in]** rocblas_int. m >= 0.

  The number of rows of the matrix A.
- **n** – **[in]** rocblas_int. n >= 0.

  The number of columns of the matrix A.
- **A** – **[inout]** pointer to type. Array on the GPU of dimension lda*n.

  On entry, the m-by-n matrix to be factored. On exit, the elements on and below the diagonal contain the factor L; the elements above the diagonal are the n - i elements of vector v(i) for i = 1,2,...,min(m,n).
- **lda** – **[in]** rocblas_int. lda >= m.

  Specifies the leading dimension of A.
- **ipiv** – **[out]** pointer to type. Array on the GPU of dimension min(m,n).

  The scalar factors of the Householder matrices H(i).

rocsolver_<type>gelqf_batched()

rocblas_status **rocsolver_zgelqf_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_double_complex *const A[], const rocblas_int lda, rocblas_double_complex *ipiv, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_cgelqf_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_float_complex *const A[], const rocblas_int lda, rocblas_float_complex *ipiv, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_dgelqf_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, double *const A[], const rocblas_int lda, double *ipiv, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_sgelqf_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, float *const A[], const rocblas_int lda, float *ipiv, const rocblas_stride strideP, const rocblas_int batch_count)

GELQF_BATCHED computes the LQ factorization of a batch of general m-by-n matrices.

(This is the blocked version of the algorithm).

The factorization of matrix A_j in the batch has the form

```
A_j = [ L_j 0 ] * Q_j
```

where L_j is lower triangular (lower trapezoidal if m > n), and Q_j is a n-by-n orthogonal/unitary matrix represented as the product of Householder matrices

```
Q_j = H_j(k) * H_j(k-1) * ... * H_j(1), with k = min(m,n)
```

Each Householder matrices H_j(i), for j = 1,2,...,batch_count, and i = 1,2,...,k, is given by

```
H_j(i) = I - ipiv_j[i-1] * v_j(i)' * v_j(i)
```

where the first i-1 elements of Householder vector v_j(i) are zero, and v_j(i)[i] = 1.

Parameters

- **handle** – **[in]** rocblas_handle.
- **m** – **[in]** rocblas_int. m >= 0.

  The number of rows of all the matrices A_j in the batch.
- **n** – **[in]** rocblas_int. n >= 0.

  The number of columns of all the matrices A_j in the batch.
- **A** – **[inout]** Array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n.

  On entry, the m-by-n matrices A_j to be factored. On exit, the elements on and below the diagonal contain the factor L_j. The elements above the diagonal are the n - i elements of vector v_j(i) for i=1,2,...,min(m,n).
- **lda** – **[in]** rocblas_int. lda >= m.

  Specifies the leading dimension of matrices A_j.
- **ipiv** – **[out]** pointer to type. Array on the GPU (the size depends on the value of strideP).

  Contains the vectors ipiv_j of scalar factors of the Householder matrices H_j(i).
- **strideP** – **[in]** rocblas_stride.

  Stride from the start of one vector ipiv_j to the next one ipiv_(j+1). There is no restriction for the value of strideP. Normal use is strideP >= min(m,n).
- **batch_count** – **[in]** rocblas_int. batch_count >= 0.

  Number of matrices in the batch.

rocsolver_<type>gelqf_strided_batched()

rocblas_status **rocsolver_zgelqf_strided_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_double_complex *A, const rocblas_int lda, const rocblas_stride strideA, rocblas_double_complex *ipiv, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_cgelqf_strided_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_float_complex *A, const rocblas_int lda, const rocblas_stride strideA, rocblas_float_complex *ipiv, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_dgelqf_strided_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, double *A, const rocblas_int lda, const rocblas_stride strideA, double *ipiv, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_sgelqf_strided_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, float *A, const rocblas_int lda, const rocblas_stride strideA, float *ipiv, const rocblas_stride strideP, const rocblas_int batch_count)

GELQF_STRIDED_BATCHED computes the LQ factorization of a batch of general m-by-n matrices.

(This is the blocked version of the algorithm).

The factorization of matrix A_j in the batch has the form

```
A_j = [ L_j 0 ] * Q_j
```

where L_j is lower triangular (lower trapezoidal if m > n), and Q_j is a n-by-n orthogonal/unitary matrix represented as the product of Householder matrices

```
Q_j = H_j(k) * H_j(k-1) * ... * H_j(1), with k = min(m,n)
```

Each Householder matrices H_j(i), for j = 1,2,...,batch_count, and i = 1,2,...,k, is given by

```
H_j(i) = I - ipiv_j[i-1] * v_j(i)' * v_j(i)
```

where the first i-1 elements of vector Householder vector v_j(i) are zero, and v_j(i)[i] = 1.

Parameters

- **handle** – **[in]** rocblas_handle.
- **m** – **[in]** rocblas_int. m >= 0.

  The number of rows of all the matrices A_j in the batch.
- **n** – **[in]** rocblas_int. n >= 0.

  The number of columns of all the matrices A_j in the batch.
- **A** – **[inout]** pointer to type. Array on the GPU (the size depends on the value of strideA).

On entry, the m-by-n matrices A_j to be factored. On exit, the elements on and below the diagonal contain the factor L_j. The elements above the diagonal are the n - i elements of vector v_j(i) for i = 1,2,...,min(m,n).

- **lda** – **[in]** rocblas_int. lda >= m.

  Specifies the leading dimension of matrices A_j.

- **strideA** – **[in]** rocblas_stride.

  Stride from the start of one matrix A_j and the next one A_(j+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n.

- **ipiv** – **[out]** pointer to type. Array on the GPU (the size depends on the value of strideP).

  Contains the vectors ipiv_j of scalar factors of the Householder matrices H_j(i).

- **strideP** – **[in]** rocblas_stride.

  Stride from the start of one vector ipiv_j to the next one ipiv_(j+1). There is no restriction for the value of strideP. Normal use is strideP >= min(m,n).

- **batch_count** – **[in]** rocblas_int. batch_count >= 0.

  Number of matrices in the batch.

rocsolver_<type>gels()

rocblas_status **rocsolver_zgels**(rocblas_handle handle, rocblas_operation trans, const rocblas_int m, const rocblas_int n, const rocblas_int nrhs, rocblas_double_complex *A, const rocblas_int lda, rocblas_double_complex *B, const rocblas_int ldb, rocblas_int *info)

rocblas_status **rocsolver_cgels**(rocblas_handle handle, rocblas_operation trans, const rocblas_int m, const rocblas_int n, const rocblas_int nrhs, rocblas_float_complex *A, const rocblas_int lda, rocblas_float_complex *B, const rocblas_int ldb, rocblas_int *info)

rocblas_status **rocsolver_dgels**(rocblas_handle handle, rocblas_operation trans, const rocblas_int m, const rocblas_int n, const rocblas_int nrhs, double *A, const rocblas_int lda, double *B, const rocblas_int ldb, rocblas_int *info)

rocblas_status **rocsolver_sgels**(rocblas_handle handle, rocblas_operation trans, const rocblas_int m, const rocblas_int n, const rocblas_int nrhs, float *A, const rocblas_int lda, float *B, const rocblas_int ldb, rocblas_int *info)

GELS solves an overdetermined (or underdetermined) linear system defined by an m-by-n matrix A, and a corresponding matrix B, using the QR factorization computed by GEQRF (or the LQ factorization computed by GELQF).

The problem solved by this function is either of the form

```
A  * X = B (no transpose), or
A' * X = B (transpose/conjugate transpose)
```

depending on the value of trans.

If m >= n (or n < m in the case of transpose/conjugate transpose), the system is overdetermined and a least-squares solution approximating X is found minimizing

```
|| B - A  * X || (no transpose), or
|| B - A' * X || (transpose/conjugate transpose)
```

If n < m (or m >= n in the case of transpose/conjugate transpose), the system is underdetermined and a unique solution for X is chosen minimizing || X ||

Note: The current implementation only supports the overdetermined, no transpose case. `rocblas_status_not_implemented` will be returned if m < n, or if trans is `rocblas_operation_transpose` or `rocblas_operation_conjugate_transpose`.

Parameters

- **handle** – **[in]** rocblas_handle.
- **trans** – **[in]** rocblas_operation.

  Specifies the form of the system of equations.
- **m** – **[in]** rocblas_int. m >= 0.

  The number of rows of matrix A.
- **n** – **[in]** rocblas_int. n >= 0.

  The number of columns of matrix A.
- **nrhs** – **[in]** rocblas_int. nrhs >= 0.

  The number of columns of matrices B and X; i.e., the columns on the right hand side.
- **A** – **[inout]** pointer to type. Array on the GPU of dimension lda*n.

  On entry, the m-by-n matrix A. On exit, the QR (or LQ) factorization of A as returned by GEQRF (or GELQF).
- **lda** – **[in]** rocblas_int. lda >= m.

  Specifies the leading dimension of matrix A.
- **B** – **[inout]** pointer to type. Array on the GPU of dimension ldb*nrhs.

  On entry, the matrix B is m-by-nrhs if non-transposed, or n-by-nrhs if transposed. On exit, when info = 0, B is overwritten by the solution vectors stored as columns.
- **ldb** – **[in]** rocblas_int. ldb >= max(m,n).

  Specifies the leading dimension of matrix B.
- **info** – **[out]** pointer to rocblas_int on the GPU.

  If info = 0, successful exit. If info = j > 0, the solution could not be computed because input matrix A is singular; the j-th diagonal element of its triangular factor is zero.

rocsolver_<type>gels_batched()

rocblas_status **rocsolver_zgels_batched**(rocblas_handle handle, rocblas_operation trans, const rocblas_int m, const rocblas_int n, const rocblas_int nrhs, rocblas_double_complex *const A[], const rocblas_int lda, rocblas_double_complex *const B[], const rocblas_int ldb, rocblas_int *info, const rocblas_int batch_count)

rocblas_status **rocsolver_cgels_batched**(rocblas_handle handle, rocblas_operation trans, const rocblas_int m, const rocblas_int n, const rocblas_int nrhs, rocblas_float_complex *const A[], const rocblas_int lda, rocblas_float_complex *const B[], const rocblas_int ldb, rocblas_int *info, const rocblas_int batch_count)

rocblas_status **rocsolver_dgels_batched**(rocblas_handle handle, rocblas_operation trans, const rocblas_int m, const rocblas_int n, const rocblas_int nrhs, double *const A[], const rocblas_int lda, double *const B[], const rocblas_int ldb, rocblas_int *info, const rocblas_int batch_count)

rocblas_status **rocsolver_sgels_batched**(rocblas_handle handle, rocblas_operation trans, const rocblas_int m, const rocblas_int n, const rocblas_int nrhs, float *const A[], const rocblas_int lda, float *const B[], const rocblas_int ldb, rocblas_int *info, const rocblas_int batch_count)

GELS_BATCHED solves batches of overdetermined (or underdetermined) linear systems defined by the array of m-by-n matrices A, and an array of corresponding matrices B, using the QR factorizations computed by GEQRF (or the LQ factorizations computed by GELQF).

The problem solved by this function is either of the form

```
A_i  * X_i = B_i (no transpose), or
A_i' * X_i = B_i (transpose/conjugate transpose)
```

depending on the value of trans.

If m >= n (or n < m in the case of transpose/conjugate transpose), the systems are overdetermined and least-squares solutions approximating X_i are found minimizing

```
|| B_i - A_i  * X_i || (no transpose), or
|| B_i - A_i' * X_i || (transpose/conjugate transpose)
```

If n < m (or m >= n in the case of transpose/conjugate transpose), the system is underdetermined and a unique solution for X_i is chosen minimizing || X_i ||

Note: The current implementation only supports the overdetermined, no transpose case. `rocblas_status_not_implemented` will be returned if m < n, or if trans is `rocblas_operation_transpose` or `rocblas_operation_conjugate_transpose`.

Parameters

- **handle** – **[in]** rocblas_handle.
- **trans** – **[in]** rocblas_operation.

  Specifies the form of the system of equations.

- **m** – **[in]** rocblas_int. m >= 0.

  The number of rows of all matrices A_i in the batch.

- **n** – **[in]** rocblas_int. n >= 0.

  The number of columns of all matrices A_i in the batch.

- **nrhs** – **[in]** rocblas_int. nrhs >= 0.

  The number of columns of all matrices B_i and X_i in the batch; i.e., the columns on the right hand side.

- **A** – **[inout]** array of pointer to type. Each pointer points to an array on the GPU of dimension lda*n.

  On entry, the m-by-n matrices A_i. On exit, the QR (or LQ) factorizations of A_i as returned by GEQRF (or GELQF).

- **lda** – **[in]** rocblas_int. lda >= m.

  Specifies the leading dimension of matrices A_i.

- **B** – **[inout]** array of pointer to type. Each pointer points to an array on the GPU of dimension ldb*nrhs.

  On entry, the matrices B_i are m-by-nrhs if non-transposed, or n-by-nrhs if transposed. On exit, when info = 0, each B_i is overwritten by the solution vectors stored as columns.

- **ldb** – **[in]** rocblas_int. ldb >= max(m,n).

  Specifies the leading dimension of matrices B_i.

- **info** – **[out]** pointer to rocblas_int. Array of batch_count integers on the GPU.

  If info_i = 0, successful exit for solution of A_i. If info_i = j > 0, the solution of A_i could not be computed because input matrix A_i is singular; the j-th diagonal element of its triangular factor is zero.

- **batch_count** – **[in]** rocblas_int. batch_count >= 0.

  Number of matrices in the batch.

rocsolver_<type>gels_strided_batched()

rocblas_status **rocsolver_zgels_strided_batched**(rocblas_handle handle, rocblas_operation trans, const rocblas_int m, const rocblas_int n, const rocblas_int nrhs, rocblas_double_complex *A, const rocblas_int lda, const rocblas_stride strideA, rocblas_double_complex *B, const rocblas_int ldb, const rocblas_stride strideB, rocblas_int *info, const rocblas_int batch_count)

rocblas_status **rocsolver_cgels_strided_batched**(rocblas_handle handle, rocblas_operation trans, const rocblas_int m, const rocblas_int n, const rocblas_int nrhs, rocblas_float_complex *A, const rocblas_int lda, const rocblas_stride strideA, rocblas_float_complex *B, const rocblas_int ldb, const rocblas_stride strideB, rocblas_int *info, const rocblas_int batch_count)

rocblas_status **rocsolver_dgels_strided_batched**(rocblas_handle handle, rocblas_operation trans, const rocblas_int m, const rocblas_int n, const rocblas_int nrhs, double *A, const rocblas_int lda, const rocblas_stride strideA, double *B, const rocblas_int ldb, const rocblas_stride strideB, rocblas_int *info, const rocblas_int batch_count)

rocblas_status **rocsolver_sgels_strided_batched**(rocblas_handle handle, rocblas_operation trans, const rocblas_int m, const rocblas_int n, const rocblas_int nrhs, float *A, const rocblas_int lda, const rocblas_stride strideA, float *B, const rocblas_int ldb, const rocblas_stride strideB, rocblas_int *info, const rocblas_int batch_count)

GELS_STRIDED_BATCHED solves batches of overdetermined (or underdetermined) linear systems defined by the array of m-by-n matrices A, and an array of corresponding matrices B, using the QR factorizations computed by GEQRF (or the LQ factorizations computed by GELQF).

The problem solved by this function is either of the form

```
A_i  * X_i = B_i (no transpose), or
A_i' * X_i = B_i (transpose/conjugate transpose)
```

depending on the value of trans.

If m >= n (or n < m in the case of transpose/conjugate transpose), the systems are overdetermined and least-squares solutions approximating X_i are found minimizing

```
|| B_i - A_i  * X_i || (no transpose), or
|| B_i - A_i' * X_i || (transpose/conjugate transpose)
```

If n < m (or m >= n in the case of transpose/conjugate transpose), the system is underdetermined and a unique solution for X_i is chosen minimizing || X_i ||

Note: The current implementation only supports the overdetermined, no transpose case. `rocblas_status_not_implemented` will be returned if m < n, or if trans is `rocblas_operation_transpose` or `rocblas_operation_conjugate_transpose`.

Parameters

- **handle** – **[in]** rocblas_handle.
- **trans** – **[in]** rocblas_operation.

  Specifies the form of the system of equations.
- **m** – **[in]** rocblas_int. m >= 0.

  The number of rows of all matrices A_i in the batch.
- **n** – **[in]** rocblas_int. n >= 0.

  The number of columns of all matrices A_i in the batch.
- **nrhs** – **[in]** rocblas_int. nrhs >= 0.

  The number of columns of all matrices B_i and X_i in the batch; i.e., the columns on the right hand side.

- **A** – **[inout]** pointer to type. Array on the GPU (the size depends on the value of strideA).

  On entry, the m-by-n matrices A_i. On exit, the QR (or LQ) factorizations of A_i as returned by GEQRF (or GELQF).

- **lda** – **[in]** rocblas_int. lda >= m.

  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). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n

- **B** – **[inout]** pointer to type. Array on the GPU (the size depends on the value of strideB).

  On entry, the matrices B_i are m-by-nrhs if non-transposed, or n-by-nrhs if transposed. On exit, when info = 0, each B_i is overwritten by the solution vectors stored as columns.

- **ldb** – **[in]** rocblas_int. ldb >= max(m,n).

  Specifies the leading dimension of matrices B_i.

- **strideB** – **[in]** rocblas_stride.

  Stride from the start of one matrix B_i and the next one B_(i+1). There is no restriction for the value of strideB. Normal use case is strideB >= ldb*nrhs

- **info** – **[out]** pointer to rocblas_int. Array of batch_count integers on the GPU.

  If info_i = 0, successful exit for solution of A_i. If info_i = j > 0, the solution of A_i could not be computed because input matrix A_i is singular; the j-th diagonal element of its triangular factor is zero.

- **batch_count** – **[in]** rocblas_int. batch_count >= 0.

  Number of matrices in the batch.

2.6.3.3 General Matrix Diagonalizations

rocsolver_<type>gebd2()

rocblas_status **rocsolver_zgebd2**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_double_complex *A, const rocblas_int lda, double *D, double *E, rocblas_double_complex *tauq, rocblas_double_complex *taup)

rocblas_status **rocsolver_cgebd2**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_float_complex *A, const rocblas_int lda, float *D, float *E, rocblas_float_complex *tauq, rocblas_float_complex *taup)

rocblas_status **rocsolver_dgebd2**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, double *A, const rocblas_int lda, double *D, double *E, double *tauq, double *taup)

rocblas_status **rocsolver_sgebd2**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, float *A, const rocblas_int lda, float *D, float *E, float *tauq, float *taup)

GEBD2 computes the bidiagonal form of a general m-by-n matrix A.

(This is the unblocked version of the algorithm).

The bidiagonal form is given by:

```
B = Q' * A * P
```

where B is upper bidiagonal if m >= n and lower bidiagonal if m < n, and Q and P are orthogonal/unitary matrices represented as the product of Householder matrices

```
Q = H(1) * H(2) * ... *  H(n)  and P = G(1) * G(2) * ... * G(n-1), if m >= n, or
Q = H(1) * H(2) * ... * H(m-1) and P = G(1) * G(2) * ... *  G(m),  if m < n
```

Each Householder matrix H(i) and G(i) is given by

```
H(i) = I - tauq[i-1] * v(i) * v(i)', and
G(i) = I - taup[i-1] * u(i) * u(i)'
```

If m >= n, the first i-1 elements of the Householder vector v(i) are zero, and v(i)[i] = 1; while the first i elements of the Householder vector u(i) are zero, and u(i)[i+1] = 1. If m < n, the first i elements of the Householder vector v(i) are zero, and v(i)[i+1] = 1; while the first i-1 elements of the Householder vector u(i) are zero, and u(i)[i] = 1.

Parameters

- **handle** – **[in]** rocblas_handle.
- **m** – **[in]** rocblas_int. m >= 0.

  The number of rows of the matrix A.
- **n** – **[in]** rocblas_int. n >= 0.

  The number of columns of the matrix A.
- **A** – **[inout]** pointer to type. Array on the GPU of dimension lda*n.

  On entry, the m-by-n matrix to be factored. On exit, the elements on the diagonal and superdiagonal (if m >= n), or subdiagonal (if m < n) contain the bidiagonal form B. If m >= n, the elements below the diagonal are the m - i elements of vector v(i) for i = 1,2,...,n, and the elements above the superdiagonal are the n - i - 1 elements of vector u(i) for i = 1,2,...,n-1. If m < n, the elements below the subdiagonal are the m - i - 1 elements of vector v(i) for i = 1,2,...,m-1, and the elements above the diagonal are the n - i elements of vector u(i) for i = 1,2,...,m.
- **lda** – **[in]** rocblas_int. lda >= m.

  specifies the leading dimension of A.
- **D** – **[out]** pointer to real type. Array on the GPU of dimension min(m,n).

  The diagonal elements of B.
- **E** – **[out]** pointer to real type. Array on the GPU of dimension min(m,n)-1.

  The off-diagonal elements of B.
- **tauq** – **[out]** pointer to type. Array on the GPU of dimension min(m,n).

  The scalar factors of the Householder matrices H(i).
- **taup** – **[out]** pointer to type. Array on the GPU of dimension min(m,n).

  The scalar factors of the Householder matrices G(i).

rocsolver_<type>gebd2_batched()

rocblas_status **rocsolver_zgebd2_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_double_complex *const A[], const rocblas_int lda, double *D, const rocblas_stride strideD, double *E, const rocblas_stride strideE, rocblas_double_complex *tauq, const rocblas_stride strideQ, rocblas_double_complex *taup, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_cgebd2_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_float_complex *const A[], const rocblas_int lda, float *D, const rocblas_stride strideD, float *E, const rocblas_stride strideE, rocblas_float_complex *tauq, const rocblas_stride strideQ, rocblas_float_complex *taup, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_dgebd2_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, double *const A[], const rocblas_int lda, double *D, const rocblas_stride strideD, double *E, const rocblas_stride strideE, double *tauq, const rocblas_stride strideQ, double *taup, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_sgebd2_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, float *const A[], const rocblas_int lda, float *D, const rocblas_stride strideD, float *E, const rocblas_stride strideE, float *tauq, const rocblas_stride strideQ, float *taup, const rocblas_stride strideP, const rocblas_int batch_count)

GEBD2_BATCHED computes the bidiagonal form of a batch of general m-by-n matrices.

(This is the unblocked version of the algorithm).

The bidiagonal form is given by:

```
B_j = Q_j' * A_j * P_j
```

where B_j is upper bidiagonal if m >= n and lower bidiagonal if m < n, and Q_j and P_j are orthogonal/unitary matrices represented as the product of Householder matrices

```
Q_j = H_j(1) * H_j(2) * ... *  H_j(n)  and P_j = G_j(1) * G_j(2) * ... * G_j(n-1), ␣
↪if m >= n, or
Q_j = H_j(1) * H_j(2) * ... * H_j(m-1) and P_j = G_j(1) * G_j(2) * ... *  G_j(m), ␣
↪if m < n
```

Each Householder matrix H_j(i) and G_j(i), for j = 1,2,...,batch_count, is given by

```
H_j(i) = I - tauq_j[i-1] * v_j(i) * v_j(i)', and
G_j(i) = I - taup_j[i-1] * u_j(i) * u_j(i)'
```

If m >= n, the first i-1 elements of the Householder vector v_j(i) are zero, and v_j(i)[i] = 1; while the first i elements of the Householder vector u_j(i) are zero, and u_j(i)[i+1] = 1. If m < n, the first i elements of the Householder vector v_j(i) are zero, and v_j(i)[i+1] = 1; while the first i-1 elements of the Householder vector u_j(i) are zero, and u_j(i)[i] = 1.

Parameters

- **handle** – **[in]** rocblas_handle.
- **m** – **[in]** rocblas_int. m >= 0.

  The number of rows of all the matrices A_j in the batch.
- **n** – **[in]** rocblas_int. n >= 0.

  The number of columns of all the matrices A_j in the batch.
- **A** – **[inout]** Array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n.

  On entry, the m-by-n matrices A_j to be factored. On exit, the elements on the diagonal and superdiagonal (if m >= n), or subdiagonal (if m < n) contain the bidiagonal form B_j. If m >= n, the elements below the diagonal are the m - i elements of vector v_j(i) for i = 1,2,...,n, and the elements above the superdiagonal are the n - i - 1 elements of vector u_j(i) for i = 1,2,...,n-1. If m < n, the elements below the subdiagonal are the m - i - 1 elements of vector v_j(i) for i = 1,2,...,m-1, and the elements above the diagonal are the n - i elements of vector u_j(i) for i = 1,2,...,m.
- **lda** – **[in]** rocblas_int. lda >= m.

  Specifies the leading dimension of matrices A_j.
- **D** – **[out]** pointer to real type. Array on the GPU (the size depends on the value of strideD).

  The diagonal elements of B_j.
- **strideD** – **[in]** rocblas_stride.

  Stride from the start of one vector D_j and the next one D_(j+1). There is no restriction for the value of strideD. Normal use case is strideD >= min(m,n).
- **E** – **[out]** pointer to real type. Array on the GPU (the size depends on the value of strideE).

  The off-diagonal elements of B_j.
- **strideE** – **[in]** rocblas_stride.

  Stride from the start of one vector E_j and the next one E_(j+1). There is no restriction for the value of strideE. Normal use case is strideE >= min(m,n)-1.
- **tauq** – **[out]** pointer to type. Array on the GPU (the size depends on the value of strideQ).

  Contains the vectors tauq_j of scalar factors of the Householder matrices H_j(i).
- **strideQ** – **[in]** rocblas_stride.

  Stride from the start of one vector tauq_j to the next one tauq_(j+1). There is no restriction for the value of strideQ. Normal use is strideQ >= min(m,n).
- **taup** – **[out]** pointer to type. Array on the GPU (the size depends on the value of strideP).

  Contains the vectors taup_j of scalar factors of the Householder matrices G_j(i).
- **strideP** – **[in]** rocblas_stride.

  Stride from the start of one vector taup_j to the next one taup_(j+1). There is no restriction for the value of strideP. Normal use is strideP >= min(m,n).
- **batch_count** – **[in]** rocblas_int. batch_count >= 0.

  Number of matrices in the batch.

rocsolver_<type>gebd2_strided_batched()

rocblas_status **rocsolver_zgebd2_strided_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_double_complex *A, const rocblas_int lda, const rocblas_stride strideA, double *D, const rocblas_stride strideD, double *E, const rocblas_stride strideE, rocblas_double_complex *tauq, const rocblas_stride strideQ, rocblas_double_complex *taup, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_cgebd2_strided_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_float_complex *A, const rocblas_int lda, const rocblas_stride strideA, float *D, const rocblas_stride strideD, float *E, const rocblas_stride strideE, rocblas_float_complex *tauq, const rocblas_stride strideQ, rocblas_float_complex *taup, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_dgebd2_strided_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, double *A, const rocblas_int lda, const rocblas_stride strideA, double *D, const rocblas_stride strideD, double *E, const rocblas_stride strideE, double *tauq, const rocblas_stride strideQ, double *taup, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_sgebd2_strided_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, float *A, const rocblas_int lda, const rocblas_stride strideA, float *D, const rocblas_stride strideD, float *E, const rocblas_stride strideE, float *tauq, const rocblas_stride strideQ, float *taup, const rocblas_stride strideP, const rocblas_int batch_count)

GEBD2_STRIDED_BATCHED computes the bidiagonal form of a batch of general m-by-n matrices.

(This is the unblocked version of the algorithm).

The bidiagonal form is given by:

```
B_j = Q_j' * A_j * P_j
```

where B_j is upper bidiagonal if m >= n and lower bidiagonal if m < n, and Q_j and P_j are orthogonal/unitary matrices represented as the product of Householder matrices

```
Q_j = H_j(1) * H_j(2) * ... *  H_j(n)  and P_j = G_j(1) * G_j(2) * ... * G_j(n-1),␣
↪if m >= n, or
Q_j = H_j(1) * H_j(2) * ... * H_j(m-1) and P_j = G_j(1) * G_j(2) * ... *  G_j(m), ␣
↪if m < n
```

Each Householder matrix H_j(i) and G_j(i), for j = 1,2,...,batch_count, is given by

```
H_j(i) = I - tauq_j[i-1] * v_j(i) * v_j(i)', and
G_j(i) = I - taup_j[i-1] * u_j(i) * u_j(i)'
```

If m >= n, the first i-1 elements of the Householder vector v_j(i) are zero, and v_j(i)[i] = 1; while the first i elements of the Householder vector u_j(i) are zero, and u_j(i)[i+1] = 1. If m < n, the first i elements of the Householder vector v_j(i) are zero, and v_j(i)[i+1] = 1; while the first i-1 elements of the Householder vector u_j(i) are zero, and u_j(i)[i] = 1.

Parameters

- **handle** – **[in]** rocblas_handle.
- **m** – **[in]** rocblas_int. m >= 0.

  The number of rows of all the matrices A_j in the batch.
- **n** – **[in]** rocblas_int. n >= 0.

  The number of columns of all the matrices A_j in the batch.
- **A** – **[inout]** pointer to type. Array on the GPU (the size depends on the value of strideA).

  On entry, the m-by-n matrices A_j to be factored. On exit, the elements on the diagonal and superdiagonal (if m >= n), or subdiagonal (if m < n) contain the bidiagonal form B_j. If m >= n, the elements below the diagonal are the m - i elements of vector v_j(i) for i = 1,2,...,n, and the elements above the superdiagonal are the n - i - 1 elements of vector u_j(i) for i = 1,2,...,n-1. If m < n, the elements below the subdiagonal are the m - i - 1 elements of vector v_j(i) for i = 1,2,...,m-1, and the elements above the diagonal are the n - i elements of vector u_j(i) for i = 1,2,...,m.
- **lda** – **[in]** rocblas_int. lda >= m.

  Specifies the leading dimension of matrices A_j.
- **strideA** – **[in]** rocblas_stride.

  Stride from the start of one matrix A_j and the next one A_(j+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n.
- **D** – **[out]** pointer to real type. Array on the GPU (the size depends on the value of strideD).

  The diagonal elements of B_j.
- **strideD** – **[in]** rocblas_stride.

  Stride from the start of one vector D_j and the next one D_(j+1). There is no restriction for the value of strideD. Normal use case is strideD >= min(m,n).
- **E** – **[out]** pointer to real type. Array on the GPU (the size depends on the value of strideE).

  The off-diagonal elements of B_j.
- **strideE** – **[in]** rocblas_stride.

  Stride from the start of one vector E_j and the next one E_(j+1). There is no restriction for the value of strideE. Normal use case is strideE >= min(m,n)-1.
- **tauq** – **[out]** pointer to type. Array on the GPU (the size depends on the value of strideQ).

  Contains the vectors tauq_j of scalar factors of the Householder matrices H_j(i).
- **strideQ** – **[in]** rocblas_stride.

  Stride from the start of one vector tauq_j to the next one tauq_(j+1). There is no restriction for the value of strideQ. Normal use is strideQ >= min(m,n).
- **taup** – **[out]** pointer to type. Array on the GPU (the size depends on the value of strideP).

  Contains the vectors taup_j of scalar factors of the Householder matrices G_j(i).

- **strideP** – **[in]** rocblas_stride.

  Stride from the start of one vector taup_j to the next one taup_(j+1). There is no restriction for the value of strideP. Normal use is strideP >= min(m,n).

- **batch_count** – **[in]** rocblas_int. batch_count >= 0.

  Number of matrices in the batch.

rocsolver_<type>gebrd()

rocblas_status **rocsolver_zgebrd**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_double_complex *A, const rocblas_int lda, double *D, double *E, rocblas_double_complex *tauq, rocblas_double_complex *taup)

rocblas_status **rocsolver_cgebrd**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_float_complex *A, const rocblas_int lda, float *D, float *E, rocblas_float_complex *tauq, rocblas_float_complex *taup)

rocblas_status **rocsolver_dgebrd**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, double *A, const rocblas_int lda, double *D, double *E, double *tauq, double *taup)

rocblas_status **rocsolver_sgebrd**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, float *A, const rocblas_int lda, float *D, float *E, float *tauq, float *taup)

GEBRD computes the bidiagonal form of a general m-by-n matrix A.

(This is the blocked version of the algorithm).

The bidiagonal form is given by:

```
B = Q' * A * P
```

where B is upper bidiagonal if m >= n and lower bidiagonal if m < n, and Q and P are orthogonal/unitary matrices represented as the product of Householder matrices

```
Q = H(1) * H(2) * ... *  H(n)  and P = G(1) * G(2) * ... * G(n-1), if m >= n, or
Q = H(1) * H(2) * ... * H(m-1) and P = G(1) * G(2) * ... *  G(m),  if m < n
```

Each Householder matrix H(i) and G(i) is given by

```
H(i) = I - tauq[i-1] * v(i) * v(i)', and
G(i) = I - taup[i-1] * u(i) * u(i)'
```

If m >= n, the first i-1 elements of the Householder vector v(i) are zero, and v(i)[i] = 1; while the first i elements of the Householder vector u(i) are zero, and u(i)[i+1] = 1. If m < n, the first i elements of the Householder vector v(i) are zero, and v(i)[i+1] = 1; while the first i-1 elements of the Householder vector u(i) are zero, and u(i)[i] = 1.

Parameters

- **handle** – **[in]** rocblas_handle.
- **m** – **[in]** rocblas_int. m >= 0.

  The number of rows of the matrix A.

- **n** – **[in]** rocblas_int. n >= 0.

  The number of columns of the matrix A.

- **A** – **[inout]** pointer to type. Array on the GPU of dimension lda*n.

  On entry, the m-by-n matrix to be factored. On exit, the elements on the diagonal and superdiagonal (if m >= n), or subdiagonal (if m < n) contain the bidiagonal form B. If m >= n, the elements below the diagonal are the m - i elements of vector v(i) for i = 1,2,...,n, and the elements above the superdiagonal are the n - i - 1 elements of vector u(i) for i = 1,2,...,n-1. If m < n, the elements below the subdiagonal are the m - i - 1 elements of vector v(i) for i = 1,2,...,m-1, and the elements above the diagonal are the n - i elements of vector u(i) for i = 1,2,...,m.

- **lda** – **[in]** rocblas_int. lda >= m.

  specifies the leading dimension of A.

- **D** – **[out]** pointer to real type. Array on the GPU of dimension min(m,n).

  The diagonal elements of B.

- **E** – **[out]** pointer to real type. Array on the GPU of dimension min(m,n)-1.

  The off-diagonal elements of B.

- **tauq** – **[out]** pointer to type. Array on the GPU of dimension min(m,n).

  The scalar factors of the Householder matrices H(i).

- **taup** – **[out]** pointer to type. Array on the GPU of dimension min(m,n).

  The scalar factors of the Householder matrices G(i).

rocsolver_<type>gebrd_batched()

rocblas_status **rocsolver_zgebrd_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_double_complex *const A[], const rocblas_int lda, double *D, const rocblas_stride strideD, double *E, const rocblas_stride strideE, rocblas_double_complex *tauq, const rocblas_stride strideQ, rocblas_double_complex *taup, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_cgebrd_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_float_complex *const A[], const rocblas_int lda, float *D, const rocblas_stride strideD, float *E, const rocblas_stride strideE, rocblas_float_complex *tauq, const rocblas_stride strideQ, rocblas_float_complex *taup, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_dgebrd_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, double *const A[], const rocblas_int lda, double *D, const rocblas_stride strideD, double *E, const rocblas_stride strideE, double *tauq, const rocblas_stride strideQ, double *taup, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_sgebrd_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, float *const A[], const rocblas_int lda, float *D, const rocblas_stride strideD, float *E, const rocblas_stride strideE, float *tauq, const rocblas_stride strideQ, float *taup, const rocblas_stride strideP, const rocblas_int batch_count)

GEBRD_BATCHED computes the bidiagonal form of a batch of general m-by-n matrices.

(This is the blocked version of the algorithm).

The bidiagonal form is given by:

```
B_j = Q_j' * A_j * P_j
```

where B_j is upper bidiagonal if m >= n and lower bidiagonal if m < n, and Q_j and P_j are orthogonal/unitary matrices represented as the product of Householder matrices

```
Q_j = H_j(1) * H_j(2) * ... *  H_j(n)  and P_j = G_j(1) * G_j(2) * ... * G_j(n-1), ␣
↪if m >= n, or
Q_j = H_j(1) * H_j(2) * ... * H_j(m-1) and P_j = G_j(1) * G_j(2) * ... *  G_j(m), ␣
↪if m < n
```

Each Householder matrix H_j(i) and G_j(i), for j = 1,2,...,batch_count, is given by

```
H_j(i) = I - tauq_j[i-1] * v_j(i) * v_j(i)', and
G_j(i) = I - taup_j[i-1] * u_j(i) * u_j(i)'
```

If m >= n, the first i-1 elements of the Householder vector v_j(i) are zero, and v_j(i)[i] = 1; while the first i elements of the Householder vector u_j(i) are zero, and u_j(i)[i+1] = 1. If m < n, the first i elements of the Householder vector v_j(i) are zero, and v_j(i)[i+1] = 1; while the first i-1 elements of the Householder vector u_j(i) are zero, and u_j(i)[i] = 1.

Parameters

- **handle** – **[in]** rocblas_handle.
- **m** – **[in]** rocblas_int. m >= 0.

  The number of rows of all the matrices A_j in the batch.
- **n** – **[in]** rocblas_int. n >= 0.

  The number of columns of all the matrices A_j in the batch.
- **A** – **[inout]** Array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n.

  On entry, the m-by-n matrices A_j to be factored. On exit, the elements on the diagonal and superdiagonal (if m >= n), or subdiagonal (if m < n) contain the bidiagonal form B_j. If m >= n, the elements below the diagonal are the m - i elements of vector v_j(i) for i = 1,2,...,n, and the elements above the superdiagonal are the n - i - 1 elements of vector u_j(i) for i = 1,2,...,n-1. If m < n, the elements below the subdiagonal are the m - i - 1 elements of vector v_j(i) for i = 1,2,...,m-1, and the elements above the diagonal are the n - i elements of vector u_j(i) for i = 1,2,...,m.
- **lda** – **[in]** rocblas_int. lda >= m.

  Specifies the leading dimension of matrices A_j.
- **D** – **[out]** pointer to real type. Array on the GPU (the size depends on the value of strideD).

  The diagonal elements of B_j.

- **strideD** – **[in]** rocblas_stride.

  Stride from the start of one vector D_j and the next one D_(j+1). There is no restriction for the value of strideD. Normal use case is strideD >= min(m,n).

- **E** – **[out]** pointer to real type. Array on the GPU (the size depends on the value of strideE).

  The off-diagonal elements of B_j.

- **strideE** – **[in]** rocblas_stride.

  Stride from the start of one vector E_j and the next one E_(j+1). There is no restriction for the value of strideE. Normal use case is strideE >= min(m,n)-1.

- **tauq** – **[out]** pointer to type. Array on the GPU (the size depends on the value of strideQ).

  Contains the vectors tauq_j of scalar factors of the Householder matrices H_j(i).

- **strideQ** – **[in]** rocblas_stride.

  Stride from the start of one vector tauq_j to the next one tauq_(j+1). There is no restriction for the value of strideQ. Normal use is strideQ >= min(m,n).

- **taup** – **[out]** pointer to type. Array on the GPU (the size depends on the value of strideP).

  Contains the vectors taup_j of scalar factors of the Householder matrices G_j(i).

- **strideP** – **[in]** rocblas_stride.

  Stride from the start of one vector taup_j to the next one taup_(j+1). There is no restriction for the value of strideP. Normal use is strideP >= min(m,n).

- **batch_count** – **[in]** rocblas_int. batch_count >= 0.

  Number of matrices in the batch.

rocsolver_<type>gebrd_strided_batched()

rocblas_status **rocsolver_zgebrd_strided_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_double_complex *A, const rocblas_int lda, const rocblas_stride strideA, double *D, const rocblas_stride strideD, double *E, const rocblas_stride strideE, rocblas_double_complex *tauq, const rocblas_stride strideQ, rocblas_double_complex *taup, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_cgebrd_strided_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_float_complex *A, const rocblas_int lda, const rocblas_stride strideA, float *D, const rocblas_stride strideD, float *E, const rocblas_stride strideE, rocblas_float_complex *tauq, const rocblas_stride strideQ, rocblas_float_complex *taup, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_dgebrd_strided_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, double *A, const rocblas_int lda, const rocblas_stride strideA, double *D, const rocblas_stride strideD, double *E, const rocblas_stride strideE, double *tauq, const rocblas_stride strideQ, double *taup, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_sgebrd_strided_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, float *A, const rocblas_int lda, const rocblas_stride strideA, float *D, const rocblas_stride strideD, float *E, const rocblas_stride strideE, float *tauq, const rocblas_stride strideQ, float *taup, const rocblas_stride strideP, const rocblas_int batch_count)

GEBRD_STRIDED_BATCHED computes the bidiagonal form of a batch of general m-by-n matrices.

(This is the blocked version of the algorithm).

The bidiagonal form is given by:

```
B_j = Q_j' * A_j * P_j
```

where B_j is upper bidiagonal if m >= n and lower bidiagonal if m < n, and Q_j and P_j are orthogonal/unitary matrices represented as the product of Householder matrices

```
Q_j = H_j(1) * H_j(2) * ... *  H_j(n)  and P_j = G_j(1) * G_j(2) * ... * G_j(n-1), ␣
↪if m >= n, or
Q_j = H_j(1) * H_j(2) * ... * H_j(m-1) and P_j = G_j(1) * G_j(2) * ... *  G_j(m), ␣
↪if m < n
```

Each Householder matrix H_j(i) and G_j(i), for j = 1,2,...,batch_count, is given by

```
H_j(i) = I - tauq_j[i-1] * v_j(i) * v_j(i)', and
G_j(i) = I - taup_j[i-1] * u_j(i) * u_j(i)'
```

If m >= n, the first i-1 elements of the Householder vector v_j(i) are zero, and v_j(i)[i] = 1; while the first i elements of the Householder vector u_j(i) are zero, and u_j(i)[i+1] = 1. If m < n, the first i elements of the Householder vector v_j(i) are zero, and v_j(i)[i+1] = 1; while the first i-1 elements of the Householder vector u_j(i) are zero, and u_j(i)[i] = 1.

Parameters

- **handle** – **[in]** rocblas_handle.
- **m** – **[in]** rocblas_int. m >= 0.

  The number of rows of all the matrices A_j in the batch.
- **n** – **[in]** rocblas_int. n >= 0.

  The number of columns of all the matrices A_j in the batch.
- **A** – **[inout]** pointer to type. Array on the GPU (the size depends on the value of strideA).

  On entry, the m-by-n matrices A_j to be factored. On exit, the elements on the diagonal and superdiagonal (if m >= n), or subdiagonal (if m < n) contain the bidiagonal form B_j. If m >= n, the elements below the diagonal are the m - i elements of vector v_j(i) for i = 1,2,...,n, and the elements above the superdiagonal are the n - i - 1 elements of vector u_j(i) for i = 1,2,...,n-1. If m < n, the elements below the subdiagonal are the m - i - 1 elements of vector

v_j(i) for i = 1,2,...,m-1, and the elements above the diagonal are the n - i elements of vector u_j(i) for i = 1,2,...,m.

- **lda** – **[in]** rocblas_int. lda >= m.

  Specifies the leading dimension of matrices A_j.

- **strideA** – **[in]** rocblas_stride.

  Stride from the start of one matrix A_j and the next one A_(j+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n.

- **D** – **[out]** pointer to real type. Array on the GPU (the size depends on the value of strideD).

  The diagonal elements of B_j.

- **strideD** – **[in]** rocblas_stride.

  Stride from the start of one vector D_j and the next one D_(j+1). There is no restriction for the value of strideD. Normal use case is strideD >= min(m,n).

- **E** – **[out]** pointer to real type. Array on the GPU (the size depends on the value of strideE).

  The off-diagonal elements of B_j.

- **strideE** – **[in]** rocblas_stride.

  Stride from the start of one vector E_j and the next one E_(j+1). There is no restriction for the value of strideE. Normal use case is strideE >= min(m,n)-1.

- **tauq** – **[out]** pointer to type. Array on the GPU (the size depends on the value of strideQ).

  Contains the vectors tauq_j of scalar factors of the Householder matrices H_j(i).

- **strideQ** – **[in]** rocblas_stride.

  Stride from the start of one vector tauq_j to the next one tauq_(j+1). There is no restriction for the value of strideQ. Normal use is strideQ >= min(m,n).

- **taup** – **[out]** pointer to type. Array on the GPU (the size depends on the value of strideP).

  Contains the vectors taup_j of scalar factors of the Householder matrices G_j(i).

- **strideP** – **[in]** rocblas_stride.

  Stride from the start of one vector taup_j to the next one taup_(j+1). There is no restriction for the value of strideP. Normal use is strideP >= min(m,n).

- **batch_count** – **[in]** rocblas_int. batch_count >= 0.

  Number of matrices in the batch.

2.6.3.4 Symmetric Matrix Diagonalizations

rocsolver_<type>sytd2()

rocblas_status **rocsolver_dsytd2**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, double *A, const rocblas_int lda, double *D, double *E, double *tau)

rocblas_status **rocsolver_ssytd2**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, float *A, const rocblas_int lda, float *D, float *E, float *tau)

SYTD2 computes the tridiagonal form of a real symmetric matrix A.

(This is the unblocked version of the algorithm).

The tridiagonal form is given by:

```
T = Q' * A * Q
```

where T is symmetric tridiagonal and Q is an orthogonal matrix represented as the product of Householder matrices

```
Q = H(1) * H(2) * ... *  H(n-1) if uplo indicates lower, or
Q = H(n-1) * H(n-2) * ... * H(1) if uplo indicates upper.
```

Each Householder matrix H(i) is given by

```
H(i) = I - tau[i] * v(i) * v(i)'
```

where tau[i] is the corresponding Householder scalar. When uplo indicates lower, the first i elements of the Householder vector v(i) are zero, and v(i)[i+1] = 1. If uplo indicates upper, the last n-i elements of the Householder vector v(i) are zero, and v(i)[i] = 1.

Parameters

- **handle** – **[in]** rocblas_handle.
- **uplo** – **[in]** rocblas_fill.

  Specifies whether the upper or lower part of the symmetric matrix A is stored. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.
- **n** – **[in]** rocblas_int. n >= 0.

  The number of rows and columns of the matrix A.
- **A** – **[inout]** pointer to type. Array on the GPU of dimension lda*n.

  On entry, the matrix to be factored. On exit, if upper, then the elements on the diagonal and superdiagonal contain the tridiagonal form T; the elements above the superdiagonal contain the i-1 non-zero elements of vectors v(i) stored as columns. If lower, then the elements on the diagonal and subdiagonal contain the tridiagonal form T; the elements below the subdiagonal contain the n-i-1 non-zero elements of vectors v(i) stored as columns.
- **lda** – **[in]** rocblas_int. lda >= n.

  specifies the leading dimension of A.
- **D** – **[out]** pointer to type. Array on the GPU of dimension n.

  The diagonal elements of T.
- **E** – **[out]** pointer to type. Array on the GPU of dimension n-1.

  The off-diagonal elements of T.
- **tau** – **[out]** pointer to type. Array on the GPU of dimension n-1.

  The scalar factors of the Householder matrices H(i).

rocsolver_<type>sytd2_batched()

rocblas_status **rocsolver_dsytd2_batched**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, double *const A[], const rocblas_int lda, double *D, const rocblas_stride strideD, double *E, const rocblas_stride strideE, double *tau, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_ssytd2_batched**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, float *const A[], const rocblas_int lda, float *D, const rocblas_stride strideD, float *E, const rocblas_stride strideE, float *tau, const rocblas_stride strideP, const rocblas_int batch_count)

SYTD2_BATCHED computes the tridiagonal form of a batch of real symmetric matrices A_j.

(This is the unblocked version of the algorithm).

The tridiagonal form of A_j is given by:

```
T_j = Q_j' * A_j * Q_j, for j = 1,2,...,batch_count
```

where T_j is symmetric tridiagonal and Q_j is an orthogonal matrix represented as the product of Householder matrices

```
Q_j = H_j(1) * H_j(2) * ... *  H_j(n-1) if uplo indicates lower, or
Q_j = H_j(n-1) * H_j(n-2) * ... * H_j(1) if uplo indicates upper.
```

Each Householder matrix H_j(i) is given by

```
H_j(i) = I - tau_j[i] * v_j(i) * v_j(i)'
```

where tau_j[i] is the corresponding Householder scalar. When uplo indicates lower, the first i elements of the Householder vector v_j(i) are zero, and v_j(i)[i+1] = 1. If uplo indicates upper, the last n-i elements of the Householder vector v_j(i) are zero, and v_j(i)[i] = 1.

Parameters

- **handle** – **[in]** rocblas_handle.
- **uplo** – **[in]** rocblas_fill.

  Specifies whether the upper or lower part of the symmetric matrix A_j is stored. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.
- **n** – **[in]** rocblas_int. n >= 0.

  The number of rows and columns of the matrices A_j.
- **A** – **[inout]** array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n.

  On entry, the matrices A_j to be factored. On exit, if upper, then the elements on the diagonal and superdiagonal of A_j contain the tridiagonal form T_j; the elements above the superdiagonal contain the i-1 non-zero elements of vectors v_j(i) stored as columns. If lower, then the elements on the diagonal and subdiagonal contain the tridiagonal form T_j; the elements below the subdiagonal contain the n-i-1 non-zero elements of vectors v_j(i) stored as columns.
- **lda** – **[in]** rocblas_int. lda >= n.

  specifies the leading dimension of A_j.

- **D** – **[out]** pointer to type. Array on the GPU (the size depends on the value of strideD).

  The diagonal elements of T_j.

- **strideD** – **[in]** rocblas_stride.

  Stride from the start of one vector D_j and the next one D_(j+1). There is no restriction for the value of strideD. Normal use case is strideD >= n.

- **E** – **[out]** pointer to type. Array on the GPU (the size depends on the value of strideE).

  The off-diagonal elements of T_j.

- **strideE** – **[in]** rocblas_stride.

  Stride from the start of one vector E_j and the next one E_(j+1). There is no restriction for the value of strideE. Normal use case is strideE >= n-1.

- **tau** – **[out]** pointer to type. Array on the GPU (the size depends on the value of strideP).

  Contains the vectors tau_j of scalar factors of the Householder matrices H_j(i).

- **strideP** – **[in]** rocblas_stride.

  Stride from the start of one vector tau_j to the next one tau_(j+1). There is no restriction for the value of strideP. Normal use is strideP >= n-1.

- **batch_count** – **[in]** rocblas_int. batch_count >= 0.

  Number of matrices in the batch.

rocsolver_<type>sytd2_strided_batched()

rocblas_status **rocsolver_dsytd2_strided_batched**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, double *A, const rocblas_int lda, const rocblas_stride strideA, double *D, const rocblas_stride strideD, double *E, const rocblas_stride strideE, double *tau, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_ssytd2_strided_batched**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, float *A, const rocblas_int lda, const rocblas_stride strideA, float *D, const rocblas_stride strideD, float *E, const rocblas_stride strideE, float *tau, const rocblas_stride strideP, const rocblas_int batch_count)

SYTD2_STRIDED_BATCHED computes the tridiagonal form of a batch of real symmetric matrices A_j.

(This is the unblocked version of the algorithm).

The tridiagonal form of A_j is given by:

```
T_j = Q_j' * A_j * Q_j, for j = 1,2,...,batch_count
```

where T_j is symmetric tridiagonal and Q_j is an orthogonal matrix represented as the product of Householder matrices

```
Q_j = H_j(1) * H_j(2) * ... *  H_j(n-1) if uplo indicates lower, or
Q_j = H_j(n-1) * H_j(n-2) * ... * H_j(1) if uplo indicates upper.
```

Each Householder matrix H_j(i) is given by

```
H_j(i) = I - tau_j[i] * v_j(i) * v_j(i)'
```

where tau_j[i] is the corresponding Householder scalar. When uplo indicates lower, the first i elements of the Householder vector v_j(i) are zero, and v_j(i)[i+1] = 1. If uplo indicates upper, the last n-i elements of the Householder vector v_j(i) are zero, and v_j(i)[i] = 1.

Parameters

- **handle** – **[in]** rocblas_handle.
- **uplo** – **[in]** rocblas_fill.

  Specifies whether the upper or lower part of the symmetric matrix A_j is stored. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.
- **n** – **[in]** rocblas_int. n >= 0.

  The number of rows and columns of the matrices A_j.
- **A** – **[inout]** pointer to type. Array on the GPU (the size depends on the value of strideA).

  On entry, the matrices A_j to be factored. On exit, if upper, then the elements on the diagonal and superdiagonal of A_j contain the tridiagonal form T_j; the elements above the superdiagonal contain the i-1 non-zero elements of vectors v_j(i) stored as columns. If lower, then the elements on the diagonal and subdiagonal contain the tridiagonal form T_j; the elements below the subdiagonal contain the n-i-1 non-zero elements of vectors v_j(i) stored as columns.
- **lda** – **[in]** rocblas_int. lda >= n.

  specifies the leading dimension of A_j.
- **strideA** – **[in]** rocblas_stride.

  Stride from the start of one matrix A_j and the next one A_(j+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n.
- **D** – **[out]** pointer to type. Array on the GPU (the size depends on the value of strideD).

  The diagonal elements of T_j.
- **strideD** – **[in]** rocblas_stride.

  Stride from the start of one vector D_j and the next one D_(j+1). There is no restriction for the value of strideD. Normal use case is strideD >= n.
- **E** – **[out]** pointer to type. Array on the GPU (the size depends on the value of strideE).

  The off-diagonal elements of T_j.
- **strideE** – **[in]** rocblas_stride.

  Stride from the start of one vector E_j and the next one E_(j+1). There is no restriction for the value of strideE. Normal use case is strideE >= n-1.
- **tau** – **[out]** pointer to type. Array on the GPU (the size depends on the value of strideP).

  Contains the vectors tau_j of scalar factors of the Householder matrices H_j(i).
- **strideP** – **[in]** rocblas_stride.

  Stride from the start of one vector tau_j to the next one tau_(j+1). There is no restriction for the value of strideP. Normal use is strideP >= n-1.

- **batch_count** – **[in]** rocblas_int. batch_count >= 0.

  Number of matrices in the batch.

rocsolver_<type>hetd2()

rocblas_status **rocsolver_zhetd2**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, rocblas_double_complex *A, const rocblas_int lda, double *D, double *E, rocblas_double_complex *tau)

rocblas_status **rocsolver_chetd2**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, rocblas_float_complex *A, const rocblas_int lda, float *D, float *E, rocblas_float_complex *tau)

HETD2 computes the tridiagonal form of a complex hermitian matrix A.

(This is the unblocked version of the algorithm).

The tridiagonal form is given by:

```
T = Q' * A * Q
```

where T is hermitian tridiagonal and Q is an unitary matrix represented as the product of Householder matrices

```
Q = H(1) * H(2) * ... *  H(n-1) if uplo indicates lower, or
Q = H(n-1) * H(n-2) * ... * H(1) if uplo indicates upper.
```

Each Householder matrix H(i) is given by

```
H(i) = I - tau[i] * v(i) * v(i)'
```

where tau[i] is the corresponding Householder scalar. When uplo indicates lower, the first i elements of the Householder vector v(i) are zero, and v(i)[i+1] = 1. If uplo indicates upper, the last n-i elements of the Householder vector v(i) are zero, and v(i)[i] = 1.

Parameters

- **handle** – **[in]** rocblas_handle.
- **uplo** – **[in]** rocblas_fill.

  Specifies whether the upper or lower part of the hermitian matrix A is stored. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.

- **n** – **[in]** rocblas_int. n >= 0.

  The number of rows and columns of the matrix A.

- **A** – **[inout]** pointer to type. Array on the GPU of dimension lda*n.

  On entry, the matrix to be factored. On exit, if upper, then the elements on the diagonal and superdiagonal contain the tridiagonal form T; the elements above the superdiagonal contain the i-1 non-zero elements of vectors v(i) stored as columns. If lower, then the elements on the diagonal and subdiagonal contain the tridiagonal form T; the elements below the subdiagonal contain the n-i-1 non-zero elements of vectors v(i) stored as columns.

- **lda** – **[in]** rocblas_int. lda >= n.

  specifies the leading dimension of A.

- **D** – **[out]** pointer to real type. Array on the GPU of dimension n.

  The diagonal elements of T.

- **E** – **[out]** pointer to real type. Array on the GPU of dimension n-1.

  The off-diagonal elements of T.

- **tau** – **[out]** pointer to type. Array on the GPU of dimension n-1.

  The scalar factors of the Householder matrices H(i).

rocsolver_<type>hetd2_batched()

rocblas_status **rocsolver_zhetd2_batched**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, rocblas_double_complex *const A[], const rocblas_int lda, double *D, const rocblas_stride strideD, double *E, const rocblas_stride strideE, rocblas_double_complex *tau, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_chetd2_batched**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, rocblas_float_complex *const A[], const rocblas_int lda, float *D, const rocblas_stride strideD, float *E, const rocblas_stride strideE, rocblas_float_complex *tau, const rocblas_stride strideP, const rocblas_int batch_count)

HETD2_BATCHED computes the tridiagonal form of a batch of complex hermitian matrices A_j.

(This is the unblocked version of the algorithm).

The tridiagonal form of A_j is given by:

```
T_j = Q_j' * A_j * Q_j, for j = 1,2,...,batch_count
```

where T_j is hermitian tridiagonal and Q_j is a unitary matrix represented as the product of Householder matrices

```
Q_j = H_j(1) * H_j(2) * ... *  H_j(n-1) if uplo indicates lower, or
Q_j = H_j(n-1) * H_j(n-2) * ... * H_j(1) if uplo indicates upper.
```

Each Householder matrix H_j(i) is given by

```
H_j(i) = I - tau_j[i] * v_j(i) * v_j(i)'
```

where tau_j[i] is the corresponding Householder scalar. When uplo indicates lower, the first i elements of the Householder vector v_j(i) are zero, and v_j(i)[i+1] = 1. If uplo indicates upper, the last n-i elements of the Householder vector v_j(i) are zero, and v_j(i)[i] = 1.

Parameters

- **handle** – **[in]** rocblas_handle.
- **uplo** – **[in]** rocblas_fill.

  Specifies whether the upper or lower part of the hermitian matrix A_j is stored. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.

- **n** – **[in]** rocblas_int. n >= 0.

  The number of rows and columns of the matrices A_j.

- **A** – **[inout]** array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n.

  On entry, the matrices A_j to be factored. On exit, if upper, then the elements on the diagonal and superdiagonal of A_j contain the tridiagonal form T_j; the elements above the superdiagonal contain the i-1 non-zero elements of vectors v_j(i) stored as columns. If lower, then the elements on the diagonal and subdiagonal contain the tridiagonal form T_j; the elements below the subdiagonal contain the n-i-1 non-zero elements of vectors v_j(i) stored as columns.

- **lda** – **[in]** rocblas_int. lda >= n.

  specifies the leading dimension of A_j.

- **D** – **[out]** pointer to real type. Array on the GPU (the size depends on the value of strideD).

  The diagonal elements of T_j.

- **strideD** – **[in]** rocblas_stride.

  Stride from the start of one vector D_j and the next one D_(j+1). There is no restriction for the value of strideD. Normal use case is strideD >= n.

- **E** – **[out]** pointer to real type. Array on the GPU (the size depends on the value of strideE).

  The off-diagonal elements of T_j.

- **strideE** – **[in]** rocblas_stride.

  Stride from the start of one vector E_j and the next one E_(j+1). There is no restriction for the value of strideE. Normal use case is strideE >= n-1.

- **tau** – **[out]** pointer to type. Array on the GPU (the size depends on the value of strideP).

  Contains the vectors tau_j of scalar factors of the Householder matrices H_j(i).

- **strideP** – **[in]** rocblas_stride.

  Stride from the start of one vector tau_j to the next one tau_(j+1). There is no restriction for the value of strideP. Normal use is strideP >= n-1.

- **batch_count** – **[in]** rocblas_int. batch_count >= 0.

  Number of matrices in the batch.

rocsolver_<type>hetd2_strided_batched()

rocblas_status **rocsolver_zhetd2_strided_batched**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, rocblas_double_complex *A, const rocblas_int lda, const rocblas_stride strideA, double *D, const rocblas_stride strideD, double *E, const rocblas_stride strideE, rocblas_double_complex *tau, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_chetd2_strided_batched**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, rocblas_float_complex *A, const rocblas_int lda, const rocblas_stride strideA, float *D, const rocblas_stride strideD, float *E, const rocblas_stride strideE, rocblas_float_complex *tau, const rocblas_stride strideP, const rocblas_int batch_count)

HETD2_STRIDED_BATCHED computes the tridiagonal form of a batch of complex hermitian matrices A_j.

(This is the unblocked version of the algorithm).

The tridiagonal form of A_j is given by:

```
T_j = Q_j' * A_j * Q_j, for j = 1,2,...,batch_count
```

where T_j is hermitian tridiagonal and Q_j is a unitary matrix represented as the product of Householder matrices

```
Q_j = H_j(1) * H_j(2) * ... *  H_j(n-1) if uplo indicates lower, or
Q_j = H_j(n-1) * H_j(n-2) * ... * H_j(1) if uplo indicates upper.
```

Each Householder matrix H_j(i) is given by

```
H_j(i) = I - tau_j[i] * v_j(i) * v_j(i)'
```

where tau_j[i] is the corresponding Householder scalar. When uplo indicates lower, the first i elements of the Householder vector v_j(i) are zero, and v_j(i)[i+1] = 1. If uplo indicates upper, the last n-i elements of the Householder vector v_j(i) are zero, and v_j(i)[i] = 1.

Parameters

- **handle** – **[in]** rocblas_handle.
- **uplo** – **[in]** rocblas_fill.

  Specifies whether the upper or lower part of the hermitian matrix A_j is stored. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.

- **n** – **[in]** rocblas_int. n >= 0.

  The number of rows and columns of the matrices A_j.

- **A** – **[inout]** pointer to type. Array on the GPU (the size depends on the value of strideA).

  On entry, the matrices A_j to be factored. On exit, if upper, then the elements on the diagonal and superdiagonal of A_j contain the tridiagonal form T_j; the elements above the superdiagonal contain the i-1 non-zero elements of vectors v_j(i) stored as columns. If lower, then the elements on the diagonal and subdiagonal contain the tridiagonal form T_j; the elements below the subdiagonal contain the n-i-1 non-zero elements of vectors v_j(i) stored as columns.

- **lda** – **[in]** rocblas_int. lda >= n.

  specifies the leading dimension of A_j.

- **strideA** – **[in]** rocblas_stride.

  Stride from the start of one matrix A_j and the next one A_(j+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n.

- **D** – **[out]** pointer to real type. Array on the GPU (the size depends on the value of strideD).

  The diagonal elements of T_j.

- **strideD** – **[in]** rocblas_stride.

  Stride from the start of one vector D_j and the next one D_(j+1). There is no restriction for the value of strideD. Normal use case is strideD >= n.

- **E** – **[out]** pointer to real type. Array on the GPU (the size depends on the value of strideE).

  The off-diagonal elements of T_j.

- **strideE** – **[in]** rocblas_stride.

  Stride from the start of one vector E_j and the next one E_(j+1). There is no restriction for the value of strideE. Normal use case is strideE >= n-1.

- **tau** – **[out]** pointer to type. Array on the GPU (the size depends on the value of strideP).

  Contains the vectors tau_j of scalar factors of the Householder matrices H_j(i).

- **strideP** – **[in]** rocblas_stride.

  Stride from the start of one vector tau_j to the next one tau_(j+1). There is no restriction for the value of strideP. Normal use is strideP >= n-1.

- **batch_count** – **[in]** rocblas_int. batch_count >= 0.

  Number of matrices in the batch.

rocsolver_<type>sytrd()

rocblas_status **rocsolver_dsytrd**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, double *A, const rocblas_int lda, double *D, double *E, double *tau)

rocblas_status **rocsolver_ssytrd**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, float *A, const rocblas_int lda, float *D, float *E, float *tau)

SYTRD computes the tridiagonal form of a real symmetric matrix A.

(This is the blocked version of the algorithm).

The tridiagonal form is given by:

```
T = Q' * A * Q
```

where T is symmetric tridiagonal and Q is an orthogonal matrix represented as the product of Householder matrices

```
Q = H(1) * H(2) * ... *  H(n-1) if uplo indicates lower, or
Q = H(n-1) * H(n-2) * ... * H(1) if uplo indicates upper.
```

Each Householder matrix H(i) is given by

```
H(i) = I - tau[i] * v(i) * v(i)'
```

where tau[i] is the corresponding Householder scalar. When uplo indicates lower, the first i elements of the Householder vector v(i) are zero, and v(i)[i+1] = 1. If uplo indicates upper, the last n-i elements of the Householder vector v(i) are zero, and v(i)[i] = 1.

Parameters

- **handle** – **[in]** rocblas_handle.
- **uplo** – **[in]** rocblas_fill.

  Specifies whether the upper or lower part of the symmetric matrix A is stored. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.

- **n** – **[in]** rocblas_int. n >= 0.

  The number of rows and columns of the matrix A.

- **A** – **[inout]** pointer to type. Array on the GPU of dimension lda*n.

  On entry, the matrix to be factored. On exit, if upper, then the elements on the diagonal and superdiagonal contain the tridiagonal form T; the elements above the superdiagonal contain the i-1 non-zero elements of vectors v(i) stored as columns. If lower, then the elements on the diagonal and subdiagonal contain the tridiagonal form T; the elements below the subdiagonal contain the n-i-1 non-zero elements of vectors v(i) stored as columns.

- **lda** – **[in]** rocblas_int. lda >= n.

  specifies the leading dimension of A.

- **D** – **[out]** pointer to type. Array on the GPU of dimension n.

  The diagonal elements of T.

- **E** – **[out]** pointer to type. Array on the GPU of dimension n-1.

  The off-diagonal elements of T.

- **tau** – **[out]** pointer to type. Array on the GPU of dimension n-1.

  The scalar factors of the Householder matrices H(i).

rocsolver_<type>sytrd_batched()

rocblas_status **rocsolver_dsytrd_batched**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, double *const A[], const rocblas_int lda, double *D, const rocblas_stride strideD, double *E, const rocblas_stride strideE, double *tau, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_ssytrd_batched**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, float *const A[], const rocblas_int lda, float *D, const rocblas_stride strideD, float *E, const rocblas_stride strideE, float *tau, const rocblas_stride strideP, const rocblas_int batch_count)

SYTRD_BATCHED computes the tridiagonal form of a batch of real symmetric matrices A_j.

(This is the blocked version of the algorithm).

The tridiagonal form of A_j is given by:

```
T_j = Q_j' * A_j * Q_j, for j = 1,2,...,batch_count
```

where T_j is symmetric tridiagonal and Q_j is an orthogonal matrix represented as the product of Householder matrices

```
Q_j = H_j(1) * H_j(2) * ... *  H_j(n-1) if uplo indicates lower, or
Q_j = H_j(n-1) * H_j(n-2) * ... * H_j(1) if uplo indicates upper.
```

Each Householder matrix H_j(i) is given by

```
H_j(i) = I - tau_j[i] * v_j(i) * v_j(i)'
```

where tau_j[i] is the corresponding Householder scalar. When uplo indicates lower, the first i elements of the Householder vector v_j(i) are zero, and v_j(i)[i+1] = 1. If uplo indicates upper, the last n-i elements of the Householder vector v_j(i) are zero, and v_j(i)[i] = 1.

Parameters

- **handle** – **[in]** rocblas_handle.
- **uplo** – **[in]** rocblas_fill.

  Specifies whether the upper or lower part of the symmetric matrix A_j is stored. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.

- **n** – **[in]** rocblas_int. n >= 0.

  The number of rows and columns of the matrices A_j.

- **A** – **[inout]** array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n.

  On entry, the matrices A_j to be factored. On exit, if upper, then the elements on the diagonal and superdiagonal of A_j contain the tridiagonal form T_j; the elements above the superdiagonal contain the i-1 non-zero elements of vectors v_j(i) stored as columns. If lower, then the elements on the diagonal and subdiagonal contain the tridiagonal form T_j; the elements below the subdiagonal contain the n-i-1 non-zero elements of vectors v_j(i) stored as columns.

- **lda** – **[in]** rocblas_int. lda >= n.

  specifies the leading dimension of A_j.

- **D** – **[out]** pointer to type. Array on the GPU (the size depends on the value of strideD).

  The diagonal elements of T_j.

- **strideD** – **[in]** rocblas_stride.

  Stride from the start of one vector D_j and the next one D_(j+1). There is no restriction for the value of strideD. Normal use case is strideD >= n.

- **E** – **[out]** pointer to type. Array on the GPU (the size depends on the value of strideE).

  The off-diagonal elements of T_j.

- **strideE** – **[in]** rocblas_stride.

  Stride from the start of one vector E_j and the next one E_(j+1). There is no restriction for the value of strideE. Normal use case is strideE >= n-1.

- **tau** – **[out]** pointer to type. Array on the GPU (the size depends on the value of strideP).

  Contains the vectors tau_j of scalar factors of the Householder matrices H_j(i).

- **strideP** – **[in]** rocblas_stride.

  Stride from the start of one vector tau_j to the next one tau_(j+1). There is no restriction for the value of strideP. Normal use is strideP >= n-1.

- **batch_count** – **[in]** rocblas_int. batch_count >= 0.

  Number of matrices in the batch.

rocsolver_<type>sytrd_strided_batched()

rocblas_status **rocsolver_dsytrd_strided_batched**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, double *A, const rocblas_int lda, const rocblas_stride strideA, double *D, const rocblas_stride strideD, double *E, const rocblas_stride strideE, double *tau, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_ssytrd_strided_batched**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, float *A, const rocblas_int lda, const rocblas_stride strideA, float *D, const rocblas_stride strideD, float *E, const rocblas_stride strideE, float *tau, const rocblas_stride strideP, const rocblas_int batch_count)

SYTRD_STRIDED_BATCHED computes the tridiagonal form of a batch of real symmetric matrices A_j.

(This is the blocked version of the algorithm).

The tridiagonal form of A_j is given by:

```
T_j = Q_j' * A_j * Q_j, for j = 1,2,...,batch_count
```

where T_j is symmetric tridiagonal and Q_j is an orthogonal matrix represented as the product of Householder matrices

```
Q_j = H_j(1) * H_j(2) * ... *  H_j(n-1) if uplo indicates lower, or
Q_j = H_j(n-1) * H_j(n-2) * ... * H_j(1) if uplo indicates upper.
```

Each Householder matrix H_j(i) is given by

```
H_j(i) = I - tau_j[i] * v_j(i) * v_j(i)'
```

where tau_j[i] is the corresponding Householder scalar. When uplo indicates lower, the first i elements of the Householder vector v_j(i) are zero, and v_j(i)[i+1] = 1. If uplo indicates upper, the last n-i elements of the Householder vector v_j(i) are zero, and v_j(i)[i] = 1.

Parameters

- **handle** – **[in]** rocblas_handle.
- **uplo** – **[in]** rocblas_fill.

  Specifies whether the upper or lower part of the symmetric matrix A_j is stored. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.

- **n** – **[in]** rocblas_int. n >= 0.

  The number of rows and columns of the matrices A_j.

- **A** – **[inout]** pointer to type. Array on the GPU (the size depends on the value of strideA).

  On entry, the matrices A_j to be factored. On exit, if upper, then the elements on the diagonal and superdiagonal of A_j contain the tridiagonal form T_j; the elements above the superdiagonal contain the i-1 non-zero elements of vectors v_j(i) stored as columns. If lower, then the elements on the diagonal and subdiagonal contain the tridiagonal form T_j; the elements below the subdiagonal contain the n-i-1 non-zero elements of vectors v_j(i) stored as columns.

- **lda** – **[in]** rocblas_int. lda >= n.

  specifies the leading dimension of A_j.

- **strideA** – **[in]** rocblas_stride.

  Stride from the start of one matrix A_j and the next one A_(j+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n.

- **D** – **[out]** pointer to type. Array on the GPU (the size depends on the value of strideD).

  The diagonal elements of T_j.

- **strideD** – **[in]** rocblas_stride.

  Stride from the start of one vector D_j and the next one D_(j+1). There is no restriction for the value of strideD. Normal use case is strideD >= n.

- **E** – **[out]** pointer to type. Array on the GPU (the size depends on the value of strideE).

  The off-diagonal elements of T_j.

- **strideE** – **[in]** rocblas_stride.

  Stride from the start of one vector E_j and the next one E_(j+1). There is no restriction for the value of strideE. Normal use case is strideE >= n-1.

- **tau** – **[out]** pointer to type. Array on the GPU (the size depends on the value of strideP).

  Contains the vectors tau_j of scalar factors of the Householder matrices H_j(i).

- **strideP** – **[in]** rocblas_stride.

  Stride from the start of one vector tau_j to the next one tau_(j+1). There is no restriction for the value of strideP. Normal use is strideP >= n-1.

- **batch_count** – **[in]** rocblas_int. batch_count >= 0.

  Number of matrices in the batch.

rocsolver_<type>hetrd()

rocblas_status **rocsolver_zhetrd**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, rocblas_double_complex *A, const rocblas_int lda, double *D, double *E, rocblas_double_complex *tau)

rocblas_status **rocsolver_chetrd**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, rocblas_float_complex *A, const rocblas_int lda, float *D, float *E, rocblas_float_complex *tau)

HETRD computes the tridiagonal form of a complex hermitian matrix A.

(This is the blocked version of the algorithm).

The tridiagonal form is given by:

```
T = Q' * A * Q
```

where T is hermitian tridiagonal and Q is an unitary matrix represented as the product of Householder matrices

```
Q = H(1) * H(2) * ... *  H(n-1) if uplo indicates lower, or
Q = H(n-1) * H(n-2) * ... * H(1) if uplo indicates upper.
```

Each Householder matrix H(i) is given by

```
H(i) = I - tau[i] * v(i) * v(i)'
```

where tau[i] is the corresponding Householder scalar. When uplo indicates lower, the first i elements of the Householder vector v(i) are zero, and v(i)[i+1] = 1. If uplo indicates upper, the last n-i elements of the Householder vector v(i) are zero, and v(i)[i] = 1.

Parameters

- **handle** – **[in]** rocblas_handle.
- **uplo** – **[in]** rocblas_fill.

  Specifies whether the upper or lower part of the hermitian matrix A is stored. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.
- **n** – **[in]** rocblas_int. n >= 0.

  The number of rows and columns of the matrix A.
- **A** – **[inout]** pointer to type. Array on the GPU of dimension lda*n.

  On entry, the matrix to be factored. On exit, if upper, then the elements on the diagonal and superdiagonal contain the tridiagonal form T; the elements above the superdiagonal contain the i-1 non-zero elements of vectors v(i) stored as columns. If lower, then the elements on the diagonal and subdiagonal contain the tridiagonal form T; the elements below the subdiagonal contain the n-i-1 non-zero elements of vectors v(i) stored as columns.
- **lda** – **[in]** rocblas_int. lda >= n.

  specifies the leading dimension of A.
- **D** – **[out]** pointer to real type. Array on the GPU of dimension n.

  The diagonal elements of T.
- **E** – **[out]** pointer to real type. Array on the GPU of dimension n-1.

  The off-diagonal elements of T.
- **tau** – **[out]** pointer to type. Array on the GPU of dimension n-1.

  The scalar factors of the Householder matrices H(i).

rocsolver_<type>hetrd_batched()

rocblas_status **rocsolver_zhetrd_batched**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, rocblas_double_complex *const A[], const rocblas_int lda, double *D, const rocblas_stride strideD, double *E, const rocblas_stride strideE, rocblas_double_complex *tau, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_chetrd_batched**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, rocblas_float_complex *const A[], const rocblas_int lda, float *D, const rocblas_stride strideD, float *E, const rocblas_stride strideE, rocblas_float_complex *tau, const rocblas_stride strideP, const rocblas_int batch_count)

HETRD_BATCHED computes the tridiagonal form of a batch of complex hermitian matrices A_j.

(This is the blocked version of the algorithm).

The tridiagonal form of A_j is given by:

```
T_j = Q_j' * A_j * Q_j, for j = 1,2,...,batch_count
```

where T_j is hermitian tridiagonal and Q_j is a unitary matrix represented as the product of Householder matrices

```
Q_j = H_j(1) * H_j(2) * ... *  H_j(n-1) if uplo indicates lower, or
Q_j = H_j(n-1) * H_j(n-2) * ... * H_j(1) if uplo indicates upper.
```

Each Householder matrix H_j(i) is given by

```
H_j(i) = I - tau_j[i] * v_j(i) * v_j(i)'
```

where tau_j[i] is the corresponding Householder scalar. When uplo indicates lower, the first i elements of the Householder vector v_j(i) are zero, and v_j(i)[i+1] = 1. If uplo indicates upper, the last n-i elements of the Householder vector v_j(i) are zero, and v_j(i)[i] = 1.

Parameters

- **handle** – **[in]** rocblas_handle.
- **uplo** – **[in]** rocblas_fill.

  Specifies whether the upper or lower part of the hermitian matrix A_j is stored. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.

- **n** – **[in]** rocblas_int. n >= 0.

  The number of rows and columns of the matrices A_j.

- **A** – **[inout]** array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n.

  On entry, the matrices A_j to be factored. On exit, if upper, then the elements on the diagonal and superdiagonal of A_j contain the tridiagonal form T_j; the elements above the superdiagonal contain the i-1 non-zero elements of vectors v_j(i) stored as columns. If lower, then the elements on the diagonal and subdiagonal contain the tridiagonal form T_j; the elements below the subdiagonal contain the n-i-1 non-zero elements of vectors v_j(i) stored as columns.

- **lda** – **[in]** rocblas_int. lda >= n.

  specifies the leading dimension of A_j.

- **D** – **[out]** pointer to real type. Array on the GPU (the size depends on the value of strideD).

  The diagonal elements of T_j.

- **strideD** – **[in]** rocblas_stride.

  Stride from the start of one vector D_j and the next one D_(j+1). There is no restriction for the value of strideD. Normal use case is strideD >= n.

- **E** – **[out]** pointer to real type. Array on the GPU (the size depends on the value of strideE).

  The off-diagonal elements of T_j.

- **strideE** – **[in]** rocblas_stride.

  Stride from the start of one vector E_j and the next one E_(j+1). There is no restriction for the value of strideE. Normal use case is strideE >= n-1.

- **tau** – **[out]** pointer to type. Array on the GPU (the size depends on the value of strideP).

  Contains the vectors tau_j of scalar factors of the Householder matrices H_j(i).

- **strideP** – **[in]** rocblas_stride.

  Stride from the start of one vector tau_j to the next one tau_(j+1). There is no restriction for the value of strideP. Normal use is strideP >= n-1.

- **batch_count** – **[in]** rocblas_int. batch_count >= 0.

  Number of matrices in the batch.

rocsolver_<type>hetrd_strided_batched()

rocblas_status **rocsolver_zhetrd_strided_batched**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, rocblas_double_complex *A, const rocblas_int lda, const rocblas_stride strideA, double *D, const rocblas_stride strideD, double *E, const rocblas_stride strideE, rocblas_double_complex *tau, const rocblas_stride strideP, const rocblas_int batch_count)

rocblas_status **rocsolver_chetrd_strided_batched**(rocblas_handle handle, const rocblas_fill uplo, const rocblas_int n, rocblas_float_complex *A, const rocblas_int lda, const rocblas_stride strideA, float *D, const rocblas_stride strideD, float *E, const rocblas_stride strideE, rocblas_float_complex *tau, const rocblas_stride strideP, const rocblas_int batch_count)

HETRD_STRIDED_BATCHED computes the tridiagonal form of a batch of complex hermitian matrices A_j.

(This is the blocked version of the algorithm).

The tridiagonal form of A_j is given by:

```
T_j = Q_j' * A_j * Q_j, for j = 1,2,...,batch_count
```

where T_j is hermitian tridiagonal and Q_j is a unitary matrix represented as the product of Householder matrices

```
Q_j = H_j(1) * H_j(2) * ... *  H_j(n-1) if uplo indicates lower, or
Q_j = H_j(n-1) * H_j(n-2) * ... * H_j(1) if uplo indicates upper.
```

Each Householder matrix H_j(i) is given by

```
H_j(i) = I - tau_j[i] * v_j(i) * v_j(i)'
```

where tau_j[i] is the corresponding Householder scalar. When uplo indicates lower, the first i elements of the Householder vector v_j(i) are zero, and v_j(i)[i+1] = 1. If uplo indicates upper, the last n-i elements of the Householder vector v_j(i) are zero, and v_j(i)[i] = 1.

Parameters

- **handle** – **[in]** rocblas_handle.
- **uplo** – **[in]** rocblas_fill.

  Specifies whether the upper or lower part of the hermitian matrix A_j is stored. If uplo indicates lower (or upper), then the upper (or lower) part of A is not used.

- **n** – **[in]** rocblas_int. n >= 0.

  The number of rows and columns of the matrices A_j.

- **A** – **[inout]** pointer to type. Array on the GPU (the size depends on the value of strideA).

  On entry, the matrices A_j to be factored. On exit, if upper, then the elements on the diagonal and superdiagonal of A_j contain the tridiagonal form T_j; the elements above the superdiagonal contain the i-1 non-zero elements of vectors v_j(i) stored as columns. If lower, then the elements on the diagonal and subdiagonal contain the tridiagonal form T_j; the elements below the subdiagonal contain the n-i-1 non-zero elements of vectors v_j(i) stored as columns.

- **lda** – **[in]** rocblas_int. lda >= n.

  specifies the leading dimension of A_j.

- **strideA** – **[in]** rocblas_stride.

  Stride from the start of one matrix A_j and the next one A_(j+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n.

- **D** – **[out]** pointer to real type. Array on the GPU (the size depends on the value of strideD).

  The diagonal elements of T_j.

- **strideD** – **[in]** rocblas_stride.

  Stride from the start of one vector D_j and the next one D_(j+1). There is no restriction for the value of strideD. Normal use case is strideD >= n.

- **E** – **[out]** pointer to real type. Array on the GPU (the size depends on the value of strideE).

  The off-diagonal elements of T_j.

- **strideE** – **[in]** rocblas_stride.

  Stride from the start of one vector E_j and the next one E_(j+1). There is no restriction for the value of strideE. Normal use case is strideE >= n-1.

- **tau** – **[out]** pointer to type. Array on the GPU (the size depends on the value of strideP).

  Contains the vectors tau_j of scalar factors of the Householder matrices H_j(i).

- **strideP** – **[in]** rocblas_stride.

  Stride from the start of one vector tau_j to the next one tau_(j+1). There is no restriction for the value of strideP. Normal use is strideP >= n-1.

- **batch_count** – **[in]** rocblas_int. batch_count >= 0.

  Number of matrices in the batch.

2.6.3.5 General Matrix Inversion

rocsolver_<type>getri()

rocblas_status **rocsolver_zgetri**(rocblas_handle handle, const rocblas_int n, rocblas_double_complex *A, const rocblas_int lda, rocblas_int *ipiv, rocblas_int *info)

rocblas_status **rocsolver_cgetri**(rocblas_handle handle, const rocblas_int n, rocblas_float_complex *A, const rocblas_int lda, rocblas_int *ipiv, rocblas_int *info)

rocblas_status **rocsolver_dgetri**(rocblas_handle handle, const rocblas_int n, double *A, const rocblas_int lda, rocblas_int *ipiv, rocblas_int *info)

rocblas_status **rocsolver_sgetri**(rocblas_handle handle, const rocblas_int n, float *A, const rocblas_int lda, rocblas_int *ipiv, rocblas_int *info)

GETRI inverts a general n-by-n matrix A using the LU factorization computed by GETRF.

The inverse is computed by solving the linear system

```
inv(A) * L = inv(U)
```

where L is the lower triangular factor of A with unit diagonal elements, and U is the upper triangular factor.

Parameters

- **handle** – **[in]** rocblas_handle.
- **n** – **[in]** rocblas_int. n >= 0.

  The number of rows and columns of the matrix A.
- **A** – **[inout]** pointer to type. Array on the GPU of dimension lda*n.

  On entry, the factors L and U of the factorization A = P*L*U returned by GETRF. On exit, the inverse of A if info = 0; otherwise undefined.
- **lda** – **[in]** rocblas_int. lda >= n.

  Specifies the leading dimension of A.
- **ipiv** – **[in]** pointer to rocblas_int. Array on the GPU of dimension n.

  The pivot indices returned by GETRF.
- **info** – **[out]** pointer to a rocblas_int on the GPU.

  If info = 0, successful exit. If info = i > 0, U is singular. U(i,i) is the first zero pivot.

rocsolver_<type>getri_batched()

rocblas_status **rocsolver_zgetri_batched**(rocblas_handle handle, const rocblas_int n, rocblas_double_complex *const A[], const rocblas_int lda, rocblas_int *ipiv, const rocblas_stride strideP, rocblas_int *info, const rocblas_int batch_count)

rocblas_status **rocsolver_cgetri_batched**(rocblas_handle handle, const rocblas_int n, rocblas_float_complex *const A[], const rocblas_int lda, rocblas_int *ipiv, const rocblas_stride strideP, rocblas_int *info, const rocblas_int batch_count)

rocblas_status **rocsolver_dgetri_batched**(rocblas_handle handle, const rocblas_int n, double *const A[], const rocblas_int lda, rocblas_int *ipiv, const rocblas_stride strideP, rocblas_int *info, const rocblas_int batch_count)

rocblas_status **rocsolver_sgetri_batched**(rocblas_handle handle, const rocblas_int n, float *const A[], const rocblas_int lda, rocblas_int *ipiv, const rocblas_stride strideP, rocblas_int *info, const rocblas_int batch_count)

GETRI_BATCHED inverts a batch of general n-by-n matrices using the LU factorization computed by GETRF_BATCHED.

The inverse is computed by solving the linear system

```
inv(A_j) * L_j = inv(U_j)
```

where L_j is the lower triangular factor of A_j with unit diagonal elements, and U_j is the upper triangular factor.

Parameters

- **handle** – **[in]** rocblas_handle.
- **n** – **[in]** rocblas_int. n >= 0.

  The number of rows and columns of all matrices A_j in the batch.
- **A** – **[inout]** array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n.

  On entry, the factors L_j and U_j of the factorization A = P_j*L_j*U_j returned by GETRF_BATCHED. On exit, the inverses of A_j if info_j = 0; otherwise undefined.
- **lda** – **[in]** rocblas_int. lda >= n.

  Specifies the leading dimension of matrices A_j.
- **ipiv** – **[in]** pointer to rocblas_int. Array on the GPU (the size depends on the value of strideP).

  The pivot indices returned by GETRF_BATCHED.
- **strideP** – **[in]** rocblas_stride.

  Stride from the start of one vector ipiv_j to the next one ipiv_(i+j). There is no restriction for the value of strideP. Normal use case is strideP >= n.
- **info** – **[out]** pointer to rocblas_int. Array of batch_count integers on the GPU.

  If info_j = 0, successful exit for inversion of A_j. If info_j = i > 0, U_j is singular. U_j(i,i) is the first zero pivot.
- **batch_count** – **[in]** rocblas_int. batch_count >= 0.

  Number of matrices in the batch.

rocsolver_<type>getri_strided_batched()

rocblas_status **rocsolver_zgetri_strided_batched**(rocblas_handle handle, const rocblas_int n, rocblas_double_complex *A, const rocblas_int lda, const rocblas_stride strideA, rocblas_int *ipiv, const rocblas_stride strideP, rocblas_int *info, const rocblas_int batch_count)

rocblas_status **rocsolver_cgetri_strided_batched**(rocblas_handle handle, const rocblas_int n, rocblas_float_complex *A, const rocblas_int lda, const rocblas_stride strideA, rocblas_int *ipiv, const rocblas_stride strideP, rocblas_int *info, const rocblas_int batch_count)

rocblas_status **rocsolver_dgetri_strided_batched**(rocblas_handle handle, const rocblas_int n, double *A, const rocblas_int lda, const rocblas_stride strideA, rocblas_int *ipiv, const rocblas_stride strideP, rocblas_int *info, const rocblas_int batch_count)

rocblas_status **rocsolver_sgetri_strided_batched**(rocblas_handle handle, const rocblas_int n, float *A, const rocblas_int lda, const rocblas_stride strideA, rocblas_int *ipiv, const rocblas_stride strideP, rocblas_int *info, const rocblas_int batch_count)

GETRI_STRIDED_BATCHED inverts a batch of general n-by-n matrices using the LU factorization computed by GETRF_STRIDED_BATCHED.

The inverse is computed by solving the linear system

```
inv(A_j) * L_j = inv(U_j)
```

where L_j is the lower triangular factor of A_j with unit diagonal elements, and U_j is the upper triangular factor.

Parameters

- **handle** – **[in]** rocblas_handle.
- **n** – **[in]** rocblas_int. n >= 0.

  The number of rows and columns of all matrices A_i in the batch.
- **A** – **[inout]** pointer to type. Array on the GPU (the size depends on the value of strideA).

  On entry, the factors L_j and U_j of the factorization A_j = P_j*L_j*U_j returned by GETRF_STRIDED_BATCHED. On exit, the inverses of A_j if info_j = 0; otherwise undefined.
- **lda** – **[in]** rocblas_int. lda >= n.

  Specifies the leading dimension of matrices A_j.
- **strideA** – **[in]** rocblas_stride.

  Stride from the start of one matrix A_j and the next one A_(j+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n
- **ipiv** – **[in]** pointer to rocblas_int. Array on the GPU (the size depends on the value of strideP).

  The pivot indices returned by GETRF_STRIDED_BATCHED.
- **strideP** – **[in]** rocblas_stride.

  Stride from the start of one vector ipiv_j to the next one ipiv_(j+1). There is no restriction for the value of strideP. Normal use case is strideP >= n.
- **info** – **[out]** pointer to rocblas_int. Array of batch_count integers on the GPU.

  If info_j = 0, successful exit for inversion of A_j. If info_j = i > 0, U_j is singular. U_j(i,i) is the first zero pivot.

- **batch_count** – **[in]** rocblas_int. batch_count >= 0.

  Number of matrices in the batch.

2.6.3.6 General Systems Solvers

rocsolver_<type>getrs()

rocblas_status **rocsolver_zgetrs**(rocblas_handle handle, const rocblas_operation trans, const rocblas_int n, const rocblas_int nrhs, rocblas_double_complex *A, const rocblas_int lda, const rocblas_int *ipiv, rocblas_double_complex *B, const rocblas_int ldb)

rocblas_status **rocsolver_cgetrs**(rocblas_handle handle, const rocblas_operation trans, const rocblas_int n, const rocblas_int nrhs, rocblas_float_complex *A, const rocblas_int lda, const rocblas_int *ipiv, rocblas_float_complex *B, const rocblas_int ldb)

rocblas_status **rocsolver_dgetrs**(rocblas_handle handle, const rocblas_operation trans, const rocblas_int n, const rocblas_int nrhs, double *A, const rocblas_int lda, const rocblas_int *ipiv, double *B, const rocblas_int ldb)

rocblas_status **rocsolver_sgetrs**(rocblas_handle handle, const rocblas_operation trans, const rocblas_int n, const rocblas_int nrhs, float *A, const rocblas_int lda, const rocblas_int *ipiv, float *B, const rocblas_int ldb)

GETRS solves a system of n linear equations on n variables using the LU factorization computed by GETRF.

It solves one of the following systems:

```
A  * X = B (no transpose),
A' * X = B (transpose),  or
A* * X = B (conjugate transpose)
```

depending on the value of trans.

Parameters

- **handle** – **[in]** rocblas_handle.
- **trans** – **[in]** rocblas_operation.

  Specifies the form of the system of equations.
- **n** – **[in]** rocblas_int. n >= 0.

  The order of the system, i.e. the number of columns and rows of A.
- **nrhs** – **[in]** rocblas_int. nrhs >= 0.

  The number of right hand sides, i.e., the number of columns of the matrix B.
- **A** – **[in]** pointer to type. Array on the GPU of dimension lda*n.

  The factors L and U of the factorization A = P*L*U returned by GETRF.
- **lda** – **[in]** rocblas_int. lda >= n.

  The leading dimension of A.
- **ipiv** – **[in]** pointer to rocblas_int. Array on the GPU of dimension n.

  The pivot indices returned by GETRF.

- **B** – **[inout]** pointer to type. Array on the GPU of dimension ldb*nrhs.

  On entry, the right hand side matrix B. On exit, the solution matrix X.

- **ldb** – **[in]** rocblas_int. ldb >= n.

  The leading dimension of B.

rocsolver_<type>getrs_batched()

rocblas_status **rocsolver_zgetrs_batched**(rocblas_handle handle, const rocblas_operation trans, const rocblas_int n, const rocblas_int nrhs, rocblas_double_complex *const A[], const rocblas_int lda, const rocblas_int *ipiv, const rocblas_stride strideP, rocblas_double_complex *const B[], const rocblas_int ldb, const rocblas_int batch_count)

rocblas_status **rocsolver_cgetrs_batched**(rocblas_handle handle, const rocblas_operation trans, const rocblas_int n, const rocblas_int nrhs, rocblas_float_complex *const A[], const rocblas_int lda, const rocblas_int *ipiv, const rocblas_stride strideP, rocblas_float_complex *const B[], const rocblas_int ldb, const rocblas_int batch_count)

rocblas_status **rocsolver_dgetrs_batched**(rocblas_handle handle, const rocblas_operation trans, const rocblas_int n, const rocblas_int nrhs, double *const A[], const rocblas_int lda, const rocblas_int *ipiv, const rocblas_stride strideP, double *const B[], const rocblas_int ldb, const rocblas_int batch_count)

rocblas_status **rocsolver_sgetrs_batched**(rocblas_handle handle, const rocblas_operation trans, const rocblas_int n, const rocblas_int nrhs, float *const A[], const rocblas_int lda, const rocblas_int *ipiv, const rocblas_stride strideP, float *const B[], const rocblas_int ldb, const rocblas_int batch_count)

GETRS_BATCHED solves a batch of systems of n linear equations on n variables using the LU factorization computed by GETRF_BATCHED.

For each instance j in the batch, it solves one of the following systems:

```
A_j  * X_j = B_j (no transpose),
A_j' * X_j = B_j (transpose),  or
A_j* * X_j = B_j (conjugate transpose)
```

depending on the value of trans.

Parameters

- **handle** – **[in]** rocblas_handle.
- **trans** – **[in]** rocblas_operation.

  Specifies the form of the system of equations of each instance in the batch.

- **n** – **[in]** rocblas_int. n >= 0.

  The order of the system, i.e. the number of columns and rows of all A_j matrices.

- **nrhs** – **[in]** rocblas_int. nrhs >= 0.

  The number of right hand sides, i.e., the number of columns of all the matrices B_j.

- **A** – **[in]** Array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n.

  The factors L_j and U_j of the factorization A_j = P_j*L_j*U_j returned by GETRF_BATCHED.

- **lda** – **[in]** rocblas_int. lda >= n.

  The leading dimension of matrices A_j.

- **ipiv** – **[in]** pointer to rocblas_int. Array on the GPU (the size depends on the value of strideP).

  Contains the vectors ipiv_j of pivot indices returned by GETRF_BATCHED.

- **strideP** – **[in]** rocblas_stride.

  Stride from the start of one vector ipiv_j to the next one ipiv_(j+1). There is no restriction for the value of strideP. Normal use case is strideP >= min(m,n).

- **B** – **[inout]** Array of pointers to type. Each pointer points to an array on the GPU of dimension ldb*nrhs.

  On entry, the right hand side matrices B_j. On exit, the solution matrix X_j of each system in the batch.

- **ldb** – **[in]** rocblas_int. ldb >= n.

  The leading dimension of matrices B_j.

- **batch_count** – **[in]** rocblas_int. batch_count >= 0.

  Number of instances (systems) in the batch.

rocsolver_<type>getrs_strided_batched()

rocblas_status **rocsolver_zgetrs_strided_batched**(rocblas_handle handle, const rocblas_operation trans, const rocblas_int n, const rocblas_int nrhs, rocblas_double_complex *A, const rocblas_int lda, const rocblas_stride strideA, const rocblas_int *ipiv, const rocblas_stride strideP, rocblas_double_complex *B, const rocblas_int ldb, const rocblas_stride strideB, const rocblas_int batch_count)

rocblas_status **rocsolver_cgetrs_strided_batched**(rocblas_handle handle, const rocblas_operation trans, const rocblas_int n, const rocblas_int nrhs, rocblas_float_complex *A, const rocblas_int lda, const rocblas_stride strideA, const rocblas_int *ipiv, const rocblas_stride strideP, rocblas_float_complex *B, const rocblas_int ldb, const rocblas_stride strideB, const rocblas_int batch_count)

rocblas_status **rocsolver_dgetrs_strided_batched**(rocblas_handle handle, const rocblas_operation trans, const rocblas_int n, const rocblas_int nrhs, double *A, const rocblas_int lda, const rocblas_stride strideA, const rocblas_int *ipiv, const rocblas_stride strideP, double *B, const rocblas_int ldb, const rocblas_stride strideB, const rocblas_int batch_count)

rocblas_status **rocsolver_sgetrs_strided_batched**(rocblas_handle handle, const rocblas_operation trans, const rocblas_int n, const rocblas_int nrhs, float *A, const rocblas_int lda, const rocblas_stride strideA, const rocblas_int *ipiv, const rocblas_stride strideP, float *B, const rocblas_int ldb, const rocblas_stride strideB, const rocblas_int batch_count)

GETRS_STRIDED_BATCHED solves a batch of systems of n linear equations on n variables using the LU factorization computed by GETRF_STRIDED_BATCHED.

For each instance j in the batch, it solves one of the following systems:

```
A_j  * X_j = B_j (no transpose),
A_j' * X_j = B_j (transpose),  or
A_j* * X_j = B_j (conjugate transpose)
```

depending on the value of trans.

Parameters

- **handle** – **[in]** rocblas_handle.
- **trans** – **[in]** rocblas_operation.

  Specifies the form of the system of equations of each instance in the batch.
- **n** – **[in]** rocblas_int. n >= 0.

  The order of the system, i.e. the number of columns and rows of all A_j matrices.
- **nrhs** – **[in]** rocblas_int. nrhs >= 0.

  The number of right hand sides, i.e., the number of columns of all the matrices B_j.
- **A** – **[in]** pointer to type. Array on the GPU (the size depends on the value of strideA).

  The factors L_j and U_j of the factorization A_j = P_j*L_j*U_j returned by GETRF_STRIDED_BATCHED.
- **lda** – **[in]** rocblas_int. lda >= n.

  The leading dimension of matrices A_j.
- **strideA** – **[in]** rocblas_stride.

  Stride from the start of one matrix A_j and the next one A_(j+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n.
- **ipiv** – **[in]** pointer to rocblas_int. Array on the GPU (the size depends on the value of strideP).

  Contains the vectors ipiv_j of pivot indices returned by GETRF_STRIDED_BATCHED.
- **strideP** – **[in]** rocblas_stride.

  Stride from the start of one vector ipiv_j to the next one ipiv_(j+1). There is no restriction for the value of strideP. Normal use case is strideP >= min(m,n).

- **B** – **[inout]** pointer to type. Array on the GPU (size depends on the value of strideB).

  On entry, the right hand side matrices B_j. On exit, the solution matrix X_j of each system in the batch.

- **ldb** – **[in]** rocblas_int. ldb >= n.

  The leading dimension of matrices B_j.

- **strideB** – **[in]** rocblas_stride.

  Stride from the start of one matrix B_j and the next one B_(j+1). There is no restriction for the value of strideB. Normal use case is strideB >= ldb*nrhs.

- **batch_count** – **[in]** rocblas_int. batch_count >= 0.

  Number of instances (systems) in the batch.

2.6.3.7 General Matrix Singular Value Decomposition

rocsolver_<type>gesvd()

rocblas_status **rocsolver_zgesvd**(rocblas_handle handle, const *rocblas_svect* left_svect, const *rocblas_svect* right_svect, const rocblas_int m, const rocblas_int n, rocblas_double_complex *A, const rocblas_int lda, double *S, rocblas_double_complex *U, const rocblas_int ldu, rocblas_double_complex *V, const rocblas_int ldv, double *E, const *rocblas_workmode* fast_alg, rocblas_int *info)

rocblas_status **rocsolver_cgesvd**(rocblas_handle handle, const *rocblas_svect* left_svect, const *rocblas_svect* right_svect, const rocblas_int m, const rocblas_int n, rocblas_float_complex *A, const rocblas_int lda, float *S, rocblas_float_complex *U, const rocblas_int ldu, rocblas_float_complex *V, const rocblas_int ldv, float *E, const *rocblas_workmode* fast_alg, rocblas_int *info)

rocblas_status **rocsolver_dgesvd**(rocblas_handle handle, const *rocblas_svect* left_svect, const *rocblas_svect* right_svect, const rocblas_int m, const rocblas_int n, double *A, const rocblas_int lda, double *S, double *U, const rocblas_int ldu, double *V, const rocblas_int ldv, double *E, const *rocblas_workmode* fast_alg, rocblas_int *info)

rocblas_status **rocsolver_sgesvd**(rocblas_handle handle, const *rocblas_svect* left_svect, const *rocblas_svect* right_svect, const rocblas_int m, const rocblas_int n, float *A, const rocblas_int lda, float *S, float *U, const rocblas_int ldu, float *V, const rocblas_int ldv, float *E, const *rocblas_workmode* fast_alg, rocblas_int *info)

GESVD computes the Singular Values and optionally the Singular Vectors of a general m-by-n matrix A (Singular Value Decomposition).

The SVD of matrix A is given by:

```
A = U * S * V'
```

where the m-by-n matrix S is zero except, possibly, for its min(m,n) diagonal elements, which are the singular values of A. U and V are orthogonal (unitary) matrices. The first min(m,n) columns of U and V are the left and right singular vectors of A, respectively.

The computation of the singular vectors is optional and it is controlled by the function arguments left_svect and right_svect as described below. When computed, this function returns the transpose (or transpose conjugate) of the right singular vectors, i.e. the rows of V'.

left_svect and right_svect are *rocblas_svect* enums that can take the following values:

- rocblas_svect_all: the entire matrix U (or V') is computed,
- rocblas_svect_singular: only the singular vectors (first min(m,n) columns of U or rows of V') are computed,
- rocblas_svect_overwrite: the first columns (or rows) of A are overwritten with the singular vectors, or
- rocblas_svect_none: no columns (or rows) of U (or V') are computed, i.e. no singular vectors.

left_svect and right_svect cannot both be set to overwrite. When neither is set to overwrite, the contents of A are destroyed by the time the function returns.

Note: When m >> n (or n >> m) the algorithm could be sped up by compressing the matrix A via a QR (or LQ) factorization, and working with the triangular factor afterwards (thin-SVD). If the singular vectors are also requested, its computation could be sped up as well via executing some intermediate operations out-of-place, and relying more on matrix multiplications (GEMMs); this will require, however, a larger memory workspace. The parameter fast_alg controls whether the fast algorithm is executed or not. For more details see the sections "Tuning rocSOLVER performance" and "Memory model" on the User's guide.

Parameters

- **handle** – **[in]** rocblas_handle.
- **left_svect** – **[in]** *rocblas_svect* .

  Specifies how the left singular vectors are computed.
- **right_svect** – **[in]** *rocblas_svect* .

  Specifies how the right singular vectors are computed.
- **m** – **[in]** rocblas_int. m >= 0.

  The number of rows of matrix A.
- **n** – **[in]** rocblas_int. n >= 0.

  The number of columns of matrix A.
- **A** – **[inout]** pointer to type. Array on the GPU of dimension lda*n.

  On entry the matrix A. On exit, if left_svect (or right_svect) is equal to overwrite, the first columns (or rows) contain the left (or right) singular vectors; otherwise, contents of A are destroyed.
- **lda** – **[in]** rocblas_int. lda >= m.

  The leading dimension of A.
- **S** – **[out]** pointer to real type. Array on the GPU of dimension min(m,n).

  The singular values of A in decreasing order.
- **U** – **[out]** pointer to type. Array on the GPU of dimension ldu*min(m,n) if left_svect is set to singular, or ldu*m when left_svect is equal to all.

The matrix of left singular vectors stored as columns. Not referenced if left_svect is set to overwrite or none.

- **ldu** – **[in]** rocblas_int. ldu >= m if left_svect is all or singular; ldu >= 1 otherwise.

  The leading dimension of U.

- **V** – **[out]** pointer to type. Array on the GPU of dimension ldv*n.

  The matrix of right singular vectors stored as rows (transposed / conjugate-transposed). Not referenced if right_svect is set to overwrite or none.

- **ldv** – **[in]** rocblas_int. ldv >= n if right_svect is all; ldv >= min(m,n) if right_svect is set to singular; or ldv >= 1 otherwise.

  The leading dimension of V.

- **E** – **[out]** pointer to real type. Array on the GPU of dimension min(m,n)-1.

  This array is used to work internally with the bidiagonal matrix B associated to A (using BDSQR). On exit, if info > 0, it contains the unconverged off-diagonal elements of B (or properly speaking, a bidiagonal matrix orthogonally equivalent to B). The diagonal elements of this matrix are in S; those that converged correspond to a subset of the singular values of A (not necessarily ordered).

- **fast_alg** – **[in]** *rocblas_workmode* .

  If set to rocblas_outofplace, the function will execute the fast thin-SVD version of the algorithm when possible.

- **info** – **[out]** pointer to a rocblas_int on the GPU.

  If info = 0, successful exit. If info = i > 0, BDSQR did not converge. i elements of E did not converge to zero.

rocsolver_<type>gesvd_batched()

rocblas_status **rocsolver_zgesvd_batched**(rocblas_handle handle, const *rocblas_svect* left_svect, const *rocblas_svect* right_svect, const rocblas_int m, const rocblas_int n, rocblas_double_complex *const A[], const rocblas_int lda, double *S, const rocblas_stride strideS, rocblas_double_complex *U, const rocblas_int ldu, const rocblas_stride strideU, rocblas_double_complex *V, const rocblas_int ldv, const rocblas_stride strideV, double *E, const rocblas_stride strideE, const *rocblas_workmode* fast_alg, rocblas_int *info, const rocblas_int batch_count)

rocblas_status **rocsolver_cgesvd_batched**(rocblas_handle handle, const *rocblas_svect* left_svect, const *rocblas_svect* right_svect, const rocblas_int m, const rocblas_int n, rocblas_float_complex *const A[], const rocblas_int lda, float *S, const rocblas_stride strideS, rocblas_float_complex *U, const rocblas_int ldu, const rocblas_stride strideU, rocblas_float_complex *V, const rocblas_int ldv, const rocblas_stride strideV, float *E, const rocblas_stride strideE, const *rocblas_workmode* fast_alg, rocblas_int *info, const rocblas_int batch_count)

rocblas_status **rocsolver_dgesvd_batched**(rocblas_handle handle, const *rocblas_svect* left_svect, const *rocblas_svect* right_svect, const rocblas_int m, const rocblas_int n, double *const A[], const rocblas_int lda, double *S, const rocblas_stride strideS, double *U, const rocblas_int ldu, const rocblas_stride strideU, double *V, const rocblas_int ldv, const rocblas_stride strideV, double *E, const rocblas_stride strideE, const *rocblas_workmode* fast_alg, rocblas_int *info, const rocblas_int batch_count)

rocblas_status **rocsolver_sgesvd_batched**(rocblas_handle handle, const *rocblas_svect* left_svect, const *rocblas_svect* right_svect, const rocblas_int m, const rocblas_int n, float *const A[], const rocblas_int lda, float *S, const rocblas_stride strideS, float *U, const rocblas_int ldu, const rocblas_stride strideU, float *V, const rocblas_int ldv, const rocblas_stride strideV, float *E, const rocblas_stride strideE, const *rocblas_workmode* fast_alg, rocblas_int *info, const rocblas_int batch_count)

GESVD_BATCHED computes the Singular Values and optionally the Singular Vectors of a batch of general m-by-n matrix A (Singular Value Decomposition).

The SVD of matrix A_j is given by:

```
A_j = U_j * S_j * V_j'
```

where the m-by-n matrix S_j is zero except, possibly, for its min(m,n) diagonal elements, which are the singular values of A_j. U_j and V_j are orthogonal (unitary) matrices. The first min(m,n) columns of U_j and V_j are the left and right singular vectors of A_j, respectively.

The computation of the singular vectors is optional and it is controlled by the function arguments left_svect and right_svect as described below. When computed, this function returns the transpose (or transpose conjugate) of the right singular vectors, i.e. the rows of V_j'.

left_svect and right_svect are *rocblas_svect* enums that can take the following values:

- rocblas_svect_all: the entire matrix U_j (or V_j') is computed,
- rocblas_svect_singular: only the singular vectors (first min(m,n) columns of U_j or rows of V_j') are computed,
- rocblas_svect_overwrite: the first columns (or rows) of A_j are overwritten with the singular vectors, or
- rocblas_svect_none: no columns (or rows) of U_j (or V_j') are computed, i.e. no singular vectors.

left_svect and right_svect cannot both be set to overwrite. When neither is set to overwrite, the contents of A_j are destroyed by the time the function returns.

Note: When m >> n (or n >> m) the algorithm could be sped up by compressing the matrix A_j via a QR (or LQ) factorization, and working with the triangular factor afterwards (thin-SVD). If the singular vectors are also requested, its computation could be sped up as well via executing some intermediate operations out-of-place, and relying more on matrix multiplications (GEMMs); this will require, however, a larger memory workspace. The parameter fast_alg controls whether the fast algorithm is executed or not. For more details see the sections "Tuning rocSOLVER performance" and "Memory model" on the User's guide.

Parameters

- **handle** – **[in]** rocblas_handle.

- **left_svect** – **[in]** *rocblas_svect* .

  Specifies how the left singular vectors are computed.

- **right_svect** – **[in]** *rocblas_svect* .

  Specifies how the right singular vectors are computed.

- **m** – **[in]** rocblas_int. m >= 0.

  The number of rows of all matrices A_j in the batch.

- **n** – **[in]** rocblas_int. n >= 0.

  The number of columns of all matrices A_j in the batch.

- **A** – **[inout]** Array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n.

  On entry the matrices A_j. On exit, if left_svect (or right_svect) is equal to overwrite, the first columns (or rows) of A_j contain the left (or right) corresponding singular vectors; otherwise, contents of A_j are destroyed.

- **lda** – **[in]** rocblas_int. lda >= m.

  The leading dimension of A_j.

- **S** – **[out]** pointer to real type. Array on the GPU (the size depends on the value of strideS).

  The singular values of A_j in decreasing order.

- **strideS** – **[in]** rocblas_stride.

  Stride from the start of one vector S_j to the next one S_(j+1). There is no restriction for the value of strideS. Normal use case is strideS >= min(m,n).

- **U** – **[out]** pointer to type. Array on the GPU (the side depends on the value of strideU).

  The matrices U_j of left singular vectors stored as columns. Not referenced if left_svect is set to overwrite or none.

- **ldu** – **[in]** rocblas_int. ldu >= m if left_svect is all or singular; ldu >= 1 otherwise.

  The leading dimension of U_j.

- **strideU** – **[in]** rocblas_stride.

  Stride from the start of one matrix U_j to the next one U_(j+1). There is no restriction for the value of strideU. Normal use case is strideU >= ldu*min(m,n) if left_svect is set to singular, or strideU >= ldu*m when left_svect is equal to all.

- **V** – **[out]** pointer to type. Array on the GPU (the size depends on the value of strideV).

  The matrices V_j of right singular vectors stored as rows (transposed / conjugate-transposed). Not referenced if right_svect is set to overwrite or none.

- **ldv** – **[in]** rocblas_int. ldv >= n if right_svect is all; ldv >= min(m,n) if right_svect is set to singular; or ldv >= 1 otherwise.

  The leading dimension of V.

- **strideV** – **[in]** rocblas_stride.

  Stride from the start of one matrix V_j to the next one V_(j+1). There is no restriction for the value of strideV. Normal use case is strideV >= ldv*n.

- **E** – **[out]** pointer to real type. Array on the GPU (the size depends on the value of strideE).

  This array is used to work internally with the bidiagonal matrix B_j associated to A_j (using BDSQR). On exit, if info > 0, it contains the unconverged off-diagonal elements of B_j (or properly speaking, a bidiagonal matrix orthogonally equivalent to B_j). The diagonal elements of this matrix are in S_j; those that converged correspond to a subset of the singular values of A_j (not necessarily ordered).

- **strideE** – **[in]** rocblas_stride.

  Stride from the start of one vector E_j to the next one E_(j+1). There is no restriction for the value of strideE. Normal use case is strideE >= min(m,n)-1.

- **fast_alg** – **[in]** *rocblas_workmode* .

  If set to rocblas_outofplace, the function will execute the fast thin-SVD version of the algorithm when possible.

- **info** – **[out]** pointer to a rocblas_int on the GPU.

  If info = 0, successful exit. If info = i > 0, BDSQR did not converge. i elements of E did not converge to zero.

- **batch_count** – **[in]** rocblas_int. batch_count >= 0.

  Number of matrices in the batch.

rocsolver_<type>gesvd_strided_batched()

rocblas_status **rocsolver_zgesvd_strided_batched**(rocblas_handle handle, const *rocblas_svect* left_svect, const *rocblas_svect* right_svect, const rocblas_int m, const rocblas_int n, rocblas_double_complex *A, const rocblas_int lda, const rocblas_stride strideA, double *S, const rocblas_stride strideS, rocblas_double_complex *U, const rocblas_int ldu, const rocblas_stride strideU, rocblas_double_complex *V, const rocblas_int ldv, const rocblas_stride strideV, double *E, const rocblas_stride strideE, const *rocblas_workmode* fast_alg, rocblas_int *info, const rocblas_int batch_count)

rocblas_status **rocsolver_cgesvd_strided_batched**(rocblas_handle handle, const *rocblas_svect* left_svect, const *rocblas_svect* right_svect, const rocblas_int m, const rocblas_int n, rocblas_float_complex *A, const rocblas_int lda, const rocblas_stride strideA, float *S, const rocblas_stride strideS, rocblas_float_complex *U, const rocblas_int ldu, const rocblas_stride strideU, rocblas_float_complex *V, const rocblas_int ldv, const rocblas_stride strideV, float *E, const rocblas_stride strideE, const *rocblas_workmode* fast_alg, rocblas_int *info, const rocblas_int batch_count)

rocblas_status **rocsolver_dgesvd_strided_batched**(rocblas_handle handle, const *rocblas_svect* left_svect, const *rocblas_svect* right_svect, const rocblas_int m, const rocblas_int n, double *A, const rocblas_int lda, const rocblas_stride strideA, double *S, const rocblas_stride strideS, double *U, const rocblas_int ldu, const rocblas_stride strideU, double *V, const rocblas_int ldv, const rocblas_stride strideV, double *E, const rocblas_stride strideE, const *rocblas_workmode* fast_alg, rocblas_int *info, const rocblas_int batch_count)

rocblas_status **rocsolver_sgesvd_strided_batched**(rocblas_handle handle, const *rocblas_svect* left_svect, const *rocblas_svect* right_svect, const rocblas_int m, const rocblas_int n, float *A, const rocblas_int lda, const rocblas_stride strideA, float *S, const rocblas_stride strideS, float *U, const rocblas_int ldu, const rocblas_stride strideU, float *V, const rocblas_int ldv, const rocblas_stride strideV, float *E, const rocblas_stride strideE, const *rocblas_workmode* fast_alg, rocblas_int *info, const rocblas_int batch_count)

GESVD_STRIDED_BATCHED computes the Singular Values and optionally the Singular Vectors of a batch of general m-by-n matrix A (Singular Value Decomposition).

The SVD of matrix A_j is given by:

```
A_j = U_j * S_j * V_j'
```

where the m-by-n matrix S_j is zero except, possibly, for its min(m,n) diagonal elements, which are the singular values of A_j. U_j and V_j are orthogonal (unitary) matrices. The first min(m,n) columns of U_j and V_j are the left and right singular vectors of A_j, respectively.

The computation of the singular vectors is optional and it is controlled by the function arguments left_svect and right_svect as described below. When computed, this function returns the transpose (or transpose conjugate) of the right singular vectors, i.e. the rows of V_j'.

left_svect and right_svect are *rocblas_svect* enums that can take the following values:

- rocblas_svect_all: the entire matrix U_j (or V_j') is computed,
- rocblas_svect_singular: only the singular vectors (first min(m,n) columns of U_j or rows of V_j') are computed,
- rocblas_svect_overwrite: the first columns (or rows) of A_j are overwritten with the singular vectors, or
- rocblas_svect_none: no columns (or rows) of U_j (or V_j') are computed, i.e. no singular vectors.

left_svect and right_svect cannot both be set to overwrite. When neither is set to overwrite, the contents of A_j are destroyed by the time the function returns.

Note: When m >> n (or n >> m) the algorithm could be sped up by compressing the matrix A_j via a QR (or LQ) factorization, and working with the triangular factor afterwards (thin-SVD). If the singular vectors are also requested, its computation could be sped up as well via executing some intermediate operations out-of-place, and relying more on matrix multiplications (GEMMs); this will require, however, a larger memory workspace. The parameter fast_alg controls whether the fast algorithm is executed or not. For more details see the sections "Tuning rocSOLVER performance" and "Memory model" on the User's guide.

Parameters

- **handle** – **[in]** rocblas_handle.
- **left_svect** – **[in]** *rocblas_svect* .

  Specifies how the left singular vectors are computed.
- **right_svect** – **[in]** *rocblas_svect* .

  Specifies how the right singular vectors are computed.
- **m** – **[in]** rocblas_int. m >= 0.

  The number of rows of all matrices A_j in the batch.
- **n** – **[in]** rocblas_int. n >= 0.

  The number of columns of all matrices A_j in the batch.
- **A** – **[inout]** pointer to type. Array on the GPU (the size depends on the value of strideA).

  On entry the matrices A_j. On exit, if left_svect (or right_svect) is equal to overwrite, the first columns (or rows) of A_j contain the left (or right) corresponding singular vectors; otherwise, contents of A_j are destroyed.
- **lda** – **[in]** rocblas_int. lda >= m.

  The leading dimension of A_j.
- **strideA** – **[in]** rocblas_stride.

  Stride from the start of one matrix A_j to the next one A_(j+1). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n.
- **S** – **[out]** pointer to real type. Array on the GPU (the size depends on the value of strideS).

  The singular values of A_j in decreasing order.
- **strideS** – **[in]** rocblas_stride.

  Stride from the start of one vector S_j to the next one S_(j+1). There is no restriction for the value of strideS. Normal use case is strideS >= min(m,n).
- **U** – **[out]** pointer to type. Array on the GPU (the side depends on the value of strideU).

  The matrices U_j of left singular vectors stored as columns. Not referenced if left_svect is set to overwrite or none.
- **ldu** – **[in]** rocblas_int. ldu >= m if left_svect is all or singular; ldu >= 1 otherwise.

  The leading dimension of U_j.
- **strideU** – **[in]** rocblas_stride.

  Stride from the start of one matrix U_j to the next one U_(j+1). There is no restriction for the value of strideU. Normal use case is strideU >= ldu*min(m,n) if left_svect is set to singular, or strideU >= ldu*m when left_svect is equal to all.
- **V** – **[out]** pointer to type. Array on the GPU (the size depends on the value of strideV).

  The matrices V_j of right singular vectors stored as rows (transposed / conjugate-transposed). Not referenced if right_svect is set to overwrite or none.
- **ldv** – **[in]** rocblas_int. ldv >= n if right_svect is all; ldv >= min(m,n) if right_svect is set to singular; or ldv >= 1 otherwise.

  The leading dimension of V.

- **strideV** – **[in]** rocblas_stride.

  Stride from the start of one matrix V_j to the next one V_(j+1). There is no restriction for the value of strideV. Normal use case is strideV >= ldv*n.

- **E** – **[out]** pointer to real type. Array on the GPU (the size depends on the value of strideE).

  This array is used to work internally with the bidiagonal matrix B_j associated to A_j (using BDSQR). On exit, if info > 0, it contains the unconverged off-diagonal elements of B_j (or properly speaking, a bidiagonal matrix orthogonally equivalent to B_j). The diagonal elements of this matrix are in S_j; those that converged correspond to a subset of the singular values of A_j (not necessarily ordered).

- **strideE** – **[in]** rocblas_stride.

  Stride from the start of one vector E_j to the next one E_(j+1). There is no restriction for the value of strideE. Normal use case is strideE >= min(m,n)-1.

- **fast_alg** – **[in]** *rocblas_workmode* .

  If set to rocblas_outofplace, the function will execute the fast thin-SVD version of the algorithm when possible.

- **info** – **[out]** pointer to a rocblas_int on the GPU.

  If info = 0, successful exit. If info = i > 0, BDSQR did not converge. i elements of E did not converge to zero.

- **batch_count** – **[in]** rocblas_int. batch_count >= 0.

  Number of matrices in the batch.

2.6.4 Lapack-like Functions

Other Lapack-like routines provided by rocSOLVER.

2.6.4.1 General Matrix Factorizations

rocsolver_<type>getf2_npvt()

rocblas_status **rocsolver_zgetf2_npvt**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_double_complex *A, const rocblas_int lda, rocblas_int *info)

rocblas_status **rocsolver_cgetf2_npvt**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_float_complex *A, const rocblas_int lda, rocblas_int *info)

rocblas_status **rocsolver_dgetf2_npvt**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, double *A, const rocblas_int lda, rocblas_int *info)

rocblas_status **rocsolver_sgetf2_npvt**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, float *A, const rocblas_int lda, rocblas_int *info)

GETF2_NPVT computes the LU factorization of a general m-by-n matrix A without partial pivoting.

(This is the unblocked Level-2-BLAS version of the algorithm. An optimized internal implementation without rocBLAS calls could be executed with small and mid-size matrices if optimizations are enabled (default option). For more details see the section "tuning rocSOLVER performance" on the User's guide).

The factorization has the form

```
A = L * U
```

where L is lower triangular with unit diagonal elements (lower trapezoidal if m > n), and U is upper triangular (upper trapezoidal if m < n).

Note: Although this routine can offer better performance, Gaussian elimination without pivoting is not backward stable. If numerical accuracy is compromised, use the legacy-LAPACK-like API GETF2 routines instead.

Parameters

- **handle** – **[in]** rocblas_handle.
- **m** – **[in]** rocblas_int. m >= 0.

  The number of rows of the matrix A.
- **n** – **[in]** rocblas_int. n >= 0.

  The number of columns of the matrix A.
- **A** – **[inout]** pointer to type. Array on the GPU of dimension lda*n.

  On entry, the m-by-n matrix A to be factored. On exit, the factors L and U from the factorization. The unit diagonal elements of L are not stored.
- **lda** – **[in]** rocblas_int. lda >= m.

  Specifies the leading dimension of A.
- **info** – **[out]** pointer to a rocblas_int on the GPU.

  If info = 0, successful exit. If info = i > 0, U is singular. U(i,i) is the first zero element in the diagonal. The factorization from this point might be incomplete.

rocsolver_<type>getf2_npvt_batched()

rocblas_status **rocsolver_zgetf2_npvt_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_double_complex *const A[], const rocblas_int lda, rocblas_int *info, const rocblas_int batch_count)

rocblas_status **rocsolver_cgetf2_npvt_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_float_complex *const A[], const rocblas_int lda, rocblas_int *info, const rocblas_int batch_count)

rocblas_status **rocsolver_dgetf2_npvt_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, double *const A[], const rocblas_int lda, rocblas_int *info, const rocblas_int batch_count)

rocblas_status **rocsolver_sgetf2_npvt_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, float *const A[], const rocblas_int lda, rocblas_int *info, const rocblas_int batch_count)

GETF2_NPVT_BATCHED computes the LU factorization of a batch of general m-by-n matrices without partial pivoting.

(This is the unblocked Level-2-BLAS version of the algorithm. An optimized internal implementation without rocBLAS calls could be executed with small and mid-size matrices if optimizations are enabled (default option). For more details see the section "tuning rocSOLVER performance" on the User's guide).

The factorization of matrix A_i in the batch has the form

```
A_i = L_i * U_i
```

where L_i is lower triangular with unit diagonal elements (lower trapezoidal if m > n), and U_i is upper triangular (upper trapezoidal if m < n).

Note: Although this routine can offer better performance, Gaussian elimination without pivoting is not backward stable. If numerical accuracy is compromised, use the legacy-LAPACK-like API GETF2 routines instead.

Parameters

- **handle** – **[in]** rocblas_handle.
- **m** – **[in]** rocblas_int. m >= 0.

  The number of rows of all matrices A_i in the batch.
- **n** – **[in]** rocblas_int. n >= 0.

  The number of columns of all matrices A_i in the batch.
- **A** – **[inout]** array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n.

  On entry, the m-by-n matrices A_i to be factored. On exit, the factors L_i and U_i from the factorizations. The unit diagonal elements of L_i are not stored.
- **lda** – **[in]** rocblas_int. lda >= m.

  Specifies the leading dimension of matrices A_i.
- **info** – **[out]** pointer to rocblas_int. Array of batch_count integers on the GPU.

  If info_i = 0, successful exit for factorization of A_i. If info_i = j > 0, U_i is singular. U_i(j,j) is the first zero element in the diagonal. The factorization from this point might be incomplete.
- **batch_count** – **[in]** rocblas_int. batch_count >= 0.

  Number of matrices in the batch.

rocsolver_<type>getf2_npvt_strided_batched()

rocblas_status **rocsolver_zgetf2_npvt_strided_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_double_complex *A, const rocblas_int lda, const rocblas_stride strideA, rocblas_int *info, const rocblas_int batch_count)

rocblas_status **rocsolver_cgetf2_npvt_strided_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_float_complex *A, const rocblas_int lda, const rocblas_stride strideA, rocblas_int *info, const rocblas_int batch_count)

rocblas_status **rocsolver_dgetf2_npvt_strided_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, double *A, const rocblas_int lda, const rocblas_stride strideA, rocblas_int *info, const rocblas_int batch_count)

rocblas_status **rocsolver_sgetf2_npvt_strided_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, float *A, const rocblas_int lda, const rocblas_stride strideA, rocblas_int *info, const rocblas_int batch_count)

GETF2_NPVT_STRIDED_BATCHED computes the LU factorization of a batch of general m-by-n matrices without partial pivoting.

(This is the unblocked Level-2-BLAS version of the algorithm. An optimized internal implementation without rocBLAS calls could be executed with small and mid-size matrices if optimizations are enabled (default option). For more details see the section "tuning rocSOLVER performance" on the User's guide).

The factorization of matrix A_i in the batch has the form

```
A_i = L_i * U_i
```

where L_i is lower triangular with unit diagonal elements (lower trapezoidal if m > n), and U_i is upper triangular (upper trapezoidal if m < n).

Note: Although this routine can offer better performance, Gaussian elimination without pivoting is not backward stable. If numerical accuracy is compromised, use the legacy-LAPACK-like API GETF2 routines instead.

Parameters

- **handle** – **[in]** rocblas_handle.
- **m** – **[in]** rocblas_int. m >= 0.

  The number of rows of all matrices A_i in the batch.
- **n** – **[in]** rocblas_int. n >= 0.

  The number of columns of all matrices A_i in the batch.
- **A** – **[inout]** pointer to type. Array on the GPU (the size depends on the value of strideA).

  On entry, the m-by-n matrices A_i to be factored. On exit, the factors L_i and U_i from the factorization. The unit diagonal elements of L_i are not stored.
- **lda** – **[in]** rocblas_int. lda >= m.

  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). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n
- **info** – **[out]** pointer to rocblas_int. Array of batch_count integers on the GPU.

  If info_i = 0, successful exit for factorization of A_i. If info_i = j > 0, U_i is singular. U_i(j,j) is the first zero element in the diagonal. The factorization from this point might be incomplete.
- **batch_count** – **[in]** rocblas_int. batch_count >= 0.

  Number of matrices in the batch.

rocsolver_<type>getrf_npvt()

rocblas_status **rocsolver_zgetrf_npvt**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_double_complex *A, const rocblas_int lda, rocblas_int *info)

rocblas_status **rocsolver_cgetrf_npvt**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_float_complex *A, const rocblas_int lda, rocblas_int *info)

rocblas_status **rocsolver_dgetrf_npvt**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, double *A, const rocblas_int lda, rocblas_int *info)

rocblas_status **rocsolver_sgetrf_npvt**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, float *A, const rocblas_int lda, rocblas_int *info)

GETRF_NPVT computes the LU factorization of a general m-by-n matrix A without partial pivoting.

(This is the blocked Level-3-BLAS version of the algorithm. An optimized internal implementation without rocBLAS calls could be executed with mid-size matrices if optimizations are enabled (default option). For more details see the section "tuning rocSOLVER performance" on the User's guide).

The factorization has the form

```
A = L * U
```

where L is lower triangular with unit diagonal elements (lower trapezoidal if m > n), and U is upper triangular (upper trapezoidal if m < n).

Note: Although this routine can offer better performance, Gaussian elimination without pivoting is not backward stable. If numerical accuracy is compromised, use the legacy-LAPACK-like API GETRF routines instead.

Parameters

- **handle** – **[in]** rocblas_handle.
- **m** – **[in]** rocblas_int. m >= 0.

  The number of rows of the matrix A.
- **n** – **[in]** rocblas_int. n >= 0.

  The number of columns of the matrix A.
- **A** – **[inout]** pointer to type. Array on the GPU of dimension lda*n.

  On entry, the m-by-n matrix A to be factored. On exit, the factors L and U from the factorization. The unit diagonal elements of L are not stored.
- **lda** – **[in]** rocblas_int. lda >= m.

  Specifies the leading dimension of A.
- **info** – **[out]** pointer to a rocblas_int on the GPU.

  If info = 0, successful exit. If info = i > 0, U is singular. U(i,i) is the first zero element in the diagonal. The factorization from this point might be incomplete.

rocsolver_<type>getrf_npvt_batched()

rocblas_status **rocsolver_zgetrf_npvt_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_double_complex *const A[], const rocblas_int lda, rocblas_int *info, const rocblas_int batch_count)

rocblas_status **rocsolver_cgetrf_npvt_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_float_complex *const A[], const rocblas_int lda, rocblas_int *info, const rocblas_int batch_count)

rocblas_status **rocsolver_dgetrf_npvt_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, double *const A[], const rocblas_int lda, rocblas_int *info, const rocblas_int batch_count)

rocblas_status **rocsolver_sgetrf_npvt_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, float *const A[], const rocblas_int lda, rocblas_int *info, const rocblas_int batch_count)

GETRF_NPVT_BATCHED computes the LU factorization of a batch of general m-by-n matrices without partial pivoting.

(This is the blocked Level-3-BLAS version of the algorithm. An optimized internal implementation without rocBLAS calls could be executed with mid-size matrices if optimizations are enabled (default option). For more details see the section "tuning rocSOLVER performance" on the User's guide).

The factorization of matrix A_i in the batch has the form

```
A_i = L_i * U_i
```

where L_i is lower triangular with unit diagonal elements (lower trapezoidal if m > n), and U_i is upper triangular (upper trapezoidal if m < n).

Note: Although this routine can offer better performance, Gaussian elimination without pivoting is not backward stable. If numerical accuracy is compromised, use the legacy-LAPACK-like API GETRF routines instead.

Parameters

- **handle** – **[in]** rocblas_handle.
- **m** – **[in]** rocblas_int. m >= 0.

  The number of rows of all matrices A_i in the batch.
- **n** – **[in]** rocblas_int. n >= 0.

  The number of columns of all matrices A_i in the batch.
- **A** – **[inout]** array of pointers to type. Each pointer points to an array on the GPU of dimension lda*n.

  On entry, the m-by-n matrices A_i to be factored. On exit, the factors L_i and U_i from the factorizations. The unit diagonal elements of L_i are not stored.
- **lda** – **[in]** rocblas_int. lda >= m.

  Specifies the leading dimension of matrices A_i.
- **info** – **[out]** pointer to rocblas_int. Array of batch_count integers on the GPU.

If info_i = 0, successful exit for factorization of A_i. If info_i = j > 0, U_i is singular. U_i(j,j) is the first zero element in the diagonal. The factorization from this point might be incomplete.

- **batch_count** – **[in]** rocblas_int. batch_count >= 0.

  Number of matrices in the batch.

rocsolver_<type>getrf_npvt_strided_batched()

rocblas_status **rocsolver_zgetrf_npvt_strided_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_double_complex *A, const rocblas_int lda, const rocblas_stride strideA, rocblas_int *info, const rocblas_int batch_count)

rocblas_status **rocsolver_cgetrf_npvt_strided_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, rocblas_float_complex *A, const rocblas_int lda, const rocblas_stride strideA, rocblas_int *info, const rocblas_int batch_count)

rocblas_status **rocsolver_dgetrf_npvt_strided_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, double *A, const rocblas_int lda, const rocblas_stride strideA, rocblas_int *info, const rocblas_int batch_count)

rocblas_status **rocsolver_sgetrf_npvt_strided_batched**(rocblas_handle handle, const rocblas_int m, const rocblas_int n, float *A, const rocblas_int lda, const rocblas_stride strideA, rocblas_int *info, const rocblas_int batch_count)

GETRF_NPVT_STRIDED_BATCHED computes the LU factorization of a batch of general m-by-n matrices without partial pivoting.

(This is the blocked Level-3-BLAS version of the algorithm. An optimized internal implementation without rocBLAS calls could be executed with mid-size matrices if optimizations are enabled (default option). For more details see the section "tuning rocSOLVER performance" on the User's guide).

The factorization of matrix A_i in the batch has the form

```
A_i = L_i * U_i
```

where L_i is lower triangular with unit diagonal elements (lower trapezoidal if m > n), and U_i is upper triangular (upper trapezoidal if m < n).

Note: Although this routine can offer better performance, Gaussian elimination without pivoting is not backward stable. If numerical accuracy is compromised, use the legacy-LAPACK-like API GETRF routines instead.

Parameters

- **handle** – **[in]** rocblas_handle.
- **m** – **[in]** rocblas_int. m >= 0.

  The number of rows of all matrices A_i in the batch.
- **n** – **[in]** rocblas_int. n >= 0.

  The number of columns of all matrices A_i in the batch.

- **A** – **[inout]** pointer to type. Array on the GPU (the size depends on the value of strideA).

  On entry, the m-by-n matrices A_i to be factored. On exit, the factors L_i and U_i from the factorization. The unit diagonal elements of L_i are not stored.

- **lda** – **[in]** rocblas_int. lda >= m.

  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). There is no restriction for the value of strideA. Normal use case is strideA >= lda*n

- **info** – **[out]** pointer to rocblas_int. Array of batch_count integers on the GPU.

  If info_i = 0, successful exit for factorization of A_i. If info_i = j > 0, U_i is singular. U_i(j,j) is the first zero element in the diagonal. The factorization from this point might be incomplete.

- **batch_count** – **[in]** rocblas_int. batch_count >= 0.

  Number of matrices in the batch.

2.6.5 Deprecated

2.6.5.1 Types

See the rocBLAS types documentation for information on the suggested replacements.

rocsolver_int

typedef rocblas_int **rocsolver_int**

Deprecated:
Use `rocblas_int`.

Deprecated since version 3.5: Use `rocblas_int`.

rocsolver_handle

typedef rocblas_handle **rocsolver_handle**

Deprecated:
Use `rocblas_handle`.

Deprecated since version 3.5: Use `rocblas_handle`.

rocsolver_direction

typedef *rocblas_direct* **rocsolver_direction**

Deprecated:
Use `rocblas_direct`

Deprecated since version 3.5: Use *rocblas_direct*.

rocsolver_storev

typedef *rocblas_storev* **rocsolver_storev**

Deprecated:
Use `rocblas_storev`.

Deprecated since version 3.5: Use *rocblas_storev*.

rocsolver_operation

typedef rocblas_operation **rocsolver_operation**

Deprecated:
Use `rocblas_operation`.

Deprecated since version 3.5: Use `rocblas_operation`.

rocsolver_fill

typedef rocblas_fill **rocsolver_fill**

Deprecated:
Use `rocblas_fill`.

Deprecated since version 3.5: Use `rocblas_fill`.

rocsolver_diagonal

typedef rocblas_diagonal **rocsolver_diagonal**

Deprecated:
Use `rocblas_diagonal`.

Deprecated since version 3.5: Use `rocblas_diagonal`.

rocsolver_side

typedef rocblas_side **rocsolver_side**

Deprecated:
Use `rocblas_stide`.

Deprecated since version 3.5: Use `rocblas_side`.

rocsolver_status

typedef rocblas_status **rocsolver_status**

Deprecated:
Use `rocblas_status`.

Deprecated since version 3.5: Use `rocblas_status`.

2.6.5.2 Auxiliary Functions

See the rocBLAS auxiliary functions documentation for information on the suggested replacements.

rocsolver_create_handle()

rocsolver_status **rocsolver_create_handle**(*rocsolver_handle* *handle)
Creates a handle and sets the pointer mode to `rocblas_pointer_mode_device`.

Deprecated:
Use `rocblas_create_handle`.

Deprecated since version 3.5: Use `rocblas_create_handle()`.

rocsolver_destroy_handle()

rocsolver_status **rocsolver_destroy_handle**(*rocsolver_handle* handle)

Deprecated:
Use `rocblas_destroy_handle`.

Deprecated since version 3.5: Use `rocblas_destroy_handle()`.

rocsolver_set_stream()

rocsolver_status **rocsolver_set_stream**(*rocsolver_handle* handle, hipStream_t stream)

Deprecated:
Use `rocblas_set_stream`.

Deprecated since version 3.5: Use `rocblas_set_stream()`.

rocsolver_get_stream()

rocsolver_status **rocsolver_get_stream**(*rocsolver_handle* handle, hipStream_t *stream)

Deprecated:
Use `rocblas_get_stream`.

Deprecated since version 3.5: Use `rocblas_get_stream()`.

rocsolver_set_vector()

rocsolver_status **rocsolver_set_vector**(*rocsolver_int* n, *rocsolver_int* elem_size, const void *x, *rocsolver_int* incx, void *y, *rocsolver_int* incy)

Deprecated:
Use `rocblas_set_vector`.

Deprecated since version 3.5: Use `rocblas_set_vector()`.

rocsolver_get_vector()

rocsolver_status **rocsolver_get_vector**(*rocsolver_int* n, *rocsolver_int* elem_size, const void *x, *rocsolver_int* incx, void *y, *rocsolver_int* incy)

Deprecated:
Use `rocblas_get_vector`.

Deprecated since version 3.5: Use `rocblas_get_vector()`.

rocsolver_set_matrix()

rocsolver_status **rocsolver_set_matrix**(*rocsolver_int* rows, *rocsolver_int* cols, *rocsolver_int* elem_size, const void *a, *rocsolver_int* lda, void *b, *rocsolver_int* ldb)

Deprecated:
Use `rocblas_set_matrix`.

Deprecated since version 3.5: Use `rocblas_set_matrix()`.

rocsolver_get_matrix()

rocsolver_status **rocsolver_get_matrix**(*rocsolver_int* rows, *rocsolver_int* cols, *rocsolver_int* elem_size, const void *a, *rocsolver_int* lda, void *b, *rocsolver_int* ldb)

Deprecated:
Use `rocblas_get_matrix`.

Deprecated since version 3.5: Use `rocblas_get_matrix()`.

CHAPTER

THREE

LIBRARY DESIGN DOCUMENTATION

3.1 Introduction

More to come later…

3.2 Batch rocSOLVER

More to come later…

3.3 Clients

rocSOLVER has a basic/preliminary infrastructure for testing and benchmarking similar to that of rocBLAS.

On a normal installation, client binaries `rocsolver-test` and `rocsolver-bench` should be located in the directory **<rocsolverDIR>/build/clients/staging**.

3.3.1 Testing rocSOLVER

`rocsolver-test` executes a suite of Google tests (*gtest*) that verifies the correct functioning of the library; the results computed by rocSOLVER, for random input data, are compared with the results computed by NETLib LAPACK on the CPU.

Calling the rocSOLVER gtest client with the –help flag

```
./rocsolver-test --help
```

returns information on different flags that control the behavior of the gtests.

3.3.2 Benchmarking rocSOLVER

`rocsolver-bench` runs any rocSOLVER function with random data of the specified dimensions; it compares the computed results, and provides basic performance information (as for now, execution times).

Similarly,

```
./rocsolver-bench --help
```

returns information on how to use the rocSOLVER benchmark client.

INDEX

R

rocblas_direct (*C enum*), 16
rocblas_direct.rocblas_backward_direction (*C enumerator*), 16
rocblas_direct.rocblas_forward_direction (*C enumerator*), 16
rocblas_evect (*C enum*), 17
rocblas_evect.rocblas_evect_none (*C enumerator*), 17
rocblas_evect.rocblas_evect_original (*C enumerator*), 17
rocblas_evect.rocblas_evect_tridiagonal (*C enumerator*), 17
rocblas_storev (*C enum*), 16
rocblas_storev.rocblas_column_wise (*C enumerator*), 16
rocblas_storev.rocblas_row_wise (*C enumerator*), 16
rocblas_svect (*C enum*), 16
rocblas_svect.rocblas_svect_all (*C enumerator*), 16
rocblas_svect.rocblas_svect_none (*C enumerator*), 16
rocblas_svect.rocblas_svect_overwrite (*C enumerator*), 16
rocblas_svect.rocblas_svect_singular (*C enumerator*), 16
rocblas_workmode (*C enum*), 17
rocblas_workmode.rocblas_inplace (*C enumerator*), 17
rocblas_workmode.rocblas_outofplace (*C enumerator*), 17
rocsolver_cbdsqr (*C function*), 26
rocsolver_cgebd2 (*C function*), 106
rocsolver_cgebd2_batched (*C function*), 108
rocsolver_cgebd2_strided_batched (*C function*), 110
rocsolver_cgebrd (*C function*), 112
rocsolver_cgebrd_batched (*C function*), 113
rocsolver_cgebrd_strided_batched (*C function*), 115
rocsolver_cgelq2 (*C function*), 94
rocsolver_cgelq2_batched (*C function*), 95
rocsolver_cgelq2_strided_batched (*C function*), 96
rocsolver_cgelqf (*C function*), 97
rocsolver_cgelqf_batched (*C function*), 98
rocsolver_cgelqf_strided_batched (*C function*), 100
rocsolver_cgels (*C function*), 101
rocsolver_cgels_batched (*C function*), 103
rocsolver_cgels_strided_batched (*C function*), 104
rocsolver_cgeql2 (*C function*), 86
rocsolver_cgeql2_batched (*C function*), 87
rocsolver_cgeql2_strided_batched (*C function*), 88
rocsolver_cgeqlf (*C function*), 90
rocsolver_cgeqlf_batched (*C function*), 91
rocsolver_cgeqlf_strided_batched (*C function*), 92
rocsolver_cgeqr2 (*C function*), 79
rocsolver_cgeqr2_batched (*C function*), 80
rocsolver_cgeqr2_strided_batched (*C function*), 81
rocsolver_cgeqrf (*C function*), 82
rocsolver_cgeqrf_batched (*C function*), 83
rocsolver_cgeqrf_strided_batched (*C function*), 85
rocsolver_cgesvd (*C function*), 142
rocsolver_cgesvd_batched (*C function*), 144
rocsolver_cgesvd_strided_batched (*C function*), 147
rocsolver_cgetf2 (*C function*), 71
rocsolver_cgetf2_batched (*C function*), 72
rocsolver_cgetf2_npvt (*C function*), 150
rocsolver_cgetf2_npvt_batched (*C function*), 151
rocsolver_cgetf2_npvt_strided_batched (*C function*), 152
rocsolver_cgetf2_strided_batched (*C function*), 73
rocsolver_cgetrf (*C function*), 75
rocsolver_cgetrf_batched (*C function*), 76
rocsolver_cgetrf_npvt (*C function*), 154

rocsolver_cgetrf_npvt_batched (*C function*), 155
rocsolver_cgetrf_npvt_strided_batched (*C function*), 156
rocsolver_cgetrf_strided_batched (*C function*), 77
rocsolver_cgetri (*C function*), 134
rocsolver_cgetri_batched (*C function*), 135
rocsolver_cgetri_strided_batched (*C function*), 136
rocsolver_cgetrs (*C function*), 138
rocsolver_cgetrs_batched (*C function*), 139
rocsolver_cgetrs_strided_batched (*C function*), 140
rocsolver_chetd2 (*C function*), 122
rocsolver_chetd2_batched (*C function*), 123
rocsolver_chetd2_strided_batched (*C function*), 124
rocsolver_chetrd (*C function*), 130
rocsolver_chetrd_batched (*C function*), 131
rocsolver_chetrd_strided_batched (*C function*), 133
rocsolver_clabrd (*C function*), 24
rocsolver_clacgv (*C function*), 17
rocsolver_clarf (*C function*), 21
rocsolver_clarfb (*C function*), 22
rocsolver_clarfg (*C function*), 19
rocsolver_clarft (*C function*), 20
rocsolver_claswp (*C function*), 18
rocsolver_clatrd (*C function*), 28
rocsolver_cpotf2 (*C function*), 65
rocsolver_cpotf2_batched (*C function*), 66
rocsolver_cpotf2_strided_batched (*C function*), 67
rocsolver_cpotrf (*C function*), 68
rocsolver_cpotrf_batched (*C function*), 69
rocsolver_cpotrf_strided_batched (*C function*), 70
rocsolver_create_handle (*C function*), 159
rocsolver_csteqr (*C function*), 63
rocsolver_cung2l (*C function*), 49
rocsolver_cung2r (*C function*), 46
rocsolver_cungbr (*C function*), 51
rocsolver_cungl2 (*C function*), 47
rocsolver_cunglq (*C function*), 48
rocsolver_cungql (*C function*), 50
rocsolver_cungqr (*C function*), 47
rocsolver_cungtr (*C function*), 52
rocsolver_cunm2l (*C function*), 57
rocsolver_cunm2r (*C function*), 53
rocsolver_cunmbr (*C function*), 60
rocsolver_cunml2 (*C function*), 55
rocsolver_cunmlq (*C function*), 56
rocsolver_cunmql (*C function*), 59
rocsolver_cunmqr (*C function*), 54
rocsolver_cunmtr (*C function*), 61
rocsolver_dbdsqr (*C function*), 26
rocsolver_destroy_handle (*C function*), 159
rocsolver_dgebd2 (*C function*), 106
rocsolver_dgebd2_batched (*C function*), 108
rocsolver_dgebd2_strided_batched (*C function*), 110
rocsolver_dgebrd (*C function*), 112
rocsolver_dgebrd_batched (*C function*), 113
rocsolver_dgebrd_strided_batched (*C function*), 115
rocsolver_dgelq2 (*C function*), 94
rocsolver_dgelq2_batched (*C function*), 95
rocsolver_dgelq2_strided_batched (*C function*), 96
rocsolver_dgelqf (*C function*), 97
rocsolver_dgelqf_batched (*C function*), 98
rocsolver_dgelqf_strided_batched (*C function*), 100
rocsolver_dgels (*C function*), 101
rocsolver_dgels_batched (*C function*), 103
rocsolver_dgels_strided_batched (*C function*), 104
rocsolver_dgeql2 (*C function*), 86
rocsolver_dgeql2_batched (*C function*), 87
rocsolver_dgeql2_strided_batched (*C function*), 88
rocsolver_dgeqlf (*C function*), 90
rocsolver_dgeqlf_batched (*C function*), 91
rocsolver_dgeqlf_strided_batched (*C function*), 92
rocsolver_dgeqr2 (*C function*), 79
rocsolver_dgeqr2_batched (*C function*), 80
rocsolver_dgeqr2_strided_batched (*C function*), 81
rocsolver_dgeqrf (*C function*), 82
rocsolver_dgeqrf_batched (*C function*), 83
rocsolver_dgeqrf_strided_batched (*C function*), 85
rocsolver_dgesvd (*C function*), 142
rocsolver_dgesvd_batched (*C function*), 144
rocsolver_dgesvd_strided_batched (*C function*), 147
rocsolver_dgetf2 (*C function*), 71
rocsolver_dgetf2_batched (*C function*), 72
rocsolver_dgetf2_npvt (*C function*), 150
rocsolver_dgetf2_npvt_batched (*C function*), 151
rocsolver_dgetf2_npvt_strided_batched (*C function*), 152
rocsolver_dgetf2_strided_batched (*C function*), 73
rocsolver_dgetrf (*C function*), 75
rocsolver_dgetrf_batched (*C function*), 76
rocsolver_dgetrf_npvt (*C function*), 154

rocsolver_dgetrf_npvt_batched (*C function*), 155
rocsolver_dgetrf_npvt_strided_batched (*C function*), 156
rocsolver_dgetrf_strided_batched (*C function*), 77
rocsolver_dgetri (*C function*), 134
rocsolver_dgetri_batched (*C function*), 135
rocsolver_dgetri_strided_batched (*C function*), 137
rocsolver_dgetrs (*C function*), 138
rocsolver_dgetrs_batched (*C function*), 139
rocsolver_dgetrs_strided_batched (*C function*), 140
rocsolver_diagonal (*C type*), 158
rocsolver_direction (*C type*), 158
rocsolver_dlabrd (*C function*), 24
rocsolver_dlarf (*C function*), 21
rocsolver_dlarfb (*C function*), 22
rocsolver_dlarfg (*C function*), 19
rocsolver_dlarft (*C function*), 20
rocsolver_dlaswp (*C function*), 18
rocsolver_dlatrd (*C function*), 28
rocsolver_dorg2l (*C function*), 32
rocsolver_dorg2r (*C function*), 29
rocsolver_dorgbr (*C function*), 34
rocsolver_dorgl2 (*C function*), 31
rocsolver_dorglq (*C function*), 32
rocsolver_dorgql (*C function*), 33
rocsolver_dorgqr (*C function*), 30
rocsolver_dorgtr (*C function*), 35
rocsolver_dorm2l (*C function*), 40
rocsolver_dorm2r (*C function*), 36
rocsolver_dormbr (*C function*), 43
rocsolver_dorml2 (*C function*), 38
rocsolver_dormlq (*C function*), 39
rocsolver_dormql (*C function*), 42
rocsolver_dormqr (*C function*), 37
rocsolver_dormtr (*C function*), 44
rocsolver_dpotf2 (*C function*), 65
rocsolver_dpotf2_batched (*C function*), 66
rocsolver_dpotf2_strided_batched (*C function*), 67
rocsolver_dpotrf (*C function*), 68
rocsolver_dpotrf_batched (*C function*), 69
rocsolver_dpotrf_strided_batched (*C function*), 70
rocsolver_dsteqr (*C function*), 63
rocsolver_dsterf (*C function*), 63
rocsolver_dsytd2 (*C function*), 117
rocsolver_dsytd2_batched (*C function*), 119
rocsolver_dsytd2_strided_batched (*C function*), 120
rocsolver_dsytrd (*C function*), 126
rocsolver_dsytrd_batched (*C function*), 127
rocsolver_dsytrd_strided_batched (*C function*), 129
rocsolver_fill (*C type*), 158
rocsolver_get_matrix (*C function*), 161
rocsolver_get_stream (*C function*), 160
rocsolver_get_vector (*C function*), 160
rocsolver_handle (*C type*), 157
rocsolver_int (*C type*), 157
rocsolver_operation (*C type*), 158
rocsolver_sbdsqr (*C function*), 26
rocsolver_set_matrix (*C function*), 160
rocsolver_set_stream (*C function*), 160
rocsolver_set_vector (*C function*), 160
rocsolver_sgebd2 (*C function*), 106
rocsolver_sgebd2_batched (*C function*), 108
rocsolver_sgebd2_strided_batched (*C function*), 110
rocsolver_sgebrd (*C function*), 112
rocsolver_sgebrd_batched (*C function*), 113
rocsolver_sgebrd_strided_batched (*C function*), 116
rocsolver_sgelq2 (*C function*), 94
rocsolver_sgelq2_batched (*C function*), 95
rocsolver_sgelq2_strided_batched (*C function*), 96
rocsolver_sgelqf (*C function*), 97
rocsolver_sgelqf_batched (*C function*), 98
rocsolver_sgelqf_strided_batched (*C function*), 100
rocsolver_sgels (*C function*), 101
rocsolver_sgels_batched (*C function*), 103
rocsolver_sgels_strided_batched (*C function*), 105
rocsolver_sgeql2 (*C function*), 86
rocsolver_sgeql2_batched (*C function*), 87
rocsolver_sgeql2_strided_batched (*C function*), 89
rocsolver_sgeqlf (*C function*), 90
rocsolver_sgeqlf_batched (*C function*), 91
rocsolver_sgeqlf_strided_batched (*C function*), 92
rocsolver_sgeqr2 (*C function*), 79
rocsolver_sgeqr2_batched (*C function*), 80
rocsolver_sgeqr2_strided_batched (*C function*), 81
rocsolver_sgeqrf (*C function*), 82
rocsolver_sgeqrf_batched (*C function*), 84
rocsolver_sgeqrf_strided_batched (*C function*), 85
rocsolver_sgesvd (*C function*), 142
rocsolver_sgesvd_batched (*C function*), 145
rocsolver_sgesvd_strided_batched (*C function*), 148
rocsolver_sgetf2 (*C function*), 71

rocsolver_sgetf2_batched (*C function*), 72
rocsolver_sgetf2_npvt (*C function*), 150
rocsolver_sgetf2_npvt_batched (*C function*), 151
rocsolver_sgetf2_npvt_strided_batched (*C function*), 152
rocsolver_sgetf2_strided_batched (*C function*), 73
rocsolver_sgetrf (*C function*), 75
rocsolver_sgetrf_batched (*C function*), 76
rocsolver_sgetrf_npvt (*C function*), 154
rocsolver_sgetrf_npvt_batched (*C function*), 155
rocsolver_sgetrf_npvt_strided_batched (*C function*), 156
rocsolver_sgetrf_strided_batched (*C function*), 77
rocsolver_sgetri (*C function*), 135
rocsolver_sgetri_batched (*C function*), 135
rocsolver_sgetri_strided_batched (*C function*), 137
rocsolver_sgetrs (*C function*), 138
rocsolver_sgetrs_batched (*C function*), 139
rocsolver_sgetrs_strided_batched (*C function*), 141
rocsolver_side (*C type*), 159
rocsolver_slabrd (*C function*), 24
rocsolver_slarf (*C function*), 21
rocsolver_slarfb (*C function*), 22
rocsolver_slarfg (*C function*), 19
rocsolver_slarft (*C function*), 20
rocsolver_slaswp (*C function*), 18
rocsolver_slatrd (*C function*), 28
rocsolver_sorg2l (*C function*), 32
rocsolver_sorg2r (*C function*), 29
rocsolver_sorgbr (*C function*), 34
rocsolver_sorgl2 (*C function*), 31
rocsolver_sorglq (*C function*), 32
rocsolver_sorgql (*C function*), 33
rocsolver_sorgqr (*C function*), 30
rocsolver_sorgtr (*C function*), 35
rocsolver_sorm2l (*C function*), 40
rocsolver_sorm2r (*C function*), 36
rocsolver_sormbr (*C function*), 43
rocsolver_sorml2 (*C function*), 38
rocsolver_sormlq (*C function*), 39
rocsolver_sormql (*C function*), 42
rocsolver_sormqr (*C function*), 37
rocsolver_sormtr (*C function*), 44
rocsolver_spotf2 (*C function*), 65
rocsolver_spotf2_batched (*C function*), 66
rocsolver_spotf2_strided_batched (*C function*), 67
rocsolver_spotrf (*C function*), 68
rocsolver_spotrf_batched (*C function*), 69
rocsolver_spotrf_strided_batched (*C function*), 70
rocsolver_ssteqr (*C function*), 63
rocsolver_ssterf (*C function*), 63
rocsolver_ssytd2 (*C function*), 117
rocsolver_ssytd2_batched (*C function*), 119
rocsolver_ssytd2_strided_batched (*C function*), 120
rocsolver_ssytrd (*C function*), 126
rocsolver_ssytrd_batched (*C function*), 127
rocsolver_ssytrd_strided_batched (*C function*), 129
rocsolver_status (*C type*), 159
rocsolver_storev (*C type*), 158
rocsolver_zbdsqr (*C function*), 26
rocsolver_zgebd2 (*C function*), 106
rocsolver_zgebd2_batched (*C function*), 108
rocsolver_zgebd2_strided_batched (*C function*), 110
rocsolver_zgebrd (*C function*), 112
rocsolver_zgebrd_batched (*C function*), 113
rocsolver_zgebrd_strided_batched (*C function*), 115
rocsolver_zgelq2 (*C function*), 94
rocsolver_zgelq2_batched (*C function*), 95
rocsolver_zgelq2_strided_batched (*C function*), 96
rocsolver_zgelqf (*C function*), 97
rocsolver_zgelqf_batched (*C function*), 98
rocsolver_zgelqf_strided_batched (*C function*), 100
rocsolver_zgels (*C function*), 101
rocsolver_zgels_batched (*C function*), 103
rocsolver_zgels_strided_batched (*C function*), 104
rocsolver_zgeql2 (*C function*), 86
rocsolver_zgeql2_batched (*C function*), 87
rocsolver_zgeql2_strided_batched (*C function*), 88
rocsolver_zgeqlf (*C function*), 90
rocsolver_zgeqlf_batched (*C function*), 91
rocsolver_zgeqlf_strided_batched (*C function*), 92
rocsolver_zgeqr2 (*C function*), 79
rocsolver_zgeqr2_batched (*C function*), 80
rocsolver_zgeqr2_strided_batched (*C function*), 81
rocsolver_zgeqrf (*C function*), 82
rocsolver_zgeqrf_batched (*C function*), 83
rocsolver_zgeqrf_strided_batched (*C function*), 85
rocsolver_zgesvd (*C function*), 142
rocsolver_zgesvd_batched (*C function*), 144

rocsolver_zgesvd_strided_batched (*C function*), 147
rocsolver_zgetf2 (*C function*), 71
rocsolver_zgetf2_batched (*C function*), 72
rocsolver_zgetf2_npvt (*C function*), 150
rocsolver_zgetf2_npvt_batched (*C function*), 151
rocsolver_zgetf2_npvt_strided_batched (*C function*), 152
rocsolver_zgetf2_strided_batched (*C function*), 73
rocsolver_zgetrf (*C function*), 75
rocsolver_zgetrf_batched (*C function*), 76
rocsolver_zgetrf_npvt (*C function*), 154
rocsolver_zgetrf_npvt_batched (*C function*), 155
rocsolver_zgetrf_npvt_strided_batched (*C function*), 156
rocsolver_zgetrf_strided_batched (*C function*), 77
rocsolver_zgetri (*C function*), 134
rocsolver_zgetri_batched (*C function*), 135
rocsolver_zgetri_strided_batched (*C function*), 136
rocsolver_zgetrs (*C function*), 138
rocsolver_zgetrs_batched (*C function*), 139
rocsolver_zgetrs_strided_batched (*C function*), 140
rocsolver_zhetd2 (*C function*), 122
rocsolver_zhetd2_batched (*C function*), 123
rocsolver_zhetd2_strided_batched (*C function*), 124
rocsolver_zhetrd (*C function*), 130
rocsolver_zhetrd_batched (*C function*), 131
rocsolver_zhetrd_strided_batched (*C function*), 133
rocsolver_zlabrd (*C function*), 24
rocsolver_zlacgv (*C function*), 17
rocsolver_zlarf (*C function*), 21
rocsolver_zlarfb (*C function*), 22
rocsolver_zlarfg (*C function*), 19
rocsolver_zlarft (*C function*), 20
rocsolver_zlaswp (*C function*), 18
rocsolver_zlatrd (*C function*), 28
rocsolver_zpotf2 (*C function*), 65
rocsolver_zpotf2_batched (*C function*), 66
rocsolver_zpotf2_strided_batched (*C function*), 67
rocsolver_zpotrf (*C function*), 68
rocsolver_zpotrf_batched (*C function*), 69
rocsolver_zpotrf_strided_batched (*C function*), 70
rocsolver_zsteqr (*C function*), 63
rocsolver_zung2l (*C function*), 49
rocsolver_zung2r (*C function*), 46
rocsolver_zungbr (*C function*), 51
rocsolver_zungl2 (*C function*), 47
rocsolver_zunglq (*C function*), 48
rocsolver_zungql (*C function*), 50
rocsolver_zungqr (*C function*), 47
rocsolver_zungtr (*C function*), 52
rocsolver_zunm2l (*C function*), 57
rocsolver_zunm2r (*C function*), 53
rocsolver_zunmbr (*C function*), 60
rocsolver_zunml2 (*C function*), 55
rocsolver_zunmlq (*C function*), 56
rocsolver_zunmql (*C function*), 59
rocsolver_zunmqr (*C function*), 54
rocsolver_zunmtr (*C function*), 61