Matrix computations on the GPU

CUBLAS and MAGMA by example

Andrzej Chrzęszczyk
Jan Kochanowski University, Kielce, Poland
Jakub Chrzęszczyk
National Computational Infrastructure
Australian National University, Canberra, Australia

August, 2013

Foreword

Many scientific computer applications need high-performance matrix algebra. The major hardware developments always influenced new developments in linear algebra libraries. For example in the 80's the cache-based machines appeared and LAPACK based on Level 3 BLAS was developed. In the 90's new parallel platforms influenced ScaLAPACK developments.

To fully exploit the power of current heterogeneous systems of multi/many core CPUs and GPUs (Graphics Processing Units) new tools are needed.
The main purpose of this document is to present two of them, CUBLAS and MAGMA linear algebra C/C++ libraries.
We propose a practical, hands-on approach. We show how to install and use these libraries. The detailed table of contents allows for easy navigation through over 100 code samples. We believe that the presented document can be an useful addition to the existing documentation for CUBLAS and MAGMA.

Contents

Chapter 1

CUDA installation

1.1 Installing CUDA environment

Both CUBLAS and MAGMA need CUDA (Compute Unified Device Architecture) environment. In fact CUBLAS is a part of CUDA. In this chapter we will show how to install CUDA 5.5 on Redhat 6.3. At the time of writting this text the download website was https://developer.nvidia.com/cuda-downloads. On the site one can find http://developer.download.nvidia.com/compute/cuda/5_5/rel/docs/CUDA_Getting_Started_Linux.pdf which is a good starting point.
Before getting started, we are required to remove nouveau drivers from the system (if they are enabled). Usually if the system installer detects Nvidia cards, it installs nouveau drivers. If nouveau is enabled it is impossible to install proprietary Nvidia drivers which are necessary for CUDA Toolkit. To disable nouveau it suffices to perform (as root) the following three steps.

```
# echo -e "\nblacklist nouveau" >> /etc/modprobe.d/blacklist.conf
# dracut /boot/initramfs-$(uname -r).img $(uname -r)
# reboot
```

To compile CUDA we shall also need some additional packages

```
# yum install gcc-c++ make
# yum install kernel-devel
# yum install freeglut-devel libXi-devel libXmu-devel
# yum install openmpi-devel
```

Since in Redhat OpenMPI is installed in `/usr/lib64/openmpi` the following exports will be needed in compilation of CUDA samples

```
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/lib64/openmpi/lib
$ export OPENMPI_HOME=/usr/lib64/openmpi
$ export PATH=/usr/lib64/openmpi/bin:$PATH
```

As we have mentioned the CUDA Toolkit can be downloaded from https://developer.nvidia.com/cuda-downloads

```
#In the case of Redhat 6.x
$ wget http://developer.download.nvidia.com/compute/cuda/5_5/
rel/installers/cuda_5.5.22_linux_64.run
```

The graphical display manager must not be running during the CUDA video driver installation. Hence we have to logout the desktop, switch into console mode and stop the graphical display manager. Using the text console we enter the directory with `cuda_5.5.22_linux_64.run` downloaded. The default installation directory is
`/usr/local/cuda-5.5` so `root` privileges are needed.

```
#sh cuda_5.5.22_linux_64.run
```

It is important to set properly all the paths mentioned by the installer:

```
Install NVIDIA Accelerated Graphics Driver for Linux-x86_64
319.37? ((y)es/(n)o/(q)uit): yes
Install the CUDA 5.5 Toolkit? ((y)es/(n)o/(q)uit): yes
Enter Toolkit Location [ default is /usr/local/cuda-5.5 ]:
Install the CUDA 5.5 Samples? ((y)es/(n)o/(q)uit): yes
Enter CUDA Samples Location
[ default is /root/NVIDIA_CUDA-5.5_Samples ]:
Installing the NVIDIA display driver...
Installing the CUDA Toolkit in /usr/local/cuda-5.5 ...
Installing the CUDA Samples in /root/NVIDIA_CUDA-5.5_Samples ...
Copying samples to
/root/NVIDIA_CUDA-5.5_Samples/NVIDIA_CUDA-5.5_Samples now...
Finished copying samples.

===========
= Summary =
===========

Driver:   Installed
Toolkit:  Installed in /usr/local/cuda-5.5
Samples:  Installed in /root/NVIDIA_CUDA-5.5_Samples

* Please make sure your PATH includes /usr/local/cuda-5.5/bin

* Please make sure your LD_LIBRARY_PATH
*   for 32-bit Linux distributions includes
    /usr/local/cuda-5.5/lib
```

```
*   for 64-bit Linux distributions includes
    /usr/local/cuda-5.5/lib64:/lib
* OR
*   for 32-bit Linux distributions add /usr/local/cuda-5.5/lib
*   for 64-bit Linux distributions add /usr/local/cuda-5.5/lib64
* and /lib
* to /etc/ld.so.conf and run ldconfig as root

* To uninstall CUDA, remove the CUDA files in /usr/local/cuda-5.5
* Installation Complete

Please see CUDA_Getting_Started_Linux.pdf in
/usr/local/cuda-5.5/doc/pdf
for detailed information on setting up CUDA.

Logfile is /tmp/cuda_install_30834.log
```

For users working in text mode it is important to read point 6 from CUDA_Getting_Started_Guide_For_Linux . The script from the guide

```
#!/bin/bash
/sbin/modprobe nvidia
if [ "$?" -eq 0 ]; then
  # Count the number of NVIDIA controllers found.
  NVDEVS=‘lspci | grep -i NVIDIA‘
  N3D=‘echo "$NVDEVS" | grep "3D controller" | wc -l‘
  NVGA=‘echo "$NVDEVS" | grep "VGA compatible controller" | wc -l‘
  N=‘expr $N3D + $NVGA - 1‘
  for i in ‘seq 0 $N‘; do
    mknod -m 666 /dev/nvidia$i c 195 $i
  done
  mknod -m 666 /dev/nvidiactl c 195 255
else
  exit 1
fi
```

should be copied for example to `/etc/rc.local`. The script loads the driver kernel module and creates the entries in device files `/dev/nvidia*` (this is performed automatically if a GUI environment is initialized).
After adding to `$HOME/.bashrc` the entries

```
export OPENMPI_HOME=/usr/lib64/openmpi
export CUDA_HOME=/usr/local/cuda-5.5
export LD_LIBRARY_PATH=${CUDA_HOME}/lib64
PATH=${CUDA_HOME}/bin:${OPENMPI_HOME}/bin:${PATH}
```

```
export PATH
```

and

```
# reboot
```

one can copy `samples` to `$HOME` directory and make the examples:

```
$ cp -r /usr/local/cuda-5.5/samples ~
$ cd ~/samples
$ make
```

If the CUDA software is installed and configured correctly, the executable:

```
$ ~/samples/1_Utilities/deviceQuery/deviceQuery
```

should display the properties of the detected CUDA devices.
The `nbody` executable:

```
$ ~/samples/5_Simulations/nbody/nbody -benchmark -numdevices=2
# (in the case of two devices)
```

gives the opportunity to check GPU performance.
In Tesla cards one can check the state of devices using

```
$ nvidia-smi            # man nvidia-smi
```

Chapter 2

Measuring GPUs performance

2.1 Linpack benchmark for CUDA

Registered developers can download from https://developer.nvidia.com/ the version of Linpack benchmark prepared specially for CUDA. In August, 2013 the current version for Tesla cards was `hpl-2.0_FERMI_v15.tgz`. After uncompressing one obtains the directory `hpl-2.0_FERMI_v15`. We enter the directory

```
$ cd hpl-2.0_FERMI_v15
```

The file `INSTALL` contains installation instructions. The example file `Make.CUDA` should be edited. In our system we have edited (only) the following lines:

```
TOPdir = $HOME/hpl-2.0_FERMI_v15
MPdir = /usr/lib64/openmpi              # Redhat/Centos default
MPinc = -I/usr/include/openmpi-x86_64  #  for OpenMPI
MPlib = -L/usr/lib64/openmpi/lib
LAdir = /opt/intel/mkl/lib/intel64     # MKL presence assumed !!!
LAinc = -I/opt/intel/mkl/include
LAlib = -L$(TOPdir)/src/cuda  -ldgemm -L/usr/local/cuda/lib64
 -lcuda -lcudart -lcublas -L$(LAdir) -lmkl_intel_lp64
 -lmkl_intel_thread -lmkl_core -liomp5
```

After entering the directory we can do the compilation

```
$ make
```

which creates in `hpl-2.0_FERMI_v15/bin/CUDA` a new executable `xhpl`. We can enter the directory

```
$ cd bin/CUDA
```

and edit two files `run_linpack` and `HPL.dat`. For example in `run_linpack` script file we edited (only) the two lines

```
HPL_DIR=$HOME/hpl-2.0_FERMI_v15
CPU_CORES_PER_GPU=8
```

(two eight core CPUs + two S2050 GPUs in each of two nodes). The file `HPL.dat` contains the description of the problem to be solved. Linpack solves dense `NxN` systems of linear equations in double precision. Users can specify in `HPL.dat` the number of problems, their sizes and some other parameters. The detailed description of this file can be found in `hpl-2.0_FERMI_v15/TUNING`.
For our benchmarks we have edited the sample `HPL.dat` file:

```
HPLinpack benchmark input file
Innovative Computing Laboratory, University of Tennessee
HPL.out      output file name (if any)
6            device out (6=stdout,7=stderr,file)
1            # of problems sizes (N)
100000       Ns
1            # of NBs
768          NBs
0            PMAP process mapping (0=Row-,1=Column-major)
1            # of process grids (P x Q)
2            Ps
2            Qs
16.0         threshold
1            # of panel fact
0 1 2        PFACTs (0=left, 1=Crout, 2=Right)
1            # of recursive stopping criterium
2 8          NBMINs (>= 1)
1            # of panels in recursion
2            NDIVs
1            # of recursive panel fact.
0 1 2        RFACTs (0=left, 1=Crout, 2=Right)
1            # of broadcast
0 2          BCASTs (0=1rg,1=1rM,2=2rg,3=2rM,4=Lng,5=LnM)
1            # of lookahead depth
1 0          DEPTHs (>=0)
1            SWAP (0=bin-exch,1=long,2=mix)
192          swapping threshold
1            L1 in (0=transposed,1=no-transposed) form
1            U  in (0=transposed,1=no-transposed) form
1            Equilibration (0=no,1=yes)
8            memory alignment in double (> 0)
```

Let us comment the first ten lines of this file (beginning from `HPL.out`). The remaining lines were unchanged.

1. `HPL.out` is used as output file if the number in the next line is not equal to 6 or 7.
2. Number `6` means that the output goes to `stdout`. If it is replaced by `5` (for example) then the output goes to `HPL.out`
3. The number `1` in the third line means that we want to solve exactly one system.
4. The number `100000` denotes the size of the system. Large systems can give better performance but need more memory.
5. The number `1` in the fifth line means that we shall try only one data-block size
6. The number `768` denotes the block size. The number should be a multiple of `128`. It can be selected experimentally.
7. `0` in the next line denotes row-major process mapping (not changed in sample `HPL.dat` file).
8. Next `1` denotes the number of grids used (in our example only one). Testing four cards, since we have two nodes with two GPUs in each, we choose one `PxQ=2x2` grid. `PxQ` should be equal to the total number of tested GPUs.
9. The number `2` means that the first dimension of the grid `P=2`.
10. The number `2` in the next line means that the second dimension of the grid `Q=2`.

2.2 Tests results

At our disposal we had two nodes with Redhat 6.3, CUDA 5.5 and with the following hardware :

- two socket Xeon CPU E5-2650, 2.00GHz,
- two Tesla S2050 GPUs,
- 256 GB RAM,
- Gigabit Ethernet.

2.2.1 One Tesla S2050 GPU (428.9 GFlop/s)

For one GPU we have used `P=1, Q=1` parameters in `HPL.dat` and have obtained the following results.

```
$ mpirun -np 1 ./run_linpack

================================================================================
T/V                N    NB     P     Q               Time     Gflops
--------------------------------------------------------------------------------
WR10L2L2      100000   768     1     1            1554.29     4.289e+02
--------------------------------------------------------------------------------
||Ax-b||_oo/(eps*(||A||_oo*||x||_oo+||b||_oo)*N)=0.0039050 ...PASSED
================================================================================
```

2.2.2 Two Tesla S2050 GPUs (679.0 GFlop/s)

For two GPUs we have used `P=1, Q=2` parameters in `HPL.dat` and have obtained the following results.

```
$ mpirun -np 2 ./run_linpack

================================================================================
T/V                N    NB     P     Q               Time     Gflops
--------------------------------------------------------------------------------
WR10L2L2      100000   768     1     2             981.87    6.790e+02
--------------------------------------------------------------------------------
||Ax-b||_oo/(eps*(||A||_oo*||x||_oo+||b||_oo)*N)=0.0035832 ...PASSED
================================================================================
```

Remark. For two CPUs, using the CPU Linpack we have obtained `273.8 GFlop/s` for `N=100000`.

2.2.3 Four Tesla S2050 GPUs (1363 GFlop/s)

For four GPUs we have used `P=2, Q=2` parameters in `HPL.dat` and have obtained the following results.

```
# For N=100000
$ mpirun -np 4  -host node1,node2 ./run_linpack

================================================================================
T/V                N    NB     P     Q               Time     Gflops
--------------------------------------------------------------------------------
WR10L2L2      100000   768     2     2             561.98    1.186e+03
--------------------------------------------------------------------------------
||Ax-b||_oo/(eps*(||A||_oo*||x||_oo+||b||_oo)*N)=0.0037021 ...PASSED
```

```
================================================================================

# For N=200000
$ mpirun -np 4  -host node1,node2 ./run_linpack

================================================================================
T/V                N    NB     P     Q               Time     Gflops
--------------------------------------------------------------------------------
WR10L2L2      200000   1024     2     2            3912.98    1.363e+03
--------------------------------------------------------------------------------
||Ax-b||_oo/(eps*(||A||_oo*||x||_oo+||b||_oo)*N)=0.0038225 ...PASSED
================================================================================
```

Remark. Setting the number of solved systems to 20 and their size to 200000 we have checked that the system is able to work with the 1300 GFlop/s performance over 30 hours.

2.2.4 Two Tesla K20m GPUs (1789 GFlop/s)

For two Kepler GPUs and two socket Xeon CPUs E5-2665 we have used `P=1, Q=2` parameters in `HPL.dat` and have obtained the following results.

```
$ mpirun -np 2 ./run_linpack

================================================================================
T/V                N    NB     P     Q               Time     Gflops
--------------------------------------------------------------------------------
WR10L2L2      100000   768     1     2             372.74    1.789e+03
--------------------------------------------------------------------------------
||Ax-b||_oo/(eps*(||A||_oo*||x||_oo+||b||_oo)*N)=0.0030869 ...PASSED
================================================================================
```

Remark. For two E5-2665 CPUs, using the CPU Linpack we have obtained `307.16 GFlop/s` for `N=100000`.

Chapter 3

CUBLAS by example

3.1 General remarks on the examples

CUBLAS is an abbreviation for CUDA Basic Linear Algebra Subprograms. In the file `/usr/local/cuda-5.5/doc/pdf/CUBLAS_Library.pdf` one can find a detailed description of the CUBLAS library syntax and we shall avoid to repeat the information contained there. Instead we present a series of examples how to use the library.
All subprograms have four versions corresponding to four data types

- `s,S - float` – real single-precision
- `d,D - double` – real double-precision,
- `c,C - cuComplex` – complex single-precision,
- `z,Z - cuDoubleComplex` –complex double-precision.

For example `cublasI<t>amax` is a template which can represent `cublasIsamax, cublasIdamax, cublasIcamax` or `cublasIzamax`.

- We shall restrict our examples in this chapter to single precision versions. The reason is that low-end devices have restricted double precision capabilities. On the other hand the changes needed in the double precision case are not significant. In most examples we use real data but the complex cases are also considered (see the subsections with the title of the form `cublasC*`).
- CUBLAS Library User Guide contains an example showing how to check for errors returned by API calls. Ideally we should check for errors on every API call. Unfortunately such an approach doubles the length of our sample codes (which are as short as possible by design). Since our set of CUBLAS sample code (without error checking) is 80 pages long we have decided to ignore the error checking and to focus on the explanations which cannot be found in User Guide. The reader

can add the error checking code from CUBLAS Library User Guide example with minor modifications.

- To obtain more compact explanations in our examples we restrict the full generality of CUBLAS to the special case where the leading dimension of matrices is equal to the number of rows and the stride between consecutive elements of vectors is equal to 1. CUBLAS allows for more flexible approach giving the user the access to submatrices an subvectors. The corresponding explanations can be found in CUBLAS Library User Guide and in BLAS manual.

3.2 CUBLAS Level-1. Scalar and vector based operations

3.2.1 `cublasIsamax, cublasIsamin` - maximal, minimal elements

This function finds the smallest index of the element of an array with the maximum /minimum magnitude.

```
//nvcc 001isamax.c -lcublas
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include "cublas_v2.h"
#define n 6                                        // length of x
int main(void){
  cudaError_t cudaStat;                        // cudaMalloc status
  cublasStatus_t stat;                  // CUBLAS functions status
  cublasHandle_t handle;                          // CUBLAS context
  int j;                                       // index of elements
  float* x;                                 // n-vector on the host
  x=(float *)malloc (n*sizeof(*x));            // host memory alloc
  for(j=0;j<n;j++)
    x[j]=(float)j;                               // x={0,1,2,3,4,5}
  printf("x: ");
  for(j=0;j<n;j++)
    printf("%4.0f,",x[j]);                                // print x
  printf("\n");
// on the device
  float* d_x;                               // d_x - x on the device
  cudaStat=cudaMalloc((void**)&d_x,n*sizeof(*x));    // device
                                             // memory alloc for x
  stat = cublasCreate(&handle);  // initialize CUBLAS context
  stat = cublasSetVector(n,sizeof(*x),x,1,d_x,1);//cp x ->d_x
  int result;          // index of the maximal/minimal element
// find the smallest index of the element of d_x with maximum
// absolute value

  stat=cublasIsamax(handle,n,d_x,1,&result);
```

```
  printf("max |x[i]|:%4.0f\n",fabs(x[result-1]));    // print
                                  // max{|x[0]|,...,|x[n-1]|}
// find the smallest index of the element of d_x with minimum
// absolute value

  stat=cublasIsamin(handle,n,d_x,1,&result);

  printf("min |x[i]|:%4.0f\n",fabs(x[result-1]));    // print
                                  // min{|x[0]|,...,|x[n-1]|}
  cudaFree(d_x);                        // free device memory
  cublasDestroy(handle);              // destroy CUBLAS context
  free(x);                                // free host memory
  return EXIT_SUCCESS;
}
// x:  0, 1, 2, 3, 4, 5,
// max |x[i]|:   5
// min |x[i]|:   0
```

3.2.2 cublasSasum - sum of absolute values

This function computes the sum of the absolute values of the elements of an array.

```
//nvcc 003sasumVec.c -lcublas
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include "cublas_v2.h"
#define n 6                                       // length of x
int main(void){
  cudaError_t cudaStat;                       // cudaMalloc status
  cublasStatus_t stat;                 // CUBLAS functions status
  cublasHandle_t handle;                        // CUBLAS context
  int j;                                      // index of elements
  float* x;                                 // n-vector on the host
  x=(float *)malloc (n*sizeof(*x));           // host memory alloc
  for(j=0;j<n;j++)
    x[j]=(float)j;                              // x={0,1,2,3,4,5}
  printf("x: ");
  for(j=0;j<n;j++)
    printf("%2.0f,",x[j]);                              // print x
  printf("\n");
// on the device
  float* d_x;                              // d_x - x on the device
  cudaStat=cudaMalloc((void**)&d_x,n*sizeof(*x));    //device
                                                    // memory alloc
  stat = cublasCreate(&handle);  // initialize CUBLAS context
  stat = cublasSetVector(n,sizeof(*x),x,1,d_x,1);// cp x->d_x
  float result;
// add absolute values of elements of the array d_x:
```

```
                                        // |d_x[0]|+...+|d_x[n-1]|

  stat=cublasSasum(handle,n,d_x,1,&result);

//print the result
  printf("sum of the absolute values of elements of x:%4.0f\n",
                                                          result);
  cudaFree(d_x);                            // free device memory
  cublasDestroy(handle);                 // destroy CUBLAS context
  free(x);                                    // free host memory
  return EXIT_SUCCESS;
}
// x:  0, 1, 2, 3, 4, 5,
// sum of the absolute values of elements of x:  15
                                      //|0|+|1|+|2|+|3|+|4|+|5|=15
```

3.2.3 cublasSaxpy - compute $\alpha x + y$

This function multiplies the vector x by the scalar α and adds it to the vector y

$$y = \alpha\, x + y.$$

```
//nvcc 004saxpy.c -lcublas
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include "cublas_v2.h"
#define n 6                                       // length of x,y
int main(void){
  cudaError_t cudaStat;                         // cudaMalloc status
  cublasStatus_t stat;                   // CUBLAS functions status
  cublasHandle_t handle;                          // CUBLAS context
  int j;                                       // index of elements
  float* x;                                 // n-vector on the host
  float* y;                                 // n-vector on the host
  x=(float *)malloc (n*sizeof(*x));// host memory alloc for x
  for(j=0;j<n;j++)
    x[j]=(float)j;                                // x={0,1,2,3,4,5}
  y=(float *)malloc (n*sizeof(*y));// host memory alloc for y
  for(j=0;j<n;j++)
    y[j]=(float)j;                                // y={0,1,2,3,4,5}
  printf("x,y:\n");
  for(j=0;j<n;j++)
    printf("%2.0f,",x[j]);                              // print x,y
  printf("\n");
// on the device
  float* d_x;                             // d_x - x on the device
  float* d_y;                             // d_y - y on the device
```

```
  cudaStat=cudaMalloc((void**)&d_x,n*sizeof(*x));    //device
                                          // memory alloc for x
  cudaStat=cudaMalloc((void**)&d_y,n*sizeof(*y));    //device
                                          // memory alloc for y
  stat = cublasCreate(&handle);  // initialize CUBLAS context
  stat = cublasSetVector(n,sizeof(*x),x,1,d_x,1); //cp x->d_x
  stat = cublasSetVector(n,sizeof(*y),y,1,d_y,1); //cp y->d_y
  float al=2.0;                                        // al=2
// multiply the vector d_x by the scalar al and add to  d_y
// d_y = al*d_x + d_y,   d_x,d_y - n-vectors; al - scalar

  stat=cublasSaxpy(handle,n,&al,d_x,1,d_y,1);

  stat=cublasGetVector(n,sizeof(float),d_y,1,y,1);//cp d_y->y
  printf("y after Saxpy:\n");          // print y after Saxpy
  for(j=0;j<n;j++)
    printf("%2.0f,",y[j]);
  printf("\n");
  cudaFree(d_x);                          // free device memory
  cudaFree(d_y);                          // free device memory
  cublasDestroy(handle);             // destroy CUBLAS context
  free(x);                                  // free host memory
  free(y);                                  // free host memory
  return EXIT_SUCCESS;
}
// x,y:
// 0, 1, 2, 3, 4, 5,

// y after Saxpy:
// 0, 3, 6, 9,12,15,// 2*x+y = 2*{0,1,2,3,4,5} + {0,1,2,3,4,5}
```

3.2.4 cublasScopy - copy vector into vector

This function copies the vector x into the vector y.

```
//nvcc 005scopy.c -lcublas
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include "cublas_v2.h"
#define n 6                                    // length of x,y
int main(void){
  cudaError_t cudaStat;                     // cudaMalloc status
  cublasStatus_t stat;                // CUBLAS functions status
  cublasHandle_t handle;                      // CUBLAS context
  int j;                                    // index of elements
  float* x;                               // n-vector on the host
  float* y;                               // n-vector on the host
  x=(float *)malloc (n*sizeof(*x));// host memory alloc for x
  for(j=0;j<n;j++)
    x[j]=(float)j;                          // x={0,1,2,3,4,5}
```

```
  printf("x: ");
  for(j=0;j<n;j++)
    printf("%2.0f,",x[j]);                          // print x
  printf("\n");
  y=(float *)malloc (n*sizeof(*y));// host memory alloc for y
// on the device
  float* d_x;                            // d_x - x on the device
  float* d_y;                            // d_y - y on the device
  cudaStat=cudaMalloc((void**)&d_x,n*sizeof(*x));   // device
                                            // memory alloc for x
  cudaStat=cudaMalloc((void**)&d_y,n*sizeof(*y));   // device
                                            // memory alloc for y
  stat = cublasCreate(&handle);  // initialize CUBLAS context
  stat = cublasSetVector(n,sizeof(*x),x,1,d_x,1); //cp x->d_x
// copy the vector d_x into d_y:  d_x -> d_y

  stat=cublasScopy(handle,n,d_x,1,d_y,1);

  stat=cublasGetVector(n,sizeof(float),d_y,1,y,1);//cp d_y->y
  printf("y after copy:\n");
  for(j=0;j<n;j++)
    printf("%2.0f,",y[j]);                          // print y
  printf("\n");
  cudaFree(d_x);                            // free device memory
  cudaFree(d_y);                            // free device memory
  cublasDestroy(handle);                 // destroy CUBLAS context
  free(x);                                    // free host memory
  free(y);                                    // free host memory
  return EXIT_SUCCESS;
}
// x:  0, 1, 2, 3, 4, 5,

// y after Scopy:              // {0,1,2,3,4,5} -> {0,1,2,3,4,5}
// 0, 1, 2, 3, 4, 5,
```

3.2.5 `cublasSdot` - dot product

This function computes the dot product of vectors x and y

$$x.y = x_0 y_0 + \ldots + x_{n-1} y_{n-1},$$

for real vectors x, y and

$$x.y = x_0 \bar{y}_0 + \ldots + x_{n-1} \bar{y}_{n-1},$$

for complex x, y.

```
//nvcc 006sdot.c -lcublas
#include <stdio.h>
#include <stdlib.h>
```

```
#include <cuda_runtime.h>
#include "cublas_v2.h"
#define n 6                                     // length of x,y
int main(void){
  cudaError_t cudaStat;                        // cudaMalloc status
  cublasStatus_t stat;                  // CUBLAS functions status
  cublasHandle_t handle;
  int j;                                       // index of elements
  float* x;                                 // n-vector on the host
  float* y;                                 // n-vector on the host
  x=(float *)malloc (n*sizeof(*x));// host memory alloc for x
  for(j=0;j<n;j++)
    x[j]=(float)j;                                // x={0,1,2,3,4,5}
  y=(float *)malloc (n*sizeof(*y));// host memory alloc for y
  for(j=0;j<n;j++)
    y[j]=(float)j;                                // y={0,1,2,3,4,5}
  printf("x,y:\n");
  for(j=0;j<n;j++)
    printf("%2.0f,",x[j]);                            // print x,y
  printf("\n");
// on the device
  float* d_x;                              // d_x - x on the device
  float* d_y;                              // d_y - y on the device
  cudaStat=cudaMalloc((void**)&d_x,n*sizeof(*x));    //device
                                           // memory alloc for x
  cudaStat=cudaMalloc((void**)&d_y,n*sizeof(*y));    //device
                                           // memory alloc for y
  stat = cublasCreate(&handle);  // initialize CUBLAS context
  stat = cublasSetVector(n,sizeof(*x),x,1,d_x,1);// cp x->d_x
  stat = cublasSetVector(n,sizeof(*y),y,1,d_y,1);// cp y->d_y
  float result;
// dot product of two vectors d_x,d_y:
// d_x[0]*d_y[0]+...+d_x[n-1]*d_y[n-1]

  stat=cublasSdot(handle,n,d_x,1,d_y,1,&result);

  printf("dot product x.y:\n");
  printf("%7.0f\n",result);                 // print the result
  cudaFree(d_x);                           // free device memory
  cudaFree(d_y);                           // free device memory
  cublasDestroy(handle);                // destroy CUBLAS context
  free(x);                                    // free host memory
  free(y);                                    // free host memory
return EXIT_SUCCESS;
}
// x,y:
// 0, 1, 2, 3, 4, 5,

// dot product x.y:                         // x.y=
//     55                                   // 1*1+2*2+3*3+4*4+5*5
```

3.2.6 `cublasSnrm2` - Euclidean norm

This function computes the Euclidean norm of the vector x

$$\|x\| = \sqrt{|x_0|^2 + \ldots + |x_{n-1}|^2},$$

where $x = \{x_0, \ldots, x_{n-1}\}$.

```
//nvcc 007snrm2.c -lcublas
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include "cublas_v2.h"
#define n 6                                          // length of x
int main(void){
  cudaError_t cudaStat;                          // cudaMalloc status
  cublasStatus_t stat;                 // CUBLAS functions status
  cublasHandle_t handle;                          // CUBLAS context
  int j;                                       // index of elements
  float* x;                                 // n-vector on the host
  x=(float *)malloc (n*sizeof(*x));// host memory alloc for x
  for(j=0;j<n;j++)
    x[j]=(float)j;                              // x={0,1,2,3,4,5}
  printf("x: ");
  for(j=0;j<n;j++)
    printf("%2.0f,",x[j]);                             // print x
  printf("\n");
// on the device
  float* d_x;                               // d_x - x on the device
  cudaStat=cudaMalloc((void**)&d_x,n*sizeof(*x));   // device
                                           // memory alloc for x
  stat = cublasCreate(&handle);  // initialize CUBLAS context
  stat = cublasSetVector(n,sizeof(*x),x,1,d_x,1);// cp x->d_x
  float result;
// Euclidean norm of the vector d_x:
// \sqrt{d_x[0]^2+...+d_x[n-1]^2}

  stat=cublasSnrm2(handle,n,d_x,1,&result);

  printf("Euclidean norm of  x: ");
  printf("%7.3f\n",result);                // print the result
  cudaFree(d_x);                          // free device memory
  cublasDestroy(handle);             // destroy CUBLAS context
  free(x);                                   // free host memory
  return EXIT_SUCCESS;
}
// x:  0, 1, 2, 3, 4, 5,
                               // ||x||=
//Euclidean norm of x: 7.416 //\sqrt{0^2+1^2+2^2+3^2+4^2+5^2}
```

3.2.7 `cublasSrot` - apply the Givens rotation

This function multiplies 2×2 Givens rotation matrix $\begin{pmatrix} c & s \\ -s & c \end{pmatrix}$ with the $2 \times n$ matrix $\begin{pmatrix} x_0 & \dots & x_{n-1} \\ y_0 & \dots & y_{n-1} \end{pmatrix}$.

```
// nvcc 008srot.c -lcublas
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include "cublas_v2.h"
#define n 6                                     // length of x,y
int main(void){
  cudaError_t cudaStat;                       // cudaMalloc status
  cublasStatus_t stat;                  // CUBLAS functions status
  cublasHandle_t handle;                        // CUBLAS context
  int j;                                      // index of elements
  float* x;                                 // n-vector on the host
  float* y;                                 // n-vector on the host
  x=(float *)malloc (n*sizeof(*x));// host memory alloc for x
  for(j=0;j<n;j++)
    x[j]=(float)j;                                // x={0,1,2,3,4,5}
  y=(float *)malloc (n*sizeof(*y));// host memory alloc for y
  for(j=0;j<n;j++)
    y[j]=(float)j*j;                            // y={0,1,4,9,16,25}
  printf("x: ");
  for(j=0;j<n;j++)
    printf("%7.0f,",x[j]);                                // print x
  printf("\n");
  printf("y: ");
  for(j=0;j<n;j++)
    printf("%7.0f,",y[j]);                                // print y
  printf("\n");
// on the device
  float* d_x;                             // d_x - x on the device
  float* d_y;                             // d_y - y on the device
  cudaStat=cudaMalloc((void**)&d_x,n*sizeof(*x));    //device
                                            // memory alloc for x
  cudaStat=cudaMalloc((void**)&d_y,n*sizeof(*y));    //device
                                            // memory alloc for y
  stat = cublasCreate(&handle);  // initialize CUBLAS context
  stat = cublasSetVector(n,sizeof(*x),x,1,d_x,1); //cp x->d_x
  stat = cublasSetVector(n,sizeof(*y),y,1,d_y,1); //cp y->d_y
  float c=0.5;
  float s=0.8669254;                            // s=sqrt(3.0)/2.0
// Givens rotation
//                        [ c s ]                    [ row(x) ]
//multiplies  2x2 matrix  [     ]  with 2xn matrix   [        ]
//                        [-s c ]                    [ row(y) ]
//
```

```
//  [1/2          sqrt(3)/2]     [0,1,2,3, 4, 5]
//  [-sqrt(3)/2       1/2  ]     [0,1,4,9,16,25]

  stat=cublasSrot(handle,n,d_x,1,d_y,1,&c,&s);

  stat=cublasGetVector(n,sizeof(float),d_x,1,x,1);//cp d_x->x

  printf("x after Srot:\n");              // print x after Srot
  for(j=0;j<n;j++)
    printf("%7.3f,",x[j]);
  printf("\n");
  stat=cublasGetVector(n,sizeof(float),d_y,1,y,1);//cp d_y->y
  printf("y after Srot:\n");              // print y after Srot
  for(j=0;j<n;j++)
    printf("%7.3f,",y[j]);
  printf("\n");
  cudaFree(d_x);                          // free device memory
  cudaFree(d_y);                          // free device memory
  cublasDestroy(handle);               // destroy CUBLAS context
  free(x);                                  // free host memory
  free(y);                                  // free host memory
  return EXIT_SUCCESS;
}
// x:     0,     1,     2,     3,     4,     5,
// y:     0,     1,     4,     9,    16,    25,

// x after Srot:
//   0.000,   1.367,   4.468,   9.302,  15.871,  24.173,
// y after Srot:
//   0.000,  -0.367,   0.266,   1.899,   4.532,   8.165,
//                      // [x]   [ 0.5    0.867] [0 1 2 3   4   5]
//                      // [ ]= [              ]*[              ]
//                      // [y]   [-0.867   0.5 ] [0 1 4 9 16 25]
```

3.2.8 cublasSrotg - construct the Givens rotation matrix

This function constructs the Givens rotation matrix $G = \begin{pmatrix} c & s \\ -s & c \end{pmatrix}$ that zeros out the 2×1 vector $\begin{pmatrix} a \\ b \end{pmatrix}$ i.e. $\begin{pmatrix} c & s \\ -s & c \end{pmatrix}\begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} r \\ 0 \end{pmatrix}$, where $c^2 + s^2 = 1, \quad r^2 = a^2 + b^2$.

```
// nvcc 009srotg.c -lcublas
// This function is provided for completeness and runs
// exclusively on the host
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include "cublas_v2.h"
int main(void){
```

```
  cublasStatus_t stat;             // CUBLAS functions status
  cublasHandle_t handle;                    // CUBLAS context
  int j;
  float a=1.0;
  float b=1.0;
  printf("a: %7.3f\n",a);                          // print a
  printf("b: %7.3f\n",b);                          // print b
  stat = cublasCreate(&handle);  // initialize CUBLAS context
  float c;
  float s;
//                                        [  c  s ]
// find the Givens rotation matrix G =[       ]
//                                        [ -s  c ]
//              [a] [r]
//  such that G*[ ]=[ ]
//              [b] [0]
//
// c^2+s^2=1,   r=\sqrt{a^2+b^2}, a is replaced by r

  stat=cublasSrotg(handle,&a,&b,&c,&s);

  printf("After Srotg:\n");
  printf("a: %7.5f\n",a);                          // print a
  printf("c: %7.5f\n",c);                          // print c
  printf("s: %7.5f\n",s);                          // print s
  cublasDestroy(handle);            // destroy CUBLAS context
  return EXIT_SUCCESS;
}
// a:   1.000
// b:   1.000

// After Srotg:
// a: 1.41421                                 // \sqrt{1^2+1^2}
// c: 0.70711                                      // cos(pi/4)
// s: 0.70711                                      // sin(pi/4)
//                        // [ 0.70711 0.70711] [1] [1.41422]
//                        // [                ]*[ ]=[       ]
//                        // [-0.70711 0.70711] [1] [   0   ]
```

3.2.9 cublasSrotm - apply the modified Givens rotation

This function multiplies the modified Givens 2×2 matrix $\begin{pmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{pmatrix}$ with $2 \times n$ matrix $\begin{pmatrix} x_0 & \dots & x_{n-1} \\ y_0 & \dots & y_{n-1} \end{pmatrix}$.

```
// nvcc 010srotmVec.c  -lcublas
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
```

```
#include "cublas_v2.h"
#define n 6                                     // length of x,y
int main(void){
  cudaError_t cudaStat;                       // cudaMalloc status
  cublasStatus_t stat;                  // CUBLAS functions status
  cublasHandle_t handle;                        // CUBLAS context
  int j;                                      // index of elements
  float* x;                                 // n-vector on the host
  float* y;                                 // n-vector on the host
  float* param;
  x=(float *)malloc (n*sizeof(*x));// host memory alloc for x
  for(j=0;j<n;j++)
    x[j]=(float)j;                            // x={0,1,2,3,4,5}
  printf("x:\n");
  for(j=0;j<n;j++)
    printf("%3.0f,",x[j]);                              // print x
  printf("\n");
  y=(float *)malloc (n*sizeof(*y));// host memory alloc for y
  for(j=0;j<n;j++)
    y[j]=(float)j*j;                          // y={0,1,4,9,16,25}
  printf("y:\n");
  for(j=0;j<n;j++)
    printf("%3.0f,",y[j]);                              // print y
  printf("\n");
  param=(float *)malloc (5*sizeof(*param));
  param[0]=1.0f;                                          // flag
  param[1]=0.5f;                         // param[1],...,param[4]
  param[2]=1.0f;                  //-entries of the Givens matrix
  param[3]=-1.0f;                  // h11=param[1]  h12=param[2]
  param[4]=0.5f;                   // h21=param[3]  h22=param[4]
// on the device
  float* d_x;                              // d_x - x on the device
  float* d_y;                              // d_y - y on the device
  cudaStat=cudaMalloc((void**)&d_x,n*sizeof(*x));    //device
                                        // memory alloc for x
  cudaStat=cudaMalloc((void**)&d_y,n*sizeof(*y));    //device
                                        // memory alloc for y
  stat =cublasCreate(&handle);   // initialize CUBLAS context
  stat =cublasSetVector(n,sizeof(*x),x,1,d_x,1);//copy x->d_x
  stat =cublasSetVector(n,sizeof(*y),y,1,d_y,1);//copy y->d_y
//                                                  [0.5  1.0 ]
// multiply the 2x2 modified Givens matrix      H=[         ]
// by the 2xn matrix with two rows x and y        [-1.0 0.5 ]

  stat=cublasSrotm(handle,n,d_x,1,d_y,1,param);

  stat=cublasGetVector(n,sizeof(float),d_x,1,x,1);//cp d_x->x
  printf("x after Srotm x:\n");        // print x after Srotm
  for(j=0;j<n;j++)
    printf("%7.3f,",x[j]);
  printf("\n");
  stat=cublasGetVector(n,sizeof(float),d_y,1,y,1);//cp d_y->y
```

```
  printf("y after Srotm y:\n");        // print y after Srotm
  for(j=0;j<n;j++)
    printf("%7.3f,",y[j]);
  printf("\n");
  cudaFree(d_x);                        // free device memory
  cudaFree(d_y);                        // free device memory
  cublasDestroy(handle);              // destroy CUBLAS context
  free(x);                                // free host memory
  free(y);                                // free host memory
  free(param);                            // free host memory
  return EXIT_SUCCESS;
}
// x:
//  0,  1,  2,  3,  4,  5,
// y:
//  0,  1,  4,  9, 16, 25,

// x after Srotm:
//  0.000,  1.500,  5.000, 10.500, 18.000, 27.500,
// y after Srotm:
//  0.000, -0.500,  0.000,  1.500,  4.000,  7.500,
//                       // [x]  [ 0.5   1 ] [0 1 2 3  4  5]
//                       // [ ]= [         ]*[             ]
//                       // [y]  [ -1   0.5] [0 1 4 9 16 25]
```

3.2.10 cublasSrotmg - construct the modified Givens rotation matrix

This function constructs the modified Givens transformation $\begin{pmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{pmatrix}$ that zeros out the second entry of the vector $\begin{pmatrix} \sqrt{d1} * x1 \\ \sqrt{d2} * y1 \end{pmatrix}$.

```
// nvcc 011srotmg.c -lcublas
// this function is provided for completeness
// and runs exclusively on the Host
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include "cublas_v2.h"
int main(void){
  cublasStatus_t stat;              // CUBLAS functions status
  cublasHandle_t handle;                    // CUBLAS context
  float d1=5.0f;                                  // d1=5.0
  float d2=5.0f;                                  // d2=5.0
  float param[5];       //   [param[1]  param[2]]   [h11 h12]
                        //   [                  ] = [       ]
                        //   [param[3]  param[4]]   [h21 h22]
  param[0]=1.0f;                          // param[0] is a flag
// if param[0]=1.0, then h12=1=param[2], h21=-1=param[3]
```

```
  printf("d1: %7.3f\n",d1);                          // print d1
  printf("d2: %7.3f\n",d2);                          // print d2
  stat = cublasCreate(&handle);  // initialize CUBLAS context
  float x1=1.0f;                                        // x1=1
  float y1=2.0f;                                        // y1=2
  printf("x1: %7.3f\n",x1);                          // print x1
  printf("y1: %7.3f\n",y1);                          // print y1
//find modified Givens rotation matrix H={{h11,h12},{h21,h22}}
//such that the second entry of H*{\sqrt{d1}*x1,\sqrt{d2}*y1}^T
//is zero

  stat=cublasSrotmg(handle,&d1,&d2,&x1,&y1,param);

  printf("After srotmg:\n");
  printf("param[0]: %4.2f\n",param[0]);
  printf("h11: %7.5f\n",param[1]);
  printf("h22: %7.5f\n",param[4]);
//check if the second entry of H*{\sqrt{d1)*x1,\sqrt{d2}*y1}^T
//is zero; the values of d1,d2,x1 are overwritten so we use
//their initial values
  printf("%7.5f\n",(-1.0)*sqrt(5.0)*1.0+
                                  param[4]*sqrt(5.0)*2.0);
  cublasDestroy(handle);            // destroy CUBLAS context
  return EXIT_SUCCESS;
}
// d1:   5.000          // [d1] [5]    [x1] [1]       [0.5  1 ]
// d2:   5.000          // [  ]=[ ],   [  ]=[ ],   H=[       ]
// x1:   1.000          // [d2] [5]    [x2] [2]       [-1  0.5]
// y1:   2.000

// After srotmg:
// param[0]: 1.00
// h11: 0.50000
// h22: 0.50000

//   [sqrt(d1)*x1] [0.5  1 ] [sqrt(5)*1] [5.59]
// H*[           ]=[       ]*[         ]=[    ]
//   [sqrt(d2)*y1] [-1  0.5] [sqrt(5)*2] [  0 ]

// 0.00000 <== the second entry of
// H*{sqrt(d1)*x1,sqrt(d2)*y1}^T
```

3.2.11 `cublasSscal` - scale the vector

This function scales the vector x by the scalar α.

$$x = \alpha x.$$

```
// nvcc 012sscal.c -lcublas
#include <stdio.h>
#include <stdlib.h>
```

```
#include <cuda_runtime.h>
#include "cublas_v2.h"
#define n 6                                        // length of x
int main(void){
  cudaError_t cudaStat;                          // cudaMalloc status
  cublasStatus_t stat;                       // CUBLAS functions status
  cublasHandle_t handle;                           // CUBLAS context
  int j;                                         // index of elements
  float* x;                                   // n-vector on the host
  x=(float *)malloc (n*sizeof(*x));// host memory alloc for x
  for(j=0;j<n;j++)
    x[j]=(float)j;                              // x={0,1,2,3,4,5}
  printf("x:\n");
  for(j=0;j<n;j++)
    printf("%2.0f,",x[j]);                             // print x
  printf("\n");
// on the device
  float* d_x;                               // d_x - x on the device
  cudaStat=cudaMalloc((void**)&d_x,n*sizeof(*x));    //device
                                           // memory alloc for x
  stat = cublasCreate(&handle);  // initialize CUBLAS context
  stat = cublasSetVector(n,sizeof(*x),x,1,d_x,1);// cp x->d_x
  float al=2.0;                                            // al=2
// scale the vector d_x by the scalar al: d_x = al*d_x

  stat=cublasSscal(handle,n,&al,d_x,1);

  stat=cublasGetVector(n,sizeof(float),d_x,1,x,1);//cp d_x->x
  printf("x after Sscal:\n");          // print x after Sscal:
  for(j=0;j<n;j++)
    printf("%2.0f,",x[j]);                 // x={0,2,4,6,8,10}
  printf("\n");
  cudaFree(d_x);                             // free device memory
  cublasDestroy(handle);              // destroy CUBLAS context
  free(x);                                   // free host memory
  return EXIT_SUCCESS;
}
// x:
// 0, 1, 2, 3, 4, 5,

// x after Sscal:
// 0, 2, 4, 6, 8,10,                          // 2*{0,1,2,3,4,5}
```

3.2.12 cublasSswap - swap two vectors

This function interchanges the elements of vector x and y

$$x \leftarrow y, \quad y \leftarrow x.$$

```
// nvcc 013sswap.c -lcublas
#include <stdio.h>
```

```
#include <stdlib.h>
#include <cuda_runtime.h>
#include "cublas_v2.h"
#define n 6                                     // length of x,y
int main(void){
  cudaError_t cudaStat;                       // cudaMalloc status
  cublasStatus_t stat;                  // CUBLAS functions status
  cublasHandle_t handle;                        // CUBLAS context
  int j;                                      // index of elements
  float* x;                                  // n-vector on the host
  float* y;                                  // n-vector on the host
  x=(float *)malloc (n*sizeof(*x));// host memory alloc for x
  for(j=0;j<n;j++)
    x[j]=(float)j;                             // x={0,1,2,3,4,5}
  printf("x:\n");
  for(j=0;j<n;j++)
    printf("%2.0f,",x[j]);                             // print x
  printf("\n");
  y=(float *)malloc (n*sizeof(*y));// host memory alloc for y
  for(j=0;j<n;j++)
    y[j]=(float)2*j;                           // y={0,2,4,6,8,10}
  printf("y:\n");
  for(j=0;j<n;j++)
    printf("%2.0f,",y[j]);                             // print y
  printf("\n");
// on the device
  float* d_x;                               // d_x - x on the device
  float* d_y;                               // d_y - y on the device
  cudaStat=cudaMalloc((void**)&d_x,n*sizeof(*x));   // device
                                          // memory alloc for x
  cudaStat=cudaMalloc((void**)&d_y,n*sizeof(*y));   // device
                                          //memory alloc for y
  stat = cublasCreate(&handle);  // initialize CUBLAS context
  stat = cublasSetVector(n,sizeof(*x),x,1,d_x,1);// cp x->d_x
  stat = cublasSetVector(n,sizeof(*y),y,1,d_y,1);// cp y->d_y
// swap the vectors d_x,d_y:   d_x<--d_y, d_y<--d_x

  stat=cublasSswap(handle,n,d_x,1,d_y,1);

  stat=cublasGetVector(n,sizeof(float),d_y,1,y,1);//cp d_y->y
  stat=cublasGetVector(n,sizeof(float),d_x,1,x,1);//cp d_x->x
  printf("x after Sswap:\n");          // print x after Sswap:
  for(j=0;j<n;j++)
    printf("%2.0f,",x[j]);                 // x={0,2,4,6,8,10}
  printf("\n");
  printf("y after Sswap:\n");          // print y after Sswap:
  for(j=0;j<n;j++)
    printf("%2.0f,",y[j]);                   // y={0,1,2,3,4,5}
  printf("\n");
  cudaFree(d_x);                            // free device memory
  cudaFree(d_y);                            // free device memory
  cublasDestroy(handle);              // destroy CUBLAS context
```

```
  free(x);                                    // free host memory
  free(y);                                    // free host memory
  return EXIT_SUCCESS;
}
// x:
// 0, 1, 2, 3, 4, 5,
// y:
// 0, 2, 4, 6, 8,10,

// x after Sswap:
// 0, 2, 4, 6, 8,10,                    // x <- y
// y after Sswap:
// 0, 1, 2, 3, 4, 5,                    // y <- x
```

3.3 CUBLAS Level-2. Matrix-vector operations

3.3.1 cublasSgbmv – banded matrix-vector multiplication

This function performs the banded matrix-vector multiplication

$$y = \alpha\, op(A)x + \beta y,$$

where A is a banded matrix with ku superdiagonals and kl subdiagonals, x, y are vectors, α, β are scalars and $op(A)$ can be equal to A (CUBLAS_OP_N case), A^T (transposition) in CUBLAS_OP_T case or A^H (conjugate transposition) in CUBLAS_OP_C case. The highest superdiagonal is stored in row 0, starting from position ku, the next superdiagonal is stored in row 1 starting from position $ku-1, \ldots$. The main diagonal is stored in row ku, starting from position 0, the first subdiagonal is stored in row $ku+1$, starting from position 0, the next subdiagonal is stored in row $ku+2$ from position $0, \ldots$.

```
// nvcc 013sgbmv.c -lcublas
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <cuda_runtime.h>
#include "cublas_v2.h"
#define IDX2C(i,j,ld) (((j)*(ld))+( i ))
#define m 5                                   // number of rows
#define n 6                                 // number of columns
#define ku 2                          // number of superdiagonals
#define kl 1                            // number of subdiagonals
int main(void){
  cudaError_t cudaStat;                       // cudaMalloc status
  cublasStatus_t stat;                 // CUBLAS functions status
  cublasHandle_t handle;                        // CUBLAS context
  int i,j;                                 // row and column index
// declaration and allocation of a,x,y on the host
  float* a; //mxn matrix on the host    // a:
```

```
  float* x; //n-vector on the host      //   20 15 11
  float* y; //m-vector on the host      //   25 21 16 12
  a=(float*)malloc(m*n*sizeof(float)); //      26 22 17 13
// host memory alloc for a             //         27 23 18 14
  x=(float*)malloc(n*sizeof(float));   //            28 24 19
// host memory alloc for x
  y=(float*)malloc(m*sizeof(float));//host memory alloc for y
  int ind=11;
// highest superdiagonal 11,12,13,14 in first row,
// starting from i=ku
  for(i=ku;i<n;i++) a[IDX2C(0,i,m)]=(float)ind++;
// next superdiagonal 15,16,17,18,19 in next row,
// starting from i=ku-1
  for(i=ku-1;i<n;i++) a[IDX2C(1,i,m)]=(float)ind++;
// main diagonal 20,21,22,23,24 in  row ku, starting from i=0
  for(i=0;i<n-1;i++) a[IDX2C(ku,i,m)]=(float)ind++;
// subdiagonal 25,26,27,28 in ku+1 row, starting from i=0
  for(i=0;i<n-2;i++) a[IDX2C(ku+1,i,m)]=(float)ind++;
  for(i=0;i<n;i++) x[i]=1.0f;             // x={1,1,1,1,1,1}^T
  for(i=0;i<m;i++) y[i]=0.0f;               // y={0,0,0,0,0}^T
// on the device
  float* d_a;                          // d_a - a on the device
  float* d_x;                          // d_x - x on the device
  float* d_y;                          // d_y - y on the device
  cudaStat=cudaMalloc((void**)&d_a,m*n*sizeof(*a)); // device
                                       // memory alloc for a
  cudaStat=cudaMalloc((void**)&d_x,n*sizeof(*x));   // device
                                       // memory alloc for x
  cudaStat=cudaMalloc((void**)&d_y,m*sizeof(*y));   // device
                                       // memory alloc for y
  stat = cublasCreate(&handle);
  stat =cublasSetMatrix(m,n,sizeof(*a),a,m,d_a,m);//cp a->d_a
  stat = cublasSetVector(n,sizeof(*x),x,1,d_x,1); //cp x->d_x
  stat = cublasSetVector(m,sizeof(*y),y,1,d_y,1); //cp y->d_y
  float al=1.0f;                                      // al=1
  float bet=1.0f;                                    // bet=1
// banded matrix-vector multiplication:
//  d_y = al*d_a*d_x + bet*d_y;      d_a - mxn banded matrix;
//  d_x - n-vector, d_y - m-vector; al,bet - scalars

  stat=cublasSgbmv(handle,CUBLAS_OP_N,m,n,kl,ku,&al,d_a,m,d_x,1,
                                                    &bet,d_y,1);

  stat=cublasGetVector(m,sizeof(*y),d_y,1,y,1);// copy d_y->y
  printf("y after Sgbmv:\n");          // print y after Sgbmv
  for(j=0;j<m;j++){
      printf("%7.0f",y[j]);
      printf("\n");
  }
  cudaFree(d_a);                        // free device memory
  cudaFree(d_x);                        // free device memory
  cudaFree(d_y);                        // free device memory
```

```
  cublasDestroy(handle);              // destroy CUBLAS context
  free(a);                                  // free host memory
  free(x);                                  // free host memory
  free(y);                                  // free host memory
return EXIT_SUCCESS;
}
// y after Sgbmv:                 //                          [1]
//     46                         //  [ 20 15 11           ] [1]
//     74                         //  [ 25 21 16 12        ] [1]
//     78                         //  [    26 22 17 13     ]*[ ]
//     82                         //  [       27 23 18 14  ] [1]
//     71                         //  [          28 24 19  ] [1]
                                   //                          [1]
```

3.3.2 cublasSgemv – matrix-vector multiplication

This function performs matrix-vector multiplication

$$y = \alpha\, op(A)x + \beta y,$$

where A is a matrix, x, y are vectors, α, β are scalars and $op(A)$ can be equal to A (CUBLAS_OP_N case), A^T (transposition) in CUBLAS_OP_T case or A^H (conjugate transposition) in CUBLAS_OP_C case. A is stored column by column.

```
// nvcc 014sgemv.c -lcublas
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <cuda_runtime.h>
#include "cublas_v2.h"
#define IDX2C(i,j,ld) (((j)*(ld))+( i ))
#define m 6                               // number of rows of a
#define n 5                            // number of columns of a
int main(void){
  cudaError_t cudaStat;                     // cudaMalloc status
  cublasStatus_t stat;                // CUBLAS functions status
  cublasHandle_t handle;                       // CUBLAS context
  int i,j;                          // i-row index, j-column index
  float* a;                           // a -mxn matrix on the host
  float* x;                            // x - n-vector on the host
  float* y;                            // y - m-vector on the host
  a=(float*)malloc(m*n*sizeof(float));//host mem. alloc for a
  x=(float*)malloc(n*sizeof(float));  //host mem. alloc for x
  y=(float*)malloc(m*sizeof(float));  //host mem. alloc for y
// define an mxn matrix a - column by column
  int ind=11;                                  // a:
  for(j=0;j<n;j++){                            // 11,17,23,29,35
    for(i=0;i<m;i++){                          // 12,18,24,30,36
      a[IDX2C(i,j,m)]=(float)ind++;            // 13,19,25,31,37
```

```
    }                                                  // 14,20,26,32,38
   }                                                   // 15,21,27,33,39
                                                       // 16,22,28,34,40
  printf("a:\n");
   for(i=0;i<m;i++){
     for(j=0;j<n;j++){
       printf("%4.0f",a[IDX2C(i,j,m)]); // print a row by row
     }
    printf("\n");
   }
  for(i=0;i<n;i++) x[i]=1.0f;                  // x={1,1,1,1,1}^T
  for(i=0;i<m;i++) y[i]=0.0f;                 // y={0,0,0,0,0,0}^T
// on the device
  float* d_a;                              // d_a - a on the device
  float* d_x;                              // d_x - x on the device
  float* d_y;                              // d_y - y on the device
  cudaStat=cudaMalloc((void**)&d_a,m*n*sizeof(*a)); // device
                                           // memory alloc for a
  cudaStat=cudaMalloc((void**)&d_x,n*sizeof(*x));   // device
                                           // memory alloc for x
  cudaStat=cudaMalloc((void**)&d_y,m*sizeof(*y));   // device
                                           // memory alloc for y
  stat = cublasCreate(&handle);
  stat =cublasSetMatrix(m,n,sizeof(*a),a,m,d_a,m);//cp a->d_a
  stat = cublasSetVector(n,sizeof(*x),x,1,d_x,1); //cp x->d_x
  stat = cublasSetVector(m,sizeof(*y),y,1,d_y,1); //cp y->d_y
  float al=1.0f;                                          // al=1
  float bet=1.0f;                                        // bet=1
// matrix-vector multiplication:   d_y = al*d_a*d_x + bet*d_y
// d_a - mxn matrix; d_x - n-vector, d_y - m-vector;
// al,bet - scalars

  stat=cublasSgemv(handle,CUBLAS_OP_N,m,n,&al,d_a,m,d_x,1,&bet,
                                                      d_y,1);

  stat=cublasGetVector(m,sizeof(*y),d_y,1,y,1); //copy d_y->y
  printf("y after Sgemv::\n");
  for(j=0;j<m;j++){
      printf("%5.0f",y[j]);             // print y after Sgemv
      printf("\n");
  }
  cudaFree(d_a);                           // free device memory
  cudaFree(d_x);                           // free device memory
  cudaFree(d_y);                           // free device memory
  cublasDestroy(handle);                 // destroy CUBLAS context
  free(a);                                   // free host memory
  free(x);                                   // free host memory
  free(y);                                   // free host memory
return EXIT_SUCCESS;
}
// a:
//   11   17   23   29   35
```

```
//   12   18   24   30   36
//   13   19   25   31   37
//   14   20   26   32   38
//   15   21   27   33   39
//   16   22   28   34   40

// y after Sgemv:
//   115                          //   [11   17   23   29   35]   [1]
//   120                          //   [12   18   24   30   36]   [1]
//   125                          //   [13   19   25   31   37]*  [1]
//   130                          //   [14   20   26   32   38]   [1]
//   135                          //   [15   21   27   33   39]   [1]
//   140                          //   [16   22   28   34   40]
```

3.3.3 `cublasSger` - rank one update

This function performs the rank-1 update

$$A = \alpha x y^T + A \quad \text{or} \quad A = \alpha x y^H + A,$$

where x, y are vectors, A is a $m \times n$ matrix and α is a scalar.

```
// nvcc 015sger.c -lcublas
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <cuda_runtime.h>
#include "cublas_v2.h"
#define IDX2C(i,j,ld) (((j)*(ld))+( i ))
#define m 6                              // number of rows of a
#define n 5                            // number of columns of a
int main(void){
  cudaError_t cudaStat;                    // cudaMalloc status
  cublasStatus_t stat;               // CUBLAS functions status
  cublasHandle_t handle;                      // CUBLAS context
  int i,j;                       // i-row index, j-column index
  float* a;                        // a -mxn matrix on the host
  float* x;                          // x -n-vector on the host
  float* y;                          // y -m-vector on the host
  a=(float*)malloc(m*n*sizeof(float));//host mem. alloc for a
  x=(float*)malloc(n*sizeof(float));  //host mem. alloc for x
  y=(float*)malloc(m*sizeof(float));  //host mem. alloc for y
// define an mxn matrix a column by column
  int ind=11;                                    // a:
  for(j=0;j<n;j++){                              // 11,17,23,29,35
    for(i=0;i<m;i++){                            // 12,18,24,30,36
      a[IDX2C(i,j,m)]=(float)ind++;              // 13,19,25,31,37
    }                                            // 14,20,26,32,38
   }                                             // 15,21,27,33,39
                                                 // 16,22,28,34,40
```

```
  printf("a:\n");
   for(i=0;i<m;i++){
     for(j=0;j<n;j++){
       printf("%4.0f",a[IDX2C(i,j,m)]); // print a row by row
     }
    printf("\n");
   }
  for(i=0;i<m;i++) x[i]=1.0f;              // x={1,1,1,1,1,1}^T
  for(i=0;i<n;i++) y[i]=1.0f;                // y={1,1,1,1,1}^T
// on the device
  float* d_a;                            // d_a - a on the device
  float* d_x;                            // d_x - x on the device
  float* d_y;                            // d_y - y on the device
  cudaStat=cudaMalloc((void**)&d_a,m*n*sizeof(*a)); // device
                                         // memory alloc for a
  cudaStat=cudaMalloc((void**)&d_x,m*sizeof(*x));   // device
                                         // memory alloc for x
  cudaStat=cudaMalloc((void**)&d_y,n*sizeof(*y));   // device
                                         // memory alloc for y
  stat = cublasCreate(&handle);  // initialize CUBLAS context
  stat =cublasSetMatrix(m,n,sizeof(*a),a,m,d_a,m);//cp a->d_a
  stat = cublasSetVector(m,sizeof(*x),x,1,d_x,1); //cp x->d_x
  stat = cublasSetVector(n,sizeof(*y),y,1,d_y,1); //cp y->d_y
  float al=2.0f;                                          // al=2
// rank-1 update of d_a:  d_a = al*d_x*d_y^T + d_a
// d_a -mxn matrix;  d_x -m-vector, d_y -n-vector; al -scalar

  stat=cublasSger(handle,m,n,&al,d_x,1,d_y,1,d_a,m);

  stat=cublasGetMatrix(m,n,sizeof(*a),d_a,m,a,m); //cp d_a->a
// print the updated a row by row
  printf("a after Sger :\n");
  for(i=0;i<m;i++){
    for(j=0;j<n;j++){
      printf("%4.0f",a[IDX2C(i,j,m)]);  // print a after Sger
    }
    printf("\n");
  }
  cudaFree(d_a);                         // free device memory
  cudaFree(d_x);                         // free device memory
  cudaFree(d_y);                         // free device memory
  cublasDestroy(handle);              // destroy CUBLAS context
  free(a);                                 // free host memory
  free(x);                                 // free host memory
  free(y);                                 // free host memory
  return EXIT_SUCCESS;
}
// a:
//   11   17   23   29   35
//   12   18   24   30   36
//   13   19   25   31   37
//   14   20   26   32   38
```

```
//   15   21   27   33   39
//   16   22   28   34   40

// a after Sger :
//   13   19   25   31   37
//   14   20   26   32   38
//   15   21   27   33   39
//   16   22   28   34   40
//   17   23   29   35   41
//   18   24   30   36   42

//        [1]                 [11   17   23   29   35]
//        [1]                 [12   18   24   30   36]
//        [1]                 [13   19   25   31   37]
// =   2*[ ]*[1,1,1,1,1] + [                       ]
//        [1]                 [14   20   26   32   38]
//        [1]                 [15   21   27   33   39]
//        [1]                 [16   22   28   34   40]
```

3.3.4 `cublasSsbmv` - symmetric banded matrix-vector multiplication

This function performs the symmetric banded matrix-vector multiplication

$$y = \alpha\, Ax + \beta y,$$

where A is an $n \times n$ symmetric banded matrix with k subdiagonals and superdiagonals, x, y are vectors and α, β are scalars. The matrix A can be stored in lower (`CUBLAS_FILL_MODE_LOWER`) or upper mode (`CUBLAS_FILL_MODE_UPPER`). In both modes it is stored column by column. In lower mode the main diagonal is stored in row 0 (starting at position 0) the second subdiagonal in row 1 (starting at position 0) and so on.

```
// nvcc 016ssbmv.c -lcublas
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include "cublas_v2.h"
#define IDX2C(i,j,ld) (((j)*(ld))+( i ))
#define n 6                 // number of rows and columns of a
#define k 1      // number of subdiagonals and superdiagonals
int main(void){
  cudaError_t cudaStat;                    // cudaMalloc status
  cublasStatus_t stat;              // CUBLAS functions status
  cublasHandle_t handle;                      // CUBLAS context
  int i,j;                           // row index, column index
  float *a; //nxn matrix a on the host //lower triangle of a:
  float *x; // n-vector x on the host     //11
  float *y; // n-vector y on the host     //17,12
```

```
  a=(float*)malloc(n*n*sizeof(float));     //   18,13
// memory alloc for a on the host           //      19,14
  x=(float*)malloc(n*sizeof(float));       //         20,15
// memory alloc for x on the host           //            21,16
  y=(float*)malloc(n*sizeof(float));       // mem. alloc for y
                                                  //on the host
// main diagonal and subdiagonals of a in rows
  int ind=11;
  for(i=0;i<n;i++) a[i*n]=(float)ind++;      // main diagonal:
                                           // 11,12,13,14,15,16
  for(i=0;i<n-1;i++) a[i*n+1]=(float)ind++;// first subdiag.:
                                             // 17,18,19,20,21
  for(i=0;i<n;i++){x[i]=1.0f;y[i]=0.0f;} // x={1,1,1,1,1,1}^T
                                           // y={0,0,0,0,0,0}^T
// on the device
  float* d_a;                          // d_a - a on the device
  float* d_x;                          // d_x - x on the device
  float* d_y;                          // d_y - y on the device
  cudaStat=cudaMalloc((void**)&d_a,n*n*sizeof(*a)); // device
                                         // memory alloc for a
  cudaStat=cudaMalloc((void**)&d_x,n*sizeof(*x));   // device
                                         // memory alloc for x
  cudaStat=cudaMalloc((void**)&d_y,n*sizeof(*y));   // device
                                        //  memory alloc for y
  stat = cublasCreate(&handle);  // initialize CUBLAS context
  stat =cublasSetMatrix(n,n,sizeof(*a),a,n,d_a,n);//cp a->d_a
  stat = cublasSetVector(n,sizeof(*x),x,1,d_x,1); //cp x->d_x
  stat = cublasSetVector(n,sizeof(*y),y,1,d_y,1); //cp y->d_y
  float al=1.0f;                                      // al=1
  float bet=1.0f;                                    // bet=1
// symmetric banded matrix-vector multiplication:
// d_y = al*d_a*d_x + bet*d_y,
// d_a - nxn symmetric banded matrix;
// d_x,d_y - n-vectors; al,bet - scalars

  stat=cublasSsbmv(handle,CUBLAS_FILL_MODE_LOWER,n,k,&al,d_a,n,
                                          d_x,1,&bet,d_y,1);

  stat=cublasGetVector(n,sizeof(*y),d_y,1,y,1); //copy d_y->y
  printf("y after Ssbmv:\n");
  for(j=0;j<n;j++){
      printf("%7.0f",y[j]);            // print y after Ssbmv
      printf("\n");
  }
  cudaFree(d_a);                        // free device memory
  cudaFree(d_x);                        // free device memory
  cudaFree(d_y);                        // free device memory
  cublasDestroy(handle);              // destroy CUBLAS context
  free(a);                              // free host memory
  free(x);                              // free host memory
  free(y);                              // free host memory
  return EXIT_SUCCESS;
```

```
}
// y after Ssbmv:
//     28                    //   [11 17             ] [1]    [28]
//     47                    //   [17 12 18          ] [1]    [47]
//     50                    //   [   18 13 19       ] [1] =  [50]
//     53                    //   [      19 14 20    ] [1]    [53]
//     56                    //   [         20 15 21] [1]    [56]
//     37                    //   [            21 16] [1]    [37]
```

3.3.5 cublasSspmv - symmetric packed matrix-vector multiplication

This function performs the symmetric packed matrix-vector multiplication

$$y = \alpha\, Ax + \beta y,$$

where A is a symmetric matrix in packed format, x, y are vectors and α, β - scalars. The symmetric $n \times n$ matrix A can be stored in lower (CUBLAS_FILL_MODE_LOWER) or upper mode (CUBLAS_FILL_MODE_UPPER). In lower mode the elements of the lower triangular part of A are packed together column by column without gaps.

```
// nvcc 017sspmv.c -lcublas
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include "cublas_v2.h"
#define n 6                 // number of rows and columns of a
int main(void){
  cudaError_t cudaStat;                  // cudaMalloc status
  cublasStatus_t stat;             // CUBLAS functions status
  cublasHandle_t handle;                    // CUBLAS context
  int i,j,l,m; // indices                 // a:
  float *a; //lower triangle of nxn       // 11
            //matrix a on the host        // 12,17
  float *x; // n-vector x on the host     // 13,18,22
  float *y; // n-vector y on the host     // 14,19,23,26
  a=(float*)malloc(n*(n+1)/2*sizeof(*a));// 15,20,24,27,29
//memory alloc for a on the host          // 16,21,25,28,30,31
  x=(float*)malloc(n*sizeof(float));      //memory alloc for x
                                                 //on the host
  y=(float*)malloc(n*sizeof(float));      //memory alloc for y
                                                 //on the host
//define the lower triangle of a symmetric a in packed format
//column by column without gaps
  for(i=0;i<n*(n+1)/2;i++) a[i]=(float)(11+i);
// print the upper triangle of a  row by row
  printf("upper triangle of a:\n");
  l=n;j=0;m=0;
```

```
  while(l>0){
    for(i=0;i<m;i++) printf("   ");
    for(i=j;i<j+l;i++) printf("%3.0f",a[i]);
    printf("\n");
    m++;j=j+l;l--;
  }
  for(i=0;i<n;i++){x[i]=1.0f;y[i]=0.0f;} // x={1,1,1,1,1,1}^T
                                         // y={0,0,0,0,0,0}^T
// on the device
  float* d_a;                          // d_a - a on the device
  float* d_x;                          // d_x - x on the device
  float* d_y;                          // d_y - y on the device
  cudaStat=cudaMalloc((void**)&d_a,n*(n+1)/2*sizeof(*a));
                                  // device memory alloc for a
  cudaStat=cudaMalloc((void**)&d_x,n*sizeof(*x));    //device
                                       // memory alloc for x
  cudaStat=cudaMalloc((void**)&d_y,n*sizeof(*x));    //device
                                       // memory alloc for y
  stat = cublasCreate(&handle);  // initialize CUBLAS context
  stat = cublasSetVector(n*(n+1)/2,sizeof(*a),a,1,d_a,1);
                                              // copy a->d_a
  stat = cublasSetVector(n,sizeof(*x),x,1,d_x,1); //cp x->d_x
  stat = cublasSetVector(n,sizeof(*y),y,1,d_y,1); //cp y->d_y
  float al=1.0f;                                   // al=1
  float bet=1.0f;                                  // bet=1
// symmetric packed matrix-vector multiplication:
// d_y = al*d_a*d_x + bet*d_y; d_a -symmetric nxn matrix
// in packed format; d_x,d_y - n-vectors;  al,bet - scalars

  stat=cublasSspmv(handle,CUBLAS_FILL_MODE_LOWER,n,&al,d_a,d_x,1,
                                                   &bet,d_y,1);

  stat=cublasGetVector(n,sizeof(*y),d_y,1,y,1);// copy d_y->y
  printf("y after Sspmv:\n");          // print y after Sspmv
  for(j=0;j<n;j++){
      printf("%7.0f",y[j]);
      printf("\n");
  }
  cudaFree(d_a);                        // free device memory
  cudaFree(d_x);                        // free device memory
  cudaFree(d_y);                        // free device memory
  cublasDestroy(handle);             // destroy CUBLAS context
  free(a);                                // free host memory
  free(x);                                // free host memory
  free(y);                                // free host memory
  return EXIT_SUCCESS;
}
// upper triangle of a:
// 11  12  13  14  15  16
//     17  18  19  20  21
//         22  23  24  25
//             26  27  28
```

```
//              29 30
//                 31

// y after Sspmv:
//     81                  //  [11 12 13 14 15 16] [1]    [ 81]
//    107                  //  [12 17 18 19 20 21] [1]    [107]
//    125                  //  [13 18 22 23 24 25] [1] =  [125]
//    137                  //  [14 19 23 26 27 28] [1]    [137]
//    145                  //  [15 20 24 27 29 30] [1]    [145]
//    151                  //  [16 21 25 28 30 31] [1]    [151]
```

3.3.6 `cublasSspr` - symmetric packed rank-1 update

This function performs the symmetric packed rank-1 update

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

where A is a symmetric matrix in packed format, x is a vector and α is a scalar. The symmetric $n \times n$ matrix A can be stored in lower (`CUBLAS_FILL_MODE_LOWER`) or upper mode (`CUBLAS_FILL_MODE_UPPER`). In lower mode the elements of the lower triangular part of A are packed together column by column without gaps.

```
// nvcc 018sspr.c -lcublas
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include "cublas_v2.h"
#define n 6                 // number of rows and columns of a
int main(void){
  cudaError_t cudaStat;                    // cudaMalloc status
  cublasStatus_t stat;              // CUBLAS functions status
  cublasHandle_t handle;                    //   CUBLAS context
  int i,j,l,m;                              //a:
  float *a; //lower triangle of a           //11
  float *x; // n-vector x                   //12,17
  a=(float*)malloc(n*(n+1)/2*sizeof(*a)); //13,18,22
// memory alloc for a on the host          //14,19,23,26
  x=(float*)malloc(n*sizeof(float));       //15,20,24,27,29
// memory alloc for x on the host          //16,21,25,28,30,31
//define the lower triangle of a symmetric a in packed format
//column by column without gaps
  for(i=0;i<n*(n+1)/2;i++) a[i]=(float)(11+i);
// print the upper triangle of a  row by row
  printf("upper triangle of a:\n");
  l=n;j=0;m=0;
  while(l>0){
    for(i=0;i<m;i++) printf("   ");
    for(i=j;i<j+l;i++) printf("%3.0f",a[i]);
```

```
    printf("\n");
    m++;j=j+l;l--;
  }
  for(i=0;i<n;i++){x[i]=1.0f;}           // x={1,1,1,1,1,1}^T

// on the device
  float* d_a;                          // d_a - a on the device
  float* d_x;                          // d_x - x on the device
  cudaStat=cudaMalloc((void**)&d_a,n*(n+1)/2*sizeof(*a));
                                  // device memory alloc for a
  cudaStat=cudaMalloc((void**)&d_x,n*sizeof(*x));   // device
                                       // memory alloc for x
  stat = cublasCreate(&handle);  // initialize CUBLAS context
  stat = cublasSetVector(n*(n+1)/2,sizeof(*a),a,1,d_a,1);
                                                 // copy a -> d_a
  stat = cublasSetVector(n,sizeof(*x),x,1,d_x,1);// cp x->d_x
  float al=1.0f;                                       // al=1
// rank-1 update of a symmetric matrix d_a :
// d_a = al*d_x*d_x^T + d_a
// d_a - symmetric nxn matrix in packed format; d_x n-vector;
// al - scalar

  stat=cublasSspr(handle,CUBLAS_FILL_MODE_LOWER,n,&al,d_x,1,d_a);

  stat=cublasGetVector(n*(n+1)/2,sizeof(*a),d_a,1,a,1);
                                                 // copy d_a -> a
  printf("upper triangle of updated a after Sspr:\n");
  l=n;j=0;m=0;
  while(l>0){
    for(i=0;i<m;i++) printf("   ");          // upper triangle
    for(i=j;i<j+l;i++) printf("%3.0f",a[i]);//of a after Sspr
    printf("\n");
    m++;j=j+l;l--;
  }
  cudaFree(d_a);                          // free device memory
  cudaFree(d_x);                          // free device memory
  cublasDestroy(handle);             // destroy CUBLAS context
  free(a);                                  // free host memory
  free(x);                                  // free host memory
  return EXIT_SUCCESS;
}
// upper triangle of a:
// 11 12 13 14 15 16
//    17 18 19 20 21
//       22 23 24 25
//          26 27 28
//             29 30
//                31

// upper triangle of a after Sspr://    [1]
// 12 13 14 15 16 17               //    [1]
//    18 19 20 21 22               //    [1]
```

```
//        23 24 25 26              // 1*[ ]*[1,1,1,1,1,1] + a
//           27 28 29              //    [1]
//              30 31              //    [1]
//                 32              //    [1]
```

3.3.7 cublasSspr2 - symmetric packed rank-2 update

This function performs the symmetric packed rank-2 update

$$A = \alpha(xy^T + yx^T) + A,$$

where A is a symmetric matrix in packed format, x, y are vectors and α is a scalar. The symmetric $n \times n$ matrix A can be stored in lower (CUBLAS_FILL_MODE_LOWER) or upper mode (CUBLAS_FILL_MODE_UPPER). In lower mode the elements of the lower triangular part of A are packed together column by column without gaps.

```
// nvcc 019ssspr2.c -lcublas
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include "cublas_v2.h"
#define n 6                 // number of rows and columns of a
int main(void){
  cudaError_t cudaStat;                    // cudaMalloc status
  cublasStatus_t stat;              // CUBLAS functions status
  cublasHandle_t handle;                       // CUBLAS context
  int i,j,l,m;                                       // indices
  float *a; // lower triangle of a nxn matrix on the host
  float *x; // n-vector x on the host     // a:
  float *y; // n-vector x on the host     // 11
  a=(float*)malloc(n*(n+1)/2*sizeof(*a));// 12,17
// memory alloc for a on the host         // 13,18,22
  x=(float*)malloc(n*sizeof(float));      // 14,19,23,26
// memory alloc for x on the host         // 15,20,24,27,29
  y=(float*)malloc(n*sizeof(float));      // 16,21,25,28,30,31
// memory alloc for y on the host
//define the lower triangle of a symmetric matrix a in packed
// format column  by column without gaps
  for(i=0;i<n*(n+1)/2;i++) a[i]=(float)(11+i);
// print the upper triangle of a  row by row
  printf("upper triangle of a:\n");
  l=n;j=0;m=0;
  while(l>0){
    for(i=0;i<m;i++) printf("   ");
    for(i=j;i<j+l;i++) printf("%3.0f",a[i]);
    printf("\n");
    m++;j=j+l;l--;
  }
  for(i=0;i<n;i++){x[i]=1.0f;y[i]=2.0;}  // x={1,1,1,1,1,1}^T
```

```
                                          // y={2,2,2,2,2,2}^T
// on the device
  float* d_a;                           // d_a - a on the device
  float* d_x;                           // d_x - x on the device
  float* d_y;                           // d_y - y on the device
  cudaStat=cudaMalloc((void**)&d_a,n*(n+1)/2*sizeof(*a));
                                  // device memory alloc for a
  cudaStat=cudaMalloc((void**)&d_x,n*sizeof(*x));   // device
                                    //  memory alloc for x
  cudaStat=cudaMalloc((void**)&d_y,n*sizeof(*y));   // device
                                    // memory alloc for y
  stat = cublasCreate(&handle);  // initialize CUBLAS context
  stat = cublasSetVector(n*(n+1)/2,sizeof(*a),a,1,d_a,1);
                                                // copy a -> d_a
  stat = cublasSetVector(n,sizeof(*x),x,1,d_x,1); //cp x->d_x
  stat = cublasSetVector(n,sizeof(*y),y,1,d_y,1); //cp y->d_y
  float al=1.0f;                                    // al=1.0
// rank-2 update of symmetric matrix d_a :
// d_a = al*(d_x*d_y^T + d_y*d_x^T) + d_a
// d_a - symmetric  nxn matrix in packed form; x,y - n-vect.;
// al - scalar

  stat=cublasSspr2(handle,CUBLAS_FILL_MODE_LOWER,n,&al,d_x,1,d_y,
                                                        1,d_a);

  stat=cublasGetVector(n*(n+1)/2,sizeof(*a),d_a,1,a,1);
                                          // copy d_a -> a
// print the updated upper triangle of a row by row
  printf("upper triangle of updated a after Sspr2:\n");
  l=n;j=0;m=0;
  while(l>0){
    for(i=0;i<m;i++) printf("   ");
    for(i=j;i<j+l;i++) printf("%3.0f",a[i]);
    printf("\n");
    m++;j=j+l;l--;
  }
  cudaFree(d_a);                        // free device memory
  cudaFree(d_x);                        // free device memory
  cudaFree(d_y);                        // free device memory
  cublasDestroy(handle);              // destroy CUBLAS context
  free(a);                              // free host memory
  free(x);                              // free host memory
  free(y);                              // free host memory
  return EXIT_SUCCESS;
}
// upper triangle of a:
// 11 12 13 14 15 16
//    17 18 19 20 21
//       22 23 24 25
//          26 27 28
//             29 30
//                31
```

```
// upper triangle of a after Sspr2:
// 15 16 17 18 19 20
//    21 22 23 24 25
//       26 27 28 29
//          30 31 32
//             33 34
//                35
// [15 16 17 18 19 20]    [1]               [2]
// [16 21 22 23 24 25]    [1]               [2]
// [17 22 26 27 28 29]    [1]               [2]
// [                  ]=1*([ ]*[2,2,2,2,2,2]+[ ]*[1,1,1,1,1,1])+a
// [18 23 27 30 31 32]    [1]               [2]
// [19 24 28 31 33 34]    [1]               [2]
// [20 25 29 33 34 35]    [1]               [2]
```

3.3.8 `cublasSsymv` - symmetric matrix-vector multiplication

This function performs the symmetric matrix-vector multiplication.

$$y = \alpha Ax + \beta y,$$

where A is an $n \times n$ symmetric matrix, x, y are vectors and α, β are scalars. The matrix A can be stored in lower (`CUBLAS_FILL_MODE_LOWER`) or upper (`CUBLAS_FILL_MODE_UPPER`) mode.

```
// nvcc 020ssymv.c -lcublas
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include "cublas_v2.h"
#define IDX2C(i,j,ld) (((j)*(ld))+( i ))
#define n 6                  // number of rows and columns of a
int main(void){
  cudaError_t cudaStat;                    // cudaMalloc status
  cublasStatus_t stat;              // CUBLAS functions status
  cublasHandle_t handle;                   // CUBLAS context
  int i,j;                  // i - row index, j - column index
  float* a;                          // nxn matrix on the host
  float* x;                            // n-vector on the host
  float* y;                            // n-vector on the host
  a=(float*)malloc(n*n*sizeof(float));   // host memory for a
  x=(float*)malloc(n*sizeof(float));     // host memory for x
  y=(float*)malloc(n*sizeof(float));     // host memory for y
// define the lower triangle of an nxn symmetric matrix a
// in lower mode  column by column
  int ind=11;                            // a:
  for(j=0;j<n;j++){                      // 11
    for(i=0;i<n;i++){                    // 12,17
      if(i>=j){                          // 13,18,22
```

```
        a[IDX2C(i,j,n)]=(float)ind++;     // 14,19,23,26
      }                                   // 15,20,24,27,29
    }                                     // 16,21,25,28,30,31
  }
// print the lower triangle of a row by row
  printf("lower triangle of a:\n");
   for(i=0;i<n;i++){
     for(j=0;j<n;j++){
       if(i>=j)
       printf("%5.0f",a[IDX2C(i,j,n)]);
     }
    printf("\n");
   }
  for(i=0;i<n;i++){x[i]=1.0f;y[i]=1.0;}  // x={1,1,1,1,1,1}^T
                                         // y={1,1,1,1,1,1}^T
// on the device
  float* d_a;                          // d_a - a on the device
  float* d_x;                          // d_x - x on the device
  float* d_y;                          // d_y - y on the device
  cudaStat=cudaMalloc((void**)&d_a,n*n*sizeof(*a)); // device
                                       // memory alloc for a
  cudaStat=cudaMalloc((void**)&d_x,n*sizeof(*x));   // device
                                       // memory alloc for x
  cudaStat=cudaMalloc((void**)&d_y,n*sizeof(*y));   // device
                                       // memory alloc for y
  stat = cublasCreate(&handle);  // initialize CUBLAS context
  stat = cublasSetMatrix(n,n,sizeof(*a),a,n,d_a,n);
                                            // copy a -> d_a
  stat = cublasSetVector(n,sizeof(*x),x,1,d_x,1);// cp x->d_x
  stat = cublasSetVector(n,sizeof(*y),y,1,d_y,1);// cp y->d_y
  float al=1.0f;                             // al=1.0
  float bet=1.0f;                           // bet=1.0
// symmetric matrix-vector multiplication:
// d_y = al*d_a*d_x + bet*d_y
// d_a - nxn symmetric matrix; d_x,d_y - n-vectors;
// al,bet - scalars

  stat=cublasSsymv(handle,CUBLAS_FILL_MODE_LOWER,n,&al,d_a,n,
                                        d_x,1,&bet,d_y,1);

  stat=cublasGetVector(n,sizeof(float),d_y,1,y,1); // d_y->y
  printf("y after Ssymv:\n");          // print y after Ssymv
  for(j=0;j<n;j++)
  printf("%7.0f\n",y[j]);
  cudaFree(d_a);                        // free device memory
  cudaFree(d_x);                        // free device memory
  cudaFree(d_y);                        // free device memory
  cublasDestroy(handle);             // destroy CUBLAS context
  free(a);                              // free host memory
  free(x);                              // free host memory
  free(y);                              // free host memory
  return EXIT_SUCCESS;
```

```
}
// lower triangle of a:
//   11
//   12   17
//   13   18   22
//   14   19   23   26
//   15   20   24   27   29
//   16   21   25   28   30   31

// y after Ssymv:
//     82
//    108
//    126
//    138
//    146
//    152
//
//    [11   12   13   14   15   16] [1]      [1]    [ 82]
//    [12   17   18   19   20   21] [1]      [1]    [108]
//  1*[13   18   22   23   24   25]*[1] + 1*[1] = [126]
//    [14   19   23   26   27   28] [1]      [1]    [138]
//    [15   20   24   27   29   30] [1]      [1]    [146]
//    [16   21   25   28   30   31] [1]      [1]    [152]
```

3.3.9 `cublasSsyr` - symmetric rank-1 update

This function performs the symmetric rank-1 update

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

where A is an $n \times n$ symmetric matrix, x is a vector and α is a scalar. A is stored in column-major format in lower (`CUBLAS_FILL_MODE_LOWER`) or upper (`CUBLAS_FILL_MODE_UPPER`) mode.

```
// nvcc 021ssyr.c -lcublas
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include "cublas_v2.h"
#define IDX2C(i,j,ld) (((j)*(ld))+( i ))
#define n 6                 // number of rows and columns of a
int main(void){
  cudaError_t cudaStat;                  // cudaMalloc status
  cublasStatus_t stat;             // CUBLAS functions status
  cublasHandle_t handle;                     // CUBLAS context
  int i,j;                  // i - row index, j - column index
  float* a;                          // nxn matrix on the host
  float* x;                            // n-vector on the host
  a=(float*)malloc(n*n*sizeof(float));   // host memory for a
  x=(float*)malloc(n*sizeof(float));     // host memory for x
```

```
// define the lower triangle of an nxn symmetric matrix a
// in lower mode column by column
  int ind=11;                                  // a:
  for(j=0;j<n;j++){                            // 11
    for(i=0;i<n;i++){                          // 12,17
      if(i>=j){                                // 13,18,22
        a[IDX2C(i,j,n)]=(float)ind++;          // 14,19,23,26
      }                                        // 15,20,24,27,29
    }                                          // 16,21,25,28,30,31
  }
// print the lower triangle of a row by row
  printf("lower triangle of a:\n");
   for(i=0;i<n;i++){
     for(j=0;j<n;j++){
       if(i>=j)
       printf("%5.0f",a[IDX2C(i,j,n)]);
     }
    printf("\n");
   }
  for(i=0;i<n;i++){x[i]=1.0f;}                 // x={1,1,1,1,1,1}^T

// on the device
  float* d_a;                            // d_a - a on the device
  float* d_x;                            // d_x - x on the device
  cudaStat=cudaMalloc((void**)&d_a,n*n*sizeof(*a)); // device
                                         // memory alloc for a
  cudaStat=cudaMalloc((void**)&d_x,n*sizeof(*x));   // device
                                         // memory alloc for x
  stat = cublasCreate(&handle);  // initialize CUBLAS context
  stat = cublasSetMatrix(n,n,sizeof(*a),a,n,d_a,n);//a -> d_a
  stat = cublasSetVector(n,sizeof(*x),x,1,d_x,1); //cp x->d_x
  float al=1.0f;                                       // al=1.0
// symmetric rank-1 update of d_a:  d_a = al*d_x*d_x^T  + d_a
// d_a - nxn symmetric matrix; d_x - n-vector; al - scalar

  stat=cublasSsyr(handle,CUBLAS_FILL_MODE_LOWER,n,&al,d_x,1,
                                                    d_a,n);

  stat=cublasGetMatrix(n,n,sizeof(*a),d_a,n,a,n); //cp d_a->a
// print the lower triangle of the updated a after Ssyr
  printf("lower triangle of updated a after Ssyr :\n");
  for(i=0;i<n;i++){
    for(j=0;j<n;j++){
      if(i>=j)
      printf("%5.0f",a[IDX2C(i,j,n)]);
    }
    printf("\n");
  }
  cudaFree(d_a);                           // free device memory
  cudaFree(d_x);                           // free device memory
  cublasDestroy(handle);             // destroy CUBLAS context
  free(a);                                   // free host memory
```

```
  free(x);                                  // free host memory
return EXIT_SUCCESS;
}
// lower triangle of a:
//   11
//   12   17
//   13   18   22
//   14   19   23   26
//   15   20   24   27   29
//   16   21   25   28   30   31

// lower triangle of a after Ssyr://      [1]
//   12                            //      [1]
//   13   18                       //      [1]
//   14   19   23                  // a=1*[ ]*[1,1,1,1,1,1]+ a
//   15   20   24   27             //      [1]
//   16   21   25   28   30        //      [1]
//   17   22   26   29   31   32   //      [1]
```

3.3.10 cublasSsyr2 - symmetric rank-2 update

This function performs the symmetric rank-2 update

$$A = \alpha(xy^T + yx^T) + A,$$

where A is an $n \times n$ symmetric matrix, x, y are vectors and α is a scalar. A is stored in column-major format in lower (CUBLAS_FILL_MODE_LOWER) or upper (CUBLAS_FILL_MODE_UPPER) mode.

```
// nvcc 022ssyr2.c -lcublas
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include "cublas_v2.h"
#define IDX2C(i,j,ld) (((j)*(ld))+( i ))
#define n 6                  // number of rows and columns of a
int main(void){
  cudaError_t cudaStat;                    // cudaMalloc status
  cublasStatus_t stat;              // CUBLAS functions status
  cublasHandle_t handle;                      // CUBLAS context
  int i,j;                  // i - row index, j - column index
  float* a;                           // nxn matrix on the host
  float* x;                             // n-vector on the host
  float* y;                             // n-vector on the host
  a=(float*)malloc(n*n*sizeof(float));   // host memory for a
  x=(float*)malloc(n*sizeof(float));     // host memory for x
  y=(float*)malloc(n*sizeof(float));     // host memory for y
// define the lower triangle of an nxn symmetric matrix a
// in lower mode column by column
  int ind=11;                            // a:
```

```
  for(j=0;j<n;j++){                          // 11
    for(i=0;i<n;i++){                        // 12,17
      if(i>=j){                              // 13,18,22
        a[IDX2C(i,j,n)]=(float)ind++;        // 14,19,23,26
      }                                      // 15,20,24,27,29
    }                                        // 16,21,25,28,30,31
  }
// print the lower triangle of a row by row
  printf("lower triangle of a:\n");
   for(i=0;i<n;i++){
     for(j=0;j<n;j++){
       if(i>=j)
       printf("%5.0f",a[IDX2C(i,j,n)]);
     }
    printf("\n");
   }
  for(i=0;i<n;i++){x[i]=1.0f;y[i]=2.0;}  // x={1,1,1,1,1,1}^T
                                         // y={2,2,2,2,2,2}^T
// on the device
  float* d_a;                          // d_a - a on the device
  float* d_x;                          // d_x - x on the device
  float* d_y;                          // d_y - y on the device
  cudaStat=cudaMalloc((void**)&d_a,n*n*sizeof(*a)); // device
                                       // memory alloc for a
  cudaStat=cudaMalloc((void**)&d_x,n*sizeof(*x));   // device
                                       // memory alloc for x
  cudaStat=cudaMalloc((void**)&d_y,n*sizeof(*y));   // device
                                       // memory alloc for y
  stat = cublasCreate(&handle);  // initialize CUBLAS context
  stat = cublasSetMatrix(n,n,sizeof(*a),a,n,d_a,n);//a -> d_a
  stat = cublasSetVector(n,sizeof(*x),x,1,d_x,1); //cp x->d_x
  stat = cublasSetVector(n,sizeof(*y),y,1,d_y,1); //cp y->d_y
  float al=1.0f;                                       // al=1
// symmetric rank-2 update of d_a:
// d_a = al*(d_x*d_y^T + d_y*d_x^T) + d_a
// d_a - nxn symmetric matrix; d_x,d_y -n-vectors; al -scalar

  stat=cublasSsyr2(handle,CUBLAS_FILL_MODE_LOWER,n,&al,d_x,1,
                                                  d_y,1,d_a,n);

  stat=cublasGetMatrix(n,n,sizeof(*a),d_a,n,a,n); //cp d_a->a
// print the lower triangle of the updated a
  printf("lower triangle of a after Ssyr2 :\n");
  for(i=0;i<n;i++){
    for(j=0;j<n;j++){
      if(i>=j)
      printf("%5.0f",a[IDX2C(i,j,n)]);
    }
    printf("\n");
  }
  cudaFree(d_a);                       // free device memory
  cudaFree(d_x);                       // free device memory
```

```
  cudaFree(d_y);                          // free device memory
  cublasDestroy(handle);                // destroy CUBLAS context
  free(a);                                  // free host memory
  free(x);                                  // free host memory
  free(y);                                  // free host memory
  return EXIT_SUCCESS;
}

// lower triangle of a:
//   11
//   12   17
//   13   18   22
//   14   19   23   26
//   15   20   24   27   29
//   16   21   25   28   30   31

// lower triangle of a after Ssyr2 :
//   15
//   16   21
//   17   22   26
//   18   23   27   30
//   19   24   28   31   33
//   20   25   29   32   34   35

//[15 16 17 18 19 20]    [1]                [2]
//[16 21 22 23 24 25]    [1]                [2]
//[17 22 26 27 28 29]    [1]                [2]
//[                  ]=1*([ ]*[2,2,2,2,2,2]+[ ]*[1,1,1,1,1,1])+a
//[18 23 27 30 31 32]    [1]                [2]
//[19 24 28 31 33 34]    [1]                [2]
//[20 25 29 33 34 35]    [1]                [2]
```

3.3.11 cublasStbmv - triangular banded matrix-vector multiplication

This function performs the triangular banded matrix-vector multiplication

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

where A is a triangular banded matrix, x is a vector and $op(A)$ can be equal to A (CUBLAS_OP_N case), A^T (transposition) in CUBLAS_OP_T case or A^H (Hermitian transposition) in CUBLAS_OP_C case. The matrix A is stored in lower (CUBLAS_FILL_MODE_LOWER) or upper (CUBLAS_FILL_MODE_UPPER) mode. In the lower mode the main diagonal is stored in row 0, the first subdiagonal in row 1 and so on. If the diagonal of the matrix A has non-unit elements, then the parameter CUBLAS_DIAG_NON_UNIT should be used (in the opposite case - CUBLAS_DIAG_UNIT).

```
// nvcc 023stbmv.c -lcublas
```

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include "cublas_v2.h"
#define n 6                 // number of rows and columns of a
#define k 1                           // number of subdiagonals
int main(void){
  cudaError_t cudaStat;                    // cudaMalloc status
  cublasStatus_t stat;               // CUBLAS functions status
  cublasHandle_t handle;                      // CUBLAS context
  int i,j;                                // lower triangle of a:
  float *a; //nxn matrix a on the host  //  11
  float *x; //n-vector x  on the host   //  17,12
  a=(float*)malloc(n*n*sizeof(float)); //     18,13
  x=(float*)malloc(n*sizeof(float));   //        19,14
// main diagonal and subdiagonals       //           20,15
// of a in rows                         //              21,16
  int ind=11;
// main diagonal:   11,12,13,14,15,16 in row 0:
  for(i=0;i<n;i++) a[i*n]=(float)ind++;
// first subdiagonal: 17,18,19,20,21 in row 1:
  for(i=0;i<n-1;i++) a[i*n+1]=(float)ind++;
  for(i=0;i<n;i++){x[i]=1.0f;}           // x={1,1,1,1,1,1}^T

// on the device
  float* d_a;                          // d_a - a on the device
  float* d_x;                          // d_x - x on the device
  cudaStat=cudaMalloc((void**)&d_a,n*n*sizeof(*a)); // device
                                       // memory alloc for a
  cudaStat=cudaMalloc((void**)&d_x,n*sizeof(*x));   // device
                                       // memory alloc for x
  stat = cublasCreate(&handle);  // initialize CUBLAS context
  stat = cublasSetMatrix(n,n,sizeof(*a),a,n,d_a,n); // a->d_a
  stat = cublasSetVector(n,sizeof(*x),x,1,d_x,1);// cp x->d_x
// triangular banded matrix-vector multiplication:
// d_x = d_a*d_x;
// d_a - nxn lower triangular banded matrix; d_x - n-vector

  stat=cublasStbmv(handle,CUBLAS_FILL_MODE_LOWER,CUBLAS_OP_N,
                   CUBLAS_DIAG_NON_UNIT,n,k,d_a,n,d_x,1);

  stat=cublasGetVector(n,sizeof(*x),d_x,1,x,1); //copy d_x->x
  printf("x after Stbmv :\n");         // print x after Stbmv
  for(j=0;j<n;j++){
      printf("%7.0f",x[j]);
      printf("\n");
  }
  cudaFree(d_a);                         // free device memory
  cudaFree(d_x);                         // free device memory
  cublasDestroy(handle);             // destroy CUBLAS context
  free(a);                                 // free host memory
  free(x);                                 // free host memory
```

```
  return EXIT_SUCCESS;
}
// x after Stbmv :
//      11          //    [11     0     0     0     0     0] [1]
//      29          //    [17    12     0     0     0     0] [1]
//      31          // =  [ 0    18    13     0     0     0]*[1]
//      33          //    [ 0     0    19    14     0     0] [1]
//      35          //    [ 0     0     0    20    15     0] [1]
//      37          //    [ 0     0     0     0    21    16] [1]
```

3.3.12 cublasStbsv - solve the triangular banded linear system

This function solves the triangular banded linear system with a single right-hand-side

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

where A is a triangular banded matrix, x, b are vectors and $op(A)$ can be equal to A (CUBLAS_OP_N case), A^T (transposition) in CUBLAS_OP_T case, or A^H (Hermitian transposition) in CUBLAS_OP_C case. The matrix A is stored in lower (CUBLAS_FILL_MODE_LOWER) or upper (CUBLAS_FILL_MODE_UPPER) mode. In the lower mode the main diagonal is stored in row 0, the first subdiagonal in row 1 and so on. If the diagonal of the matrix A has non-unit elements, then the parameter CUBLAS_DIAG_NON_UNIT should be used (in the opposite case - CUBLAS_DIAG_UNIT).

```
// nvcc 024stbsv.c -lcublas
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include "cublas_v2.h"
#define n 6                 // number of rows and columns of a
#define k 1                          // number of subdiagonals
int main(void){
  cudaError_t cudaStat;                    // cudaMalloc status
  cublasStatus_t stat;               // CUBLAS functions status
  cublasHandle_t handle;                      // CUBLAS context
  int i,j;                                // lower triangle of a:
  float *a; //nxn matrix a on the host    // 11
  float *x; //n-vector x on the host      // 17,12
  a=(float*)malloc(n*n*sizeof(float));    //    18,13
// memory allocation for a on the host    //       19,14
  x=(float*)malloc(n*sizeof(float));      //          20,15
// memory allocation for x on the host    //             21,16
//main diagonal and subdiagonals of a in rows:
  int ind=11;
// main diagonal:  11,12,13,14,15,16  in row 0
  for(i=0;i<n;i++) a[i*n]=(float)ind++;
// first subdiagonal: 17,18,19,20,21 in row 1
  for(i=0;i<n-1;i++) a[i*n+1]=(float)ind++;
  for(i=0;i<n;i++){x[i]=1.0f;}            // x={1,1,1,1,1,1}^T
```

```
// on the device
  float* d_a;                          // d_a - a on the device
  float* d_x;                          // d_x - x on the device
  cudaStat=cudaMalloc((void**)&d_a,n*n*sizeof(*a)); // device
                                       // memory alloc for a
  cudaStat=cudaMalloc((void**)&d_x,n*sizeof(*x));   // device
                                       // memory alloc for x
  stat = cublasCreate(&handle);  // initialize CUBLAS context
  stat = cublasSetMatrix(n,n,sizeof(*a),a,n,d_a,n);//a -> d_a
  stat = cublasSetVector(n,sizeof(*x),x,1,d_x,1); //cp x->d_x
// solve a triangular banded linear system:   d_a*d_x=d_x;
// the solution d_x  overwrites the right hand side d_x;
// d_a - nxn banded lower triangular matrix; d_x - n-vector

  stat=cublasStbsv(handle,CUBLAS_FILL_MODE_LOWER,CUBLAS_OP_N,
                       CUBLAS_DIAG_NON_UNIT,n,k,d_a,n,d_x,1);

  stat=cublasGetVector(n,sizeof(*x),d_x,1,x,1); //copy d_x->x
// print the solution
  printf("solution :\n");              // print x after Stbsv
  for(j=0;j<n;j++){
      printf("%9.6f",x[j]);
      printf("\n");
  }
  cudaFree(d_a);                        // free device memory
  cudaFree(d_x);                        // free device memory
  cublasDestroy(handle);              // destroy CUBLAS context
  free(a);                                // free host memory
  free(x);                                // free host memory
  return EXIT_SUCCESS;
}
// solution :
// 0.090909 // [11    0    0    0    0    0] [ 0.090909]  [1]
//-0.045455 // [17   12    0    0    0    0] [-0.045455]  [1]
// 0.139860 // [ 0   18   13    0    0    0] [ 0.139860] =[1]
//-0.118382 // [ 0    0   19   14    0    0] [-0.118382]  [1]
// 0.224509 // [ 0    0    0   20   15    0] [ 0.224509]  [1]
//-0.232168 // [ 0    0    0    0   21   16] [-0.232168]  [1]
```

3.3.13 cublasStpmv - triangular packed matrix-vector multiplication

This function performs the triangular packed matrix-vector multiplication

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

where A is a triangular packed matrix, x is a vector and $op(A)$ can be equal to A (CUBLAS_OP_N case), A^T (CUBLAS_OP_T case - transposition) or A^H (CUBLAS_OP_C case - conjugate transposition). A can be stored in lower (CUBLAS_FILL_MODE_LOWER) or upper (CUBLAS_FILL_MODE_UPPER) mode. In

lower mode the elements of the lower triangular part of A are packed together column by column without gaps. If the diagonal of the matrix A has non-unit elements, then the parameter `CUBLAS_DIAG_NON_UNIT` should be used (in the opposite case - `CUBLAS_DIAG_UNIT`).

```
// nvcc 025stpmv.c -lcublas
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include "cublas_v2.h"
#define n 6                 // number of rows and columns of a
int main(void){
  cudaError_t cudaStat;                   // cudaMalloc status
  cublasStatus_t stat;             // CUBLAS functions status
  cublasHandle_t handle;                     // CUBLAS context
  int i,j;
  float* a;     // lower triangle of a nxn matrix on the host
  float* x;                           //  n-vector on the host
  a=(float*)malloc(n*(n+1)/2*sizeof(float));   // host memory
                                               // alloc for a
  x=(float*)malloc(n*sizeof(float));//host memory alloc for x
//define a triangular matrix a in packed format column
// by column  without gaps                    //a:
  for(i=0;i<n*(n+1)/2;i++)                    //11
    a[i]=(float)(11+i);                       //12,17
  for(i=0;i<n;i++){x[i]=1.0f;}                //13,18,22
// x={1,1,1,1,1,1}^T                         //14,19,23,2
// on the device                             //15,20,24,27,29
  float* d_a;  // d_a - a on the device      //16,21,25,28,30,31
  float* d_x;  // d_x - x on the device
  cudaStat=cudaMalloc((void**)&d_a,n*(n+1)/2*sizeof(*a));
                                  // device memory alloc for a
  cudaStat=cudaMalloc((void**)&d_x,n*sizeof(*x));   // device
                                        // memory alloc for x
  stat = cublasCreate(&handle);  // initialize CUBLAS context
  stat = cublasSetVector(n*(n+1)/2,sizeof(*a),a,1,d_a,1);
                                              // copy a -> d_a
  stat = cublasSetVector(n,sizeof(*x),x,1,d_x,1);// cp x->d_x
// triangular packed matrix-vector multiplication:
// d_x = d_a*d_x;  d_a -nxn lower triangular matrix
// in packed format;  d_x -n-vector

  stat=cublasStpmv(handle,CUBLAS_FILL_MODE_LOWER,CUBLAS_OP_N,
                          CUBLAS_DIAG_NON_UNIT,n,d_a,d_x,1);

  stat=cublasGetVector(n,sizeof(*x),d_x,1,x,1); //copy d_x->x
  printf("x after Stpmv :\n");          // print x after Stpmv
  for(j=0;j<n;j++){
      printf("%7.0f",x[j]);
      printf("\n");
  }
```

```
  cudaFree(d_a);                          // free device memory
  cudaFree(d_x);                          // free device memory
  cublasDestroy(handle);                // destroy CUBLAS context
  free(a);                                  // free host memory
  free(x);                                  // free host memory
  return EXIT_SUCCESS;
}
// x after Stpmv :
//     11       //         [11    0    0    0    0    0] [1]
//     29       //         [12   17    0    0    0    0] [1]
//     53       //       = [13   18   22    0    0    0]*[1]
//     82       //         [14   19   23   26    0    0] [1]
//    115       //         [15   20   24   27   29    0] [1]
//    151       //         [16   21   25   28   30   31] [1]
```

3.3.14 cublasStpsv - solve the packed triangular linear system

This function solves the packed triangular linear system

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

where A is a triangular packed $n \times n$ matrix, x, b are n-vectors and $op(A)$ can be equal to A (CUBLAS_OP_N case), A^T (- transposition)) in CUBLAS_OP_T case or A^H (conjugate transposition) in CUBLAS_OP_C case. A can be stored in lower (CUBLAS_FILL_MODE_LOWER) or upper (CUBLAS_FILL_MODE_UPPER) mode. In lower mode the elements of the lower triangular part of A are packed together column by column without gaps. If the diagonal of the matrix A has non-unit elements, then the parameter CUBLAS_DIAG_NON_UNIT should be used (in the opposite case - CUBLAS_DIAG_UNIT).

```
// nvcc 026stpsv.c -lcublas
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include "cublas_v2.h"
#define n 6                 // number of rows and columns of a
int main(void){
  cudaError_t cudaStat;                  // cudaMalloc status
  cublasStatus_t stat;              // CUBLAS functions status
  cublasHandle_t handle;                      // CUBLAS context
  int i,j;                       // i-row index, j-column index
  float* a;                         // nxn matrix a on the host
  float* x;                          // n-vector x on the host
  a=(float*)malloc(n*(n+1)/2*sizeof(float));   // host memory
                                               // alloc for a
  x=(float*)malloc(n*sizeof(float));//host memory alloc for x
// define a triangular a in packed format
// column by column without gaps          //a:
  for(i=0;i<n*(n+1)/2;i++)               //11
```

```
    a[i]=(float)(11+i);                          //12,17
  for(i=0;i<n;i++){x[i]=1.0f;}                   //13,18,22
// x={1,1,1,1,1,1}^T                           //14,19,23,26
// on the device                               //15,20,24,27,29
  float* d_a;  // d_a - a on the device         //16,21,25,28,30,31
  float* d_x;  // d_x - x on the device
  cudaStat=cudaMalloc((void**)&d_a,n*(n+1)/2*sizeof(*a));
                                  // device memory alloc for a
  cudaStat=cudaMalloc((void**)&d_x,n*sizeof(*x));   // device
                                          // memory alloc for x
  stat = cublasCreate(&handle);  // initialize CUBLAS context
  stat = cublasSetVector(n*(n+1)/2,sizeof(*a),a,1,d_a,1);
                                             // copy a -> d_a
  stat = cublasSetVector(n,sizeof(*x),x,1,d_x,1);// cp x->d_x
// solve the packed triangular linear system:  d_a*d_x=d_x,
// the solution d_x overwrites the right hand side d_x
// d_a -nxn lower triang. matrix in packed form; d_x -n-vect.

  stat=cublasStpsv(handle,CUBLAS_FILL_MODE_LOWER,CUBLAS_OP_N,
                          CUBLAS_DIAG_NON_UNIT,n,d_a,d_x,1);

  stat=cublasGetVector(n,sizeof(*x),d_x,1,x,1); //copy d_x->x
  printf("solution :\n");               // print x after Stpsv
  for(j=0;j<n;j++){
      printf("%9.6f",x[j]);
      printf("\n");
  }
  cudaFree(d_a);                           // free device memory
  cudaFree(d_x);                           // free device memory
  cublasDestroy(handle);                // destroy CUBLAS context
  free(a);                                   // free host memory
  free(x);                                   // free host memory
  return EXIT_SUCCESS;
}
// solution :
// 0.090909  //      [11   0   0   0   0   0] [ 0.090909] [1]
//-0.005348  //      [12  17   0   0   0   0] [-0.005348] [1]
//-0.003889  //      [13  18  22   0   0   0]*[-0.003889]=[1]
//-0.003141  //      [14  19  23  26   0   0] [-0.003141] [1]
//-0.002708  //      [15  20  24  27  29   0] [-0.002708] [1]
//-0.002446  //      [16  21  25  28  30  31] [-0.002446] [1]
```

3.3.15 `cublasStrmv` - triangular matrix-vector multiplication

This function performs the triangular matrix-vector multiplication

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

where A is a triangular $n \times n$ matrix, x is an n-vector and $op(A)$ can be equal to A (`CUBLAS_OP_N` case), A^T (transposition) in `CUBLAS_OP_T` case or A^H (conjugate transposition) in `CUBLAS_OP_C` case. The matrix A can be stored

in lower (CUBLAS_FILL_MODE_LOWER) or upper (CUBLAS_FILL_MODE_UPPER) mode. If the diagonal of the matrix A has non-unit elements, then the parameter CUBLAS_DIAG_NON_UNIT should be used (in the opposite case - CUBLAS_DIAG_UNIT).

```
// nvcc 027strmv.c -lcublas
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include "cublas_v2.h"
#define IDX2C(i,j,ld) (((j)*(ld))+( i ))
#define n 6                  // number of rows and columns of a
int main(void){
  cudaError_t cudaStat;                    // cudaMalloc status
  cublasStatus_t stat;              // CUBLAS functions status
  cublasHandle_t handle;                      // CUBLAS context
  int i,j;                         // i-row index, j-column index
  float* a;                           // nxn matrix a on the host
  float* x;                            // n-vector x on the host
  a=(float*)malloc(n*n*sizeof(*a)); //host memory alloc for a
  x=(float*)malloc(n*sizeof(*x));   //host memory alloc for x
// define an nxn triangular matrix a in lower mode
// column by column
  int ind=11;                                  // a:
  for(j=0;j<n;j++){                            // 11
    for(i=0;i<n;i++){                          // 12,17
      if(i>=j){                                // 13,18,22
        a[IDX2C(i,j,n)]=(float)ind++;          // 14,19,23,26
      }                                        // 15,20,24,27,29
    }                                          // 16,21,25,28,30,31
  }
  for(i=0;i<n;i++) x[i]=1.0f;                  // x={1,1,1,1,1,1}^T
// on the device
  float* d_a;                            // d_a - a on the device
  float* d_x;                            // d_x - x on the device
  cudaStat=cudaMalloc((void**)&d_a,n*n*sizeof(*a)); // device
                                         // memory alloc for a
  cudaStat=cudaMalloc((void**)&d_x,n*sizeof(*x));    //device
                                        //  memory alloc for x
  stat = cublasCreate(&handle);  // initialize CUBLAS context
  stat = cublasSetMatrix(n,n,sizeof(*a),a,n,d_a,n); // a->d_a
  stat = cublasSetVector(n,sizeof(*x),x,1,d_x,1); //cp x->d_x
// triangular matrix-vector multiplication:  d_x = d_a*d_x
// d_a - triangular nxn matrix in lower mode; d_x - n-vector

  stat=cublasStrmv(handle,CUBLAS_FILL_MODE_LOWER,CUBLAS_OP_N,
                   CUBLAS_DIAG_NON_UNIT,n,d_a,n,d_x,1);

  stat=cublasGetVector(n,sizeof(*x),d_x,1,x,1); //copy d_x->x
  printf("multiplication result :\n"); // print x after Strmv
  for(j=0;j<n;j++){
```

```
        printf("%7.0f",x[j]);
        printf("\n");
    }
    cudaFree(d_a);                          // free device memory
    cudaFree(d_x);                          // free device memory
    cublasDestroy(handle);                  // destroy CUBLAS context
    free(a);                                  // free host memory
    free(x);                                  // free host memory
    return EXIT_SUCCESS;
}
// multiplication result :
//      11       //          [11    0    0    0    0    0] [1]
//      29       //          [12   17    0    0    0    0] [1]
//      53       //        = [13   18   22    0    0    0]*[1]
//      82       //          [14   19   23   26    0    0] [1]
//     115       //          [15   20   24   27   29    0] [1]
//     151       //          [16   21   25   28   30   31] [1]
```

3.3.16 cublasStrsv - solve the triangular linear system

This function solves the triangular linear system

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

where A is a triangular $n \times n$ matrix, x, b are n-vectors and $op(A)$ can be equal to A (CUBLAS_OP_N case), A^T (transposition) in CUBLAS_OP_T case or A^H (conjugate transposition) in CUBLAS_OP_C case. A can be stored in lower (CUBLAS_FILL_MODE_LOWER) or upper (CUBLAS_FILL_MODE_UPPER) mode. If the diagonal of the matrix A has non-unit elements, then the parameter CUBLAS_DIAG_NON_UNIT should be used (in the opposite case - CUBLAS_DIAG_UNIT).

```
// nvcc 028strsv.c -lcublas
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include "cublas_v2.h"
#define IDX2C(i,j,ld) (((j)*(ld))+( i ))
#define n 6                  // number of rows and columns of a
int main(void){
    cudaError_t cudaStat;                    // cudaMalloc status
    cublasStatus_t stat;                // CUBLAS functions status
    cublasHandle_t handle;                        // CUBLAS context
    int i,j;                          // i-row index, j-column index
    float* a;                           // nxn matrix a on the host
    float* x;                             // n-vector x on the host
    a=(float*)malloc(n*n*sizeof(*a)); //host memory alloc for a
    x=(float*)malloc(n*sizeof(*x));   //host memory alloc for x
// define an nxn triangular matrix a in lower mode
```

```
// column by column
  int ind=11;                                  // a:
  for(j=0;j<n;j++){                            // 11
    for(i=0;i<n;i++){                          // 12,17
      if(i>=j){                                // 13,18,22
        a[IDX2C(i,j,n)]=(float)ind++;          // 14,19,23,26
      }                                        // 15,20,24,27,29
    }                                          // 16,21,25,28,30,31
  }
  for(i=0;i<n;i++) x[i]=1.0f;                  // x={1,1,1,1,1,1}^T
// on the device
  float* d_a;                            // d_a - a on the device
  float* d_x;                            // d_x - x on the device
  cudaStat=cudaMalloc((void**)&d_a,n*n*sizeof(*a)); // device
                                         // memory alloc for a
  cudaStat=cudaMalloc((void**)&d_x,n*sizeof(*x));    //device
                                         // memory alloc for x
  stat = cublasCreate(&handle);  // initialize CUBLAS context
  stat = cublasSetMatrix(n,n,sizeof(*a),a,n,d_a,n); // a->d_a
  stat = cublasSetVector(n,sizeof(*x),x,1,d_x,1); //cp x->d_x
// solve the triangular linear system:  d_a*x=d_x,
// the solution x overwrites the right hand side d_x,
// d_a - nxn triangular matrix in lower mode; d_x - n-vector

  stat=cublasStrsv(handle,CUBLAS_FILL_MODE_LOWER,CUBLAS_OP_N,
                   CUBLAS_DIAG_NON_UNIT,n,d_a,n,d_x,1);

  stat=cublasGetVector(n,sizeof(*x),d_x,1,x,1); //copy x->d_x
  printf("solution :\n");              // print x after Strsv
  for(j=0;j<n;j++){
      printf("%9.6f",x[j]);
      printf("\n");
  }
  cudaFree(d_a);                          // free device memory
  cudaFree(d_x);                          // free device memory
  cublasDestroy(handle);             // destroy CUBLAS context
  free(a);                                  // free host memory
  free(x);                                  // free host memory
  return EXIT_SUCCESS;
}
// solution :

// 0.090909  //  [11   0   0   0   0   0] [ 0.090909] [1]
//-0.005348  //  [12  17   0   0   0   0] [-0.005348] [1]
//-0.003889  //  [13  18  22   0   0   0]*[-0.003889]=[1]
//-0.003141  //  [14  19  23  26   0   0] [-0.003141] [1]
//-0.002708  //  [15  20  24  27  29   0] [-0.002708] [1]
//-0.002446  //  [16  21  25  28  30  31] [-0.002446] [1]
```

3.3.17 cublasChemv - Hermitian matrix-vector multiplication

This function performs the Hermitian matrix-vector multiplication

$$y = \alpha Ax + \beta y,$$

where A is an $n \times n$ complex Hermitian matrix, x, y are complex n-vectors and α, β are complex scalars. A can be stored in lower (CUBLAS_FILL_MODE_LOWER) or upper (CUBLAS_FILL_MODE_UPPER) mode.

```
// nvcc 029Chemv.c -lcublas
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include "cublas_v2.h"
#include "cuComplex.h"
#define IDX2C(i,j,ld) (((j)*(ld))+( i ))
#define n 6                 // number of rows and columns of a
int main(void){
  cudaError_t cudaStat;                   // cudaMalloc status
  cublasStatus_t stat;              // CUBLAS functions status
  cublasHandle_t handle;                     // CUBLAS context
  int i,j;                       // i-row index, j-column index
  cuComplex *a;               // complex nxn matrix on the host
  cuComplex *x;                 // complex n-vector on the host
  cuComplex *y;                 // complex n-vector on the host
  a=(cuComplex*)malloc(n*n*sizeof(cuComplex));  //host memory
                                                //alloc for a
  x=(cuComplex*)malloc(n*sizeof(cuComplex));    //host memory
                                                //alloc for x
  y=(cuComplex*)malloc(n*sizeof(cuComplex));    //host memory
                                                //alloc for y
// define the lower triangle of an nxn Hermitian matrix a in
// lower mode  column by column
  int ind=11;                               // c:
  for(j=0;j<n;j++){                         // 11
    for(i=0;i<n;i++){                       // 12,17
      if(i>=j){                             // 13,18,22
        a[IDX2C(i,j,n)].x=(float)ind++;     // 14,19,23,26
        a[IDX2C(i,j,n)].y=0.0f;             // 15,20,24,27,29
      }                                     // 16,21,25,28,30,31
    }
  }
// print the lower triangle of a row by row
  printf("lower triangle of a:\n");
  for(i=0;i<n;i++){
   for(j=0;j<n;j++){
    if(i>=j)
    printf("%5.0f+%1.0f*I",a[IDX2C(i,j,n)].x,
                           a[IDX2C(i,j,n)].y);
   }
```

```
    printf("\n");
  }
  for(i=0;i<n;i++){x[i].x=1.0f;y[i].x=1.0;}
                      //x={1,1,1,1,1,1}^T  ;y={1,1,1,1,1,1}^T
// on the device
  cuComplex* d_a;                     // d_a - a on the device
  cuComplex* d_x;                     // d_x - x on the device
  cuComplex* d_y;                     // d_y - y on the device
  cudaStat=cudaMalloc((void**)&d_a,n*n*sizeof(cuComplex));
                                      //device memory alloc for a
  cudaStat=cudaMalloc((void**)&d_x,n*sizeof(cuComplex));
                                      //device memory alloc for x
  cudaStat=cudaMalloc((void**)&d_y,n*sizeof(cuComplex));
                                      //device memory alloc for y
  stat = cublasCreate(&handle);  // initialize CUBLAS context
  stat = cublasSetMatrix(n,n,sizeof(*a),a,n,d_a,n);
                                                // copy a -> d_a
  stat = cublasSetVector(n,sizeof(*x),x,1,d_x,1);//cp x->d_x
  stat = cublasSetVector(n,sizeof(*y),y,1,d_y,1);//cp y->d_y
  cuComplex al={1.0f,0.0f};                              // al=1
  cuComplex bet={1.0f,0.0f};                            // bet=1
// Hermitian matrix-vector multiplication:
// d_y=al*d_a*d_x + bet*d_y
// d_a -nxn Hermitian matrix; d_x,d_y -n-vectors;
// al,bet -scalars

  stat=cublasChemv(handle,CUBLAS_FILL_MODE_LOWER,n,&al,d_a,n,
                                              d_x,1,&bet,d_y,1);

  stat=cublasGetVector(n,sizeof(*y),d_y,1,y,1);//copy d_y->y
  printf("y after Chemv:\n");          // print y after Chemv
  for(j=0;j<n;j++){
      printf("%4.0f+%1.0f*I",y[j].x,y[j].y);
      printf("\n");
  }
  cudaFree(d_a);                           // free device memory
  cudaFree(d_x);                           // free device memory
  cudaFree(d_y);                           // free device memory
  cublasDestroy(handle);              // destroy CUBLAS context
  free(a);                                   // free host memory
  free(y);                                   // free host memory
  return EXIT_SUCCESS;
}
// lower triangle of a:
//   11+0*I
//   12+0*I   17+0*I
//   13+0*I   18+0*I   22+0*I
//   14+0*I   19+0*I   23+0*I   26+0*I
//   15+0*I   20+0*I   24+0*I   27+0*I   29+0*I
//   16+0*I   21+0*I   25+0*I   28+0*I   30+0*I   31+0*I

// y after Chemv:
```

```
//  82+0*I
// 108+0*I
// 126+0*I
// 138+0*I
// 146+0*I
// 152+0*I
//
//    [11   12   13   14   15   16] [1]     [1]   [ 82]
//    [12   17   18   19   20   21] [1]     [1]   [108]
//  1*[13   18   22   23   24   25]*[1] + 1*[1] = [126]
//    [14   19   23   26   27   28] [1]     [1]   [138]
//    [15   20   24   27   29   30] [1]     [1]   [146]
//    [16   21   25   28   30   31] [1]     [1]   [152]
```

3.3.18 `cublasChbmv` - Hermitian banded matrix-vector multiplication

This function performs the Hermitian banded matrix-vector multiplication

$$y = \alpha Ax + \beta y,$$

where A is an $n \times n$ complex Hermitian banded matrix with k subdiagonals and superdiagonals, x, y are complex n-vectors and α, β are complex scalars. A can be stored in lower (`CUBLAS_FILL_MODE_LOWER`) or upper (`CUBLAS_FILL_MODE_UPPER`) mode. If A is stored in lower mode, then the main diagonal is stored in row 0, the first subdiagonal in row 1, the second subdiagonal in row 2, etc.

```
// nvcc 030Chbmv.c -lcublas
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include "cublas_v2.h"
#include "cuComplex.h"
#define n 6                 // number of rows and columns of a
#define k 1      // number of subdiagonals and superdiagonals
int main(void){
  cudaError_t cudaStat;                  // cudaMalloc status
  cublasStatus_t stat;             // CUBLAS functions status
  cublasHandle_t handle;                     // CUBLAS context
  int i,j;                       // i-row index, j-column index
                                    // lower triangle of a:
  cuComplex *a;//nxn matrix a on the host //11
  cuComplex *x; //n-vector x on the host  //17,12
  cuComplex *y; //n-vector y on the host  //   18,13
  a=(cuComplex*)malloc(n*n*sizeof(*a));   //      19,14
// host memory alloc for a                //         20,15
  x=(cuComplex*)malloc(n*sizeof(*x));     //            21,16
// host memory alloc for x
```

```
  y=(cuComplex*)malloc(n*sizeof(*y));
// host memory alloc for y
// main diagonal and subdiagonals of a in rows
  int ind=11;
  for(i=0;i<n;i++) a[i*n].x=(float)ind++;
// main diagonal:  11,12,13,14,15,16 in row 0
  for(i=0;i<n-1;i++) a[i*n+1].x=(float)ind++;
// first subdiagonal: 17,18,19,20,21 in row 1
  for(i=0;i<n;i++){x[i].x=1.0f;y[i].x=0.0f;}
                      //x={1,1,1,1,1,1}^T;  y={0,0,0,0,0,0}^T
// on the device
  cuComplex* d_a;                     // d_a - a on the device
  cuComplex* d_x;                     // d_x - x on the device
  cuComplex* d_y;                     // d_y - y on the device
  cudaStat=cudaMalloc((void**)&d_a,n*n*sizeof(*a));  //device
                                      // memory alloc for a
  cudaStat=cudaMalloc((void**)&d_x,n*sizeof(*x));    //device
                                      // memory alloc for x
  cudaStat=cudaMalloc((void**)&d_y,n*sizeof(*y));    //device
                                      // memory alloc for y
  stat = cublasCreate(&handle);  // initialize CUBLAS context
// copy matrix and vectors from host to device
  stat = cublasSetMatrix(n,n,sizeof(*a),a,n,d_a,n);// a-> d_a
  stat = cublasSetVector(n,sizeof(*x),x,1,d_x,1);  // x-> d_x
  stat = cublasSetVector(n,sizeof(*y),y,1,d_y,1);  // y-> d_y
  cuComplex al={1.0f,0.0f};                          // al=1
  cuComplex bet={1.0f,0.0f};                        // bet=1
// Hermitian banded matrix-vector multiplication:
// d_y = al*d_a*d_x + bet*d_y
// d_a - complex Hermitian banded nxn matrix;
// d_x,d_y -complex  n-vectors; al,bet - complex scalars

  stat=cublasChbmv(handle,CUBLAS_FILL_MODE_LOWER,n,k,&al,d_a,n,
                                                 d_x,1,&bet,d_y,1);
  stat=cublasGetVector(n,sizeof(*y),d_y,1,y,1); //copy d_y->y
  printf("y after Chbmv:\n");          // print y after Chbmv
  for(j=0;j<n;j++){
    printf("%3.0f+%1.0f*I",y[j].x,y[j].y);
    printf("\n");
  }
  cudaFree(d_a);                        // free device memory
  cudaFree(d_x);                        // free device memory
  cudaFree(d_y);                        // free device memory
  cublasDestroy(handle);             // destroy CUBLAS context
  free(a);                                // free host memory
  free(x);                                // free host memory
  free(y);                                // free host memory
  return EXIT_SUCCESS;
}
// y after Chbmv:
// 28+0*I                  //   [11 17            ] [1]    [28]
// 47+0*I                  //   [17 12 18         ] [1]    [47]
```

```
// 50+0*I                    //   [   18 13 19      ] [1] = [50]
// 53+0*I                    //   [      19 14 20   ] [1]   [53]
// 56+0*I                    //   [         20 15 21] [1]   [56]
// 37+0*I                    //   [            21 16] [1]   [37]
```

3.3.19 cublasChpmv - Hermitian packed matrix-vector multiplication

This function performs the Hermitian packed matrix-vector multiplication

$$y = \alpha Ax + \beta y,$$

where A is an $n \times n$ complex Hermitian packed matrix, x, y are complex n-vectors and α, β are complex scalars. A can be stored in lower (CUBLAS_FILL_
MODE_LOWER) or upper (CUBLAS_FILL_MODE_UPPER) mode. If A is stored in lower mode, then the elements of the lower triangular part of A are packed together column by column without gaps.

```
// nvcc 031Chpmv.c -lcublas
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include "cublas_v2.h"
#define n 6                 // number of rows and columns of a
int main(void){
  cudaError_t cudaStat;                   // cudaMalloc status
  cublasStatus_t stat;              // CUBLAS functions status
  cublasHandle_t handle;                     // CUBLAS context
  int i,j,l,m;                  // i-row index, j-column index
// data preparation on the host
  cuComplex *a;                 // lower triangle of a complex
                                //  nxn matrix on the host
  cuComplex *x;              // complex n-vector x on the host
  cuComplex *y;              // complex n-vector y on the host
  a=(cuComplex*)malloc(n*(n+1)/2*sizeof(cuComplex));  // host
                                          // memory alloc for a
  x=(cuComplex*)malloc(n*sizeof(cuComplex));   // host memory
                                               // alloc for x
  y=(cuComplex*)malloc(n*sizeof(cuComplex));   // host memory
                                               // alloc for y
// define the lower triangle of a Hermitian matrix a:
// in packed format, column by column    // 11
// without gaps                          // 12,17
  for(i=0;i<n*(n+1)/2;i++)              // 13,18,22
    a[i].x=(float)(11+i);               // 14,19,23,26
// print the upp triang of a  row by row // 15,20,24,27,29
  printf("upper triangle of a:\n");     // 16,21,25,28,30,31
  l=n;j=0;m=0;
```

```
  while(l>0){                                  // print the upper
    for(i=0;i<m;i++) printf("       ");       // triangle of a
    for(i=j;i<j+l;i++) printf("%3.0f+%1.0f*I",a[i].x,a[i].y);
    printf("\n");
    m++;j=j+l;l--;
  }
  for(i=0;i<n;i++){x[i].x=1.0f;y[i].x=0.0f;}
                     //x={1,1,1,1,1,1}^T;   y={0,0,0,0,0,0}^T
// on the device
  cuComplex* d_a;                     // d_a - a on the device
  cuComplex* d_x;                     // d_x - x on the device
  cuComplex* d_y;                     // d_y - y on the device
  cudaStat=cudaMalloc((void**)&d_a,n*(n+1)/2*sizeof(*a));
                                    //device memory alloc for a
  cudaStat=cudaMalloc((void**)&d_x,n*sizeof(cuComplex));
                                    //device memory alloc for x
  cudaStat=cudaMalloc((void**)&d_y,n*sizeof(cuComplex));
                                   // device memory alloc for y
  stat = cublasCreate(&handle);  // initialize CUBLAS context
// copy matrix and vectors from the host to the device
  stat = cublasSetVector(n*(n+1)/2,sizeof(*a),a,1,d_a,1);
                                                //copy a-> d_a
  stat = cublasSetVector(n,sizeof(cuComplex),x,1,d_x,1);
                                                //copy x-> d_x
  stat = cublasSetVector(n,sizeof(cuComplex),y,1,d_y,1);
                                                //copy y-> d_y
  cuComplex al={1.0f,0.0f};                          //  al=1
  cuComplex bet={1.0f,0.0f};                         // bet=1
// Hermitian packed matrix-vector multiplication:
// d_y = al*d_a*d_x + bet*d_y;  d_a - nxn Hermitian  matrix
// in packed format; d_x,d_y - complex n-vectors;
//  al,bet - complex scalars

  stat=cublasChpmv(handle,CUBLAS_FILL_MODE_LOWER,n,&al,d_a,d_x,1,
                                                    &bet,d_y,1);
  stat=cublasGetVector(n,sizeof(cuComplex),d_y,1,y,1);
                                                // copy d_y->y
  printf("y after Chpmv :\n");          // print y after Chpmv
  for(j=0;j<n;j++){
      printf("%3.0f+%1.0f*I",y[j].x,y[j].y);
      printf("\n");
  }
  cudaFree(d_a);                          // free device memory
  cudaFree(d_x);                          // free device memory
  cudaFree(d_y);                          // free device memory
  cublasDestroy(handle);               // destroy CUBLAS context
  free(a);                                  // free host memory
  free(x);                                  // free host memory
  free(y);                                  // free host memory
  return EXIT_SUCCESS;
}
// upper triangle of a:
```

```
// 11+0*I 12+0*I 13+0*I 14+0*I 15+0*I 16+0*I
//        17+0*I 18+0*I 19+0*I 20+0*I 21+0*I
//               22+0*I 23+0*I 24+0*I 25+0*I
//                      26+0*I 27+0*I 28+0*I
//                             29+0*I 30+0*I
//                                    31+0*I

// y after Chpmv :
//  81+0*I //    [11  12  13  14  15  16] [1]     [0]   [ 81]
// 107+0*I //    [12  17  18  19  20  21] [1]     [0]   [107]
// 125+0*I //  1*[13  18  22  23  24  25]*[1] + 1*[0] = [125]
// 137+0*I //    [14  19  23  26  27  28] [1]     [0]   [137]
// 145+0*I //    [15  20  24  27  29  30] [1]     [0]   [145]
// 151+0*I //    [16  21  25  28  30  31] [1]     [0]   [151]
```

3.3.20 `cublasCher` - Hermitian rank-1 update

This function performs the Hermitian rank-1 update

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

where A is an $n \times n$ Hermitian complex matrix, x is a complex n-vector and α is a scalar. A is stored in column-major format. A can be stored in lower (`CUBLAS_FILL_MODE_LOWER`) or upper (`CUBLAS_FILL_MODE_UPPER`) mode.

```
// nvcc 032cher.c -lcublas
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include "cublas_v2.h"
#define IDX2C(i,j,ld) (((j)*(ld))+( i ))
#define n 6                 // number of rows and columns of a
int main(void){
  cudaError_t cudaStat;                   // cudaMalloc status
  cublasStatus_t stat;              // CUBLAS functions status
  cublasHandle_t handle;                      // CUBLAS context
  int i,j;                       // i-row index, j-column index
// data preparation on the host
  cuComplex *a;              //nxn complex matrix a on the host
  cuComplex *x;               //complex  n-vector x on the host
  a=(cuComplex*)malloc(n*n*sizeof(cuComplex)); // host memory
                                                 // alloc for a
  x=(cuComplex*)malloc(n*sizeof(cuComplex));   // host memory
                                                 // alloc for x
// define the lower triangle of an nxn Hermitian matrix a
// column by column
  int ind=11;                                 // a:
  for(j=0;j<n;j++){                           // 11
    for(i=0;i<n;i++){                         // 12,17
      if(i>=j){                               // 13,18,22
```

```
        a[IDX2C(i,j,n)].x=(float)ind++;  // 14,19,23,26
        a[IDX2C(i,j,n)].y=0.0f;          // 15,20,24,27,29
      }                                  // 16,21,25,28,30,31
    }
  }
// print the lower triangle of a row by row
  printf("lower triangle of a:\n");
   for(i=0;i<n;i++){
    for(j=0;j<n;j++){
     if(i>=j)
     printf("%5.0f+%1.0f*I",a[IDX2C(i,j,n)].x,a[IDX2C(i,j,n)].y);
    }
   printf("\n");
   }
  for(i=0;i<n;i++) x[i].x=1.0f;          // x={1,1,1,1,1,1}^T

// on the device
  cuComplex* d_a;                    // d_a - a on the device
  cuComplex* d_x;                    // d_x - x on the device
  cudaStat=cudaMalloc((void**)&d_a,n*n*sizeof(cuComplex));
                                    //device memory alloc for a
  cudaStat=cudaMalloc((void**)&d_x,n*sizeof(cuComplex));
                                    //device memory alloc for x
  stat = cublasCreate(&handle);  // initialize CUBLAS context
// copy the matrix and vector from the host to the device
  stat = cublasSetMatrix(n,n,sizeof(*a),a,n,d_a,n);//a -> d_a
  stat = cublasSetVector(n,sizeof(*x),x,1,d_x,1);  //x -> d_x
  float al=1.0f;                                     //  al=1
// rank-1 update of the Hermitian matrix d_a:
// d_a = al*d_x*d_x^H + d_a
// d_a - nxn Hermitian matrix; d_x - n-vector; al - scalar

 stat=cublasCher(handle,CUBLAS_FILL_MODE_LOWER,n,&al,d_x,1,d_a,n);

  stat=cublasGetMatrix(n,n,sizeof(cuComplex),d_a,n,a,n);
                                          // copy d_a-> a
// print the lower triangle of updated a
  printf("lower triangle of updated a after Cher :\n");
  for(i=0;i<n;i++){
   for(j=0;j<n;j++){
    if(i>=j)
    printf("%5.0f+%1.0f*I",a[IDX2C(i,j,n)].x,a[IDX2C(i,j,n)].y);
   }
   printf("\n");
  }
  cudaFree(d_a);                      // free device memory
  cudaFree(d_x);                      // free device memory
  cublasDestroy(handle);             // destroy CUBLAS context
  free(a);                              // free host memory
  free(x);                              // free host memory
  return EXIT_SUCCESS;
}
```

```
// lower triangle of a:
//   11+0*I
//   12+0*I   17+0*I
//   13+0*I   18+0*I   22+0*I
//   14+0*I   19+0*I   23+0*I   26+0*I
//   15+0*I   20+0*I   24+0*I   27+0*I   29+0*I
//   16+0*I   21+0*I   25+0*I   28+0*I   30+0*I   31+0*I

// lower triangle of updated a after Cher :
//   12+0*I
//   13+0*I   18+0*I
//   14+0*I   19+0*I   23+0*I
//   15+0*I   20+0*I   24+0*I   27+0*I
//   16+0*I   21+0*I   25+0*I   28+0*I   30+0*I
//   17+0*I   22+0*I   26+0*I   29+0*I   31+0*I   32+0*I
//         [1]
//         [1]
//         [1]
//  a = 1*[ ]*[1,1,1,1,1,1]+ a
//         [1]
//         [1]
//         [1]
```

3.3.21 cublasCher2 - Hermitian rank-2 update

This function performs the Hermitian rank-2 update

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

where A is an $n \times n$ Hermitian complex matrix, x, y are complex n-vectors and α is a complex scalar. A is stored in column-major format in lower (CUBLAS_FILL_MODE_LOWER) or upper (CUBLAS_FILL_MODE_UPPER) mode.

```
// nvcc 033cher2.c -lcublas
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include "cublas_v2.h"
#define IDX2C(i,j,ld) (((j)*(ld))+( i ))
#define n 6                 // number of rows and columns of a
int main(void){
  cudaError_t cudaStat;                  // cudaMalloc status
  cublasStatus_t stat;             // CUBLAS functions status
  cublasHandle_t handle;                     // CUBLAS context
  int i,j;                       // i-row index, j-column index
// data preparation on the host
  cuComplex *a;              //nxn complex matrix a on the host
  cuComplex *x;               //complex  n-vector x on the host
  cuComplex *y;               //complex  n-vector x on the host
  a=(cuComplex*)malloc(n*n*sizeof(cuComplex)); // host memory
```

```
                                                    // alloc for a
  x=(cuComplex*)malloc(n*sizeof(cuComplex));        // host memory
                                                    // alloc for x
  y=(cuComplex*)malloc(n*sizeof(cuComplex));        // host memory
                                                    // alloc for y
// define the lower triangle of an nxn Hermitian matrix a
// column by column
  int ind=11;                                 // a:
  for(j=0;j<n;j++){                           // 11
    for(i=0;i<n;i++){                         // 12,17
      if(i>=j){                               // 13,18,22
        a[IDX2C(i,j,n)].x=(float)ind++;       // 14,19,23,26
        a[IDX2C(i,j,n)].y=0.0f;               // 15,20,24,27,29
      }                                       // 16,21,25,28,30,31
    }
  }
// print the lower triangle of a row by row
  printf("lower triangle of a:\n");
   for(i=0;i<n;i++){
    for(j=0;j<n;j++){
     if(i>=j)
     printf("%5.0f+%1.0f*I",a[IDX2C(i,j,n)].x,a[IDX2C(i,j,n)].y);
    }
   printf("\n");
   }
  for(i=0;i<n;i++){x[i].x=1.0f;y[i].x=2.0;}//x={1,1,1,1,1,1}^T
                                           //y={2,2,2,2,2,2}^T
// on the device
  cuComplex* d_a;                     // d_a - a on the device
  cuComplex* d_x;                     // d_x - x on the device
  cuComplex* d_y;                     // d_y - y on the device
  cudaStat=cudaMalloc((void**)&d_a,n*n*sizeof(cuComplex));
                                     //device memory alloc for a
  cudaStat=cudaMalloc((void**)&d_x,n*sizeof(cuComplex));
                                     //device memory alloc for x
  cudaStat=cudaMalloc((void**)&d_y,n*sizeof(cuComplex));
                                     //device memory alloc for y
  stat = cublasCreate(&handle);  // initialize CUBLAS context
// copy the matrix and vectors from the host to the device
  stat = cublasSetMatrix(n,n,sizeof(*a),a,n,d_a,n);//a -> d_a
  stat = cublasSetVector(n,sizeof(*x),x,1,d_x,1);  //x -> d_x
  stat = cublasSetVector(n,sizeof(*y),y,1,d_y,1);  //y -> d_y
  cuComplex al={1.0f,0.0f};                          // al=1
// rank-2 update of the Hermitian matrix d_a:
// d_a = al*d_x*d_y^H + \bar{al}*d_y*d_x^H + d_a
// d_a - nxn Hermitian matrix; d_x,d_y - n-vectors; al -scalar

  stat=cublasCher2(handle,CUBLAS_FILL_MODE_LOWER,n,&al,d_x,1,d_y,
                                                      1,d_a,n);
  stat=cublasGetMatrix(n,n,sizeof(*a),d_a,n,a,n);//cp d_a->a
// print the lower triangle of updated a
  printf("lower triangle of updated a after Cher2 :\n");
```

```
  for(i=0;i<n;i++){
   for(j=0;j<n;j++){
    if(i>=j)
    printf("%5.0f+%1.0f*I",a[IDX2C(i,j,n)].x,a[IDX2C(i,j,n)].y);
   }
   printf("\n");
  }
  cudaFree(d_a);                          // free device memory
  cudaFree(d_x);                          // free device memory
  cudaFree(d_y);                          // free device memory
  cublasDestroy(handle);                // destroy CUBLAS context
  free(a);                                  // free host memory
  free(x);                                  // free host memory
  free(y);                                  // free host memory
  return EXIT_SUCCESS;
}
// lower triangle of a:
//   11+0*I
//   12+0*I   17+0*I
//   13+0*I   18+0*I   22+0*I
//   14+0*I   19+0*I   23+0*I   26+0*I
//   15+0*I   20+0*I   24+0*I   27+0*I   29+0*I
//   16+0*I   21+0*I   25+0*I   28+0*I   30+0*I   31+0*I

// lower triangle of updated a after Cher2 :
//   15+0*I
//   16+0*I   21+0*I
//   17+0*I   22+0*I   26+0*I
//   18+0*I   23+0*I   27+0*I   30+0*I
//   19+0*I   24+0*I   28+0*I   31+0*I   33+0*I
//   20+0*I   25+0*I   29+0*I   32+0*I   34+0*I   35+0*I

//[15 16 17 18 19 20]    [1]                  [2]
//[16 21 22 23 24 25]    [1]                  [2]
//[17 22 26 27 28 29]    [1]                  [2]
//[                  ]=1*[ ]*[2,2,2,2,2,2]+1*[ ]*[1,1,1,1,1,1])+a
//[18 23 27 30 31 32]    [1]                  [2]
//[19 24 28 31 33 34]    [1]                  [2]
//[20 25 29 33 34 35]    [1]                  [2]
```

3.3.22 `cublasChpr` - packed Hermitian rank-1 update

This function performs the Hermitian rank-1 update

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

where A is an $n \times n$ complex Hermitian matrix in packed format, x is a complex n-vector and α is a scalar. A can be stored in lower (`CUBLAS_FILL_MODE_LOWER`) or upper (`CUBLAS_FILL_MODE_UPPER`) mode. If A is stored in lower mode, then the elements of the lower triangular part of A are packed together column by column without gaps.

```
// nvcc 034chpr.c -lcublas
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include "cublas_v2.h"
#define n 6                 // number of rows and columns of a
int main(void){
  cudaError_t cudaStat;                    // cudaMalloc status
  cublasStatus_t stat;              // CUBLAS functions status
  cublasHandle_t handle;                      // CUBLAS context
  int i,j,l,m;                  // i-row index, j-column index
// data preparation on the host
  cuComplex *a;                 // lower triangle of a complex
                                  // nxn  matrix a on the host
  cuComplex *x;              // complex n-vector x on the host
  a=(cuComplex*)malloc(n*(n+1)/2*sizeof(*a));  // host memory
                                                 // alloc for a
  x=(cuComplex*)malloc(n*sizeof(cuComplex));   // host memory
                                                 // alloc for x
// define the lower triangle of a Hermi-  //11
// tian a in packed format column by      //12,17
// column  without gaps                   //13,18,22
  for(i=0;i<n*(n+1)/2;i++)               //14,19,23,26
    a[i].x=(float)(11+i);                //15,20,24,27,29
// print upper triangle of a row by row   //16,21,25,28,30,31
  printf("upper triangle of a:\n");
  l=n;j=0;m=0;
  while(l>0){                               // print the lower
    for(i=0;i<m;i++) printf("       ");      // triangle of a
    for(i=j;i<j+l;i++) printf("%3.0f+%1.0f*I",a[i].x,a[i].y);
    printf("\n");
    m++;j=j+l;l--;
  }
  for(i=0;i<n;i++){x[i].x=1.0f;}          //x={1,1,1,1,1,1}^T
// on the device
  cuComplex* d_a;                     // d_a - a on the device
  cuComplex* d_x;                     // d_x - x on the device
  cudaStat=cudaMalloc((void**)&d_a,n*(n+1)/2*sizeof(*a));
                                   //device memory alloc for a
  cudaStat=cudaMalloc((void**)&d_x,n*sizeof(cuComplex));
                                   //device memory alloc for x
  stat = cublasCreate(&handle);  // initialize CUBLAS context
// copy the matrix and vector from the host to the device
  stat = cublasSetVector(n*(n+1)/2,sizeof(*a),a,1,d_a,1);
                                                 // copy a-> d_a
  stat = cublasSetVector(n,sizeof(*x),x,1,d_x,1);  // x-> d_x
  float al=1.0f;                                    // al=1
// rank-1 update of a Hermitian packed complex matrix d_a:
// d_a = al*d_x*d_x^H + d_a;  d_a - Hermitian  nxn complex
// matrix in packed format; d_x - n-vector; al - scalar

  stat=cublasChpr(handle,CUBLAS_FILL_MODE_LOWER,n,&al,d_x,1,d_a);
```

```
  stat=cublasGetVector(n*(n+1)/2,sizeof(*a),d_a,1,a,1);
                                                  // copy d_a-> a
// print the updated upper triangle of a row by row
  printf("updated upper triangle of a after Chpr:\n");
  l=n;j=0;m=0;
  while(l>0){
    for(i=0;i<m;i++) printf("       ");
      for(i=j;i<j+l;i++)
        printf("%3.0f+%1.0f*I",a[i].x,a[i].y);
      printf("\n");
    m++;j=j+l;l--;
  }
  cudaFree(d_a);                          // free device memory
  cudaFree(d_x);                          // free device memory
  cublasDestroy(handle);                // destroy CUBLAS context
  free(a);                                  // free host memory
  free(x);                                  // free host memory
return EXIT_SUCCESS;
}
// upper triangle of a:
// 11+0*I 12+0*I 13+0*I 14+0*I 15+0*I 16+0*I
//        17+0*I 18+0*I 19+0*I 20+0*I 21+0*I
//               22+0*I 23+0*I 24+0*I 25+0*I
//                      26+0*I 27+0*I 28+0*I
//                             29+0*I 30+0*I
//                                    31+0*I

// updated upper triangle of a after Chpr:
// 12+0*I 13+0*I 14+0*I 15+0*I 16+0*I 17+0*I
//        18+0*I 19+0*I 20+0*I 21+0*I 22+0*I
//               23+0*I 24+0*I 25+0*I 26+0*I
//                      27+0*I 28+0*I 29+0*I
//                             30+0*I 31+0*I
//                                    32+0*I
//        [1]
//        [1]
//        [1]
//  a = 1*[ ]*[1,1,1,1,1,1]+ a
//        [1]
//        [1]
//        [1]
```

3.3.23 `cublasChpr2` - packed Hermitian rank-2 update

This function performs the Hermitian rank-2 update

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

where A is an $n \times n$ Hermitian complex matrix in packed format, x, y are complex n-vectors and α is a complex scalar. A can be stored in lower

(CUBLAS_FILL_MODE_LOWER) or upper (CUBLAS_FILL_MODE_UPPER) mode. If A is stored in lower mode, then the elements of the lower triangular part of A are packed together column by column without gaps.

```
// nvcc 035chpr2.c -lcublas
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include "cublas_v2.h"
#define n 6                 // number of rows and columns of a
int main(void){
  cudaError_t cudaStat;                  // cudaMalloc status
  cublasStatus_t stat;             // CUBLAS functions status
  cublasHandle_t handle;                    // CUBLAS context
  int i,j,l,m;                  // i-row index, j-column index
// data preparation on the host
  cuComplex *a;                 // lower triangle of a complex
                                // nxn matrix a on the host
  cuComplex *x;              // complex n-vector x on the host
  cuComplex *y;              // complex n-vector y on the host
  a=(cuComplex*)malloc(n*(n+1)/2*sizeof(*a));  // host memory
                                               // alloc for a
  x=(cuComplex*)malloc(n*sizeof(cuComplex));   // host memory
                                               // alloc for x
  y=(cuComplex*)malloc(n*sizeof(cuComplex));   // host memory
                                               // alloc for y
// define the lower triangle of a Hermi-  //11
// tian a in packed format column by      //12,17
// column without gaps                    //13,18,22
  for(i=0;i<n*(n+1)/2;i++)               //14,19,23,26
    a[i].x=(float)(11+i);                //15,20,24,27,29
// print upper triangle of a  row by row  //16,21,25,28,30,31
  printf("upper triangle of a:\n");
  l=n;j=0;m=0;
  while(l>0){                             // print the upper
    for(i=0;i<m;i++) printf("       ");      // triangle of a
    for(i=j;i<j+l;i++) printf("%3.0f+%1.0f*I",a[i].x,a[i].y);
    printf("\n");
    m++;j=j+l;l--;
  }
  for(i=0;i<n;i++){x[i].x=1.0f;y[i].x=2.0;}
                       //x={1,1,1,1,1,1}^T; y={2,2,2,2,2,2}^T
// on the device
  cuComplex* d_a;                    // d_a - a on the device
  cuComplex* d_x;                    // d_x - x on the device
  cuComplex* d_y;                    // d_y - y on the device
  cudaStat=cudaMalloc((void**)&d_a,n*(n+1)/2*sizeof(*a));
                                   //device memory alloc for a
  cudaStat=cudaMalloc((void**)&d_x,n*sizeof(cuComplex));
                                   //device memory alloc for x
  cudaStat=cudaMalloc((void**)&d_y,n*sizeof(cuComplex));
```

```
                                    //device memory alloc for y
  stat = cublasCreate(&handle);  // initialize CUBLAS context
// copy matrix and vectors from the host to the device
  stat = cublasSetVector(n*(n+1)/2,sizeof(*a),a,1,d_a,1);
                                                  // copy a-> d_a
  stat = cublasSetVector(n,sizeof(*x),x,1,d_x,1);  // x-> d_x
  stat = cublasSetVector(n,sizeof(*y),y,1,d_y,1);  // y-> d_y
  cuComplex al={1.0f,0.0f};                          //al=1
// rank-2 update of a Hermitian matrix d_a :
// d_a = al*d_x*d_y^H + \bar{al}*d_y*d_x^H + d_a; d_a -Herm.
// nxn matrix in packed format; d_x,d_y - n-vectors; al -scal.

  stat=cublasChpr2(handle,CUBLAS_FILL_MODE_LOWER,n,&al,d_x,1,
                                                    d_y,1,d_a);

  stat=cublasGetVector(n*(n+1)/2,sizeof(cuComplex),d_a,1,a,1);
                                                 // copy d_a -> a
// print the updated upper triangle of a row by row
  printf("updated upper triangle of a after Chpr2:\n");
  l=n;j=0;m=0;
  while(l>0){
    for(i=0;i<m;i++) printf("       ");
    for(i=j;i<j+l;i++) printf("%3.0f+%1.0f*I",a[i].x,a[i].y);
    printf("\n");
    m++;j=j+l;l--;
  }
  cudaFree(d_a);                          // free device memory
  cudaFree(d_x);                          // free device memory
  cudaFree(d_y);                          // free device memory
  cublasDestroy(handle);                // destroy CUBLAS context
  free(a);                                  // free host memory
  free(x);                                  // free host memory
  free(y);                                  // free host memory
  return EXIT_SUCCESS;
}
// upper triangle of a:
// 11+0*I 12+0*I 13+0*I 14+0*I 15+0*I 16+0*I
//        17+0*I 18+0*I 19+0*I 20+0*I 21+0*I
//               22+0*I 23+0*I 24+0*I 25+0*I
//                      26+0*I 27+0*I 28+0*I
//                             29+0*I 30+0*I
//                                    31+0*I


// updated upper triangle of a after Chpr2:
// 15+0*I 16+0*I 17+0*I 18+0*I 19+0*I 20+0*I
//        21+0*I 22+0*I 23+0*I 24+0*I 25+0*I
//               26+0*I 27+0*I 28+0*I 29+0*I
//                      30+0*I 31+0*I 32+0*I
//                             33+0*I 34+0*I
//                                    35+0*I
```

```
//[15 16 17 18 19 20]    [1]                    [2]
//[16 21 22 23 24 25]    [1]                    [2]
//[17 22 26 27 28 29]    [1]                    [2]
//[                  ]=1*[ ]*[2,2,2,2,2,2]+1*[ ]*[1,1,1,1,1,1])+a
//[18 23 27 30 31 32]    [1]                    [2]
//[19 24 28 31 33 34]    [1]                    [2]
//[20 25 29 33 34 35]    [1]                    [2]
```

3.4 CUBLAS Level-3. Matrix-matrix operations

3.4.1 `cublasSgemm` - matrix-matrix multiplication

This function performs the matrix-matrix multiplication

$$C = \alpha op(A) op(B) + \beta C,$$

where A, B are matrices in column-major format and α, β are scalars. The value of $op(A)$ can be equal to A (`CUBLAS_OP_N` case), A^T (transposition) in `CUBLAS_OP_T` case, or A^H (conjugate transposition) in `CUBLAS_OP_C` case and similarly for $op(B)$.

```
// nvcc 036sgemm.c -lcublas
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include "cublas_v2.h"
#define IDX2C(i,j,ld) (((j)*(ld))+( i ))
#define m 6                                     // a - mxk matrix
#define n 4                                     // b - kxn matrix
#define k 5                                     // c - mxn matrix
int main(void){
  cudaError_t cudaStat;                     // cudaMalloc status
  cublasStatus_t stat;                // CUBLAS functions status
  cublasHandle_t handle;                        // CUBLAS context
  int i,j;                           // i-row index,j-column index
  float* a;                           // mxk matrix a on the host
  float* b;                           // kxn matrix b on the host
  float* c;                           // mxn matrix c on the host
  a=(float*)malloc(m*k*sizeof(float));    // host memory for a
  b=(float*)malloc(k*n*sizeof(float));    // host memory for b
  c=(float*)malloc(m*n*sizeof(float));    // host memory for c
// define an mxk matrix a column by column
  int ind=11;                                   // a:
  for(j=0;j<k;j++){                             // 11,17,23,29,35
    for(i=0;i<m;i++){                           // 12,18,24,30,36
      a[IDX2C(i,j,m)]=(float)ind++;             // 13,19,25,31,37
    }                                           // 14,20,26,32,38
   }                                            // 15,21,27,33,39
                                                // 16,22,28,34,40
```

```
// print a row by row
  printf("a:\n");
   for(i=0;i<m;i++){
     for(j=0;j<k;j++){
       printf("%5.0f",a[IDX2C(i,j,m)]);
     }
    printf("\n");
   }
// define a kxn matrix b column by column
  ind=11;                                          // b:
  for(j=0;j<n;j++){                                // 11,16,21,26
    for(i=0;i<k;i++){                              // 12,17,22,27
      b[IDX2C(i,j,k)]=(float)ind++;                // 13,18,23,28
    }                                              // 14,19,24,29
  }                                                // 15,20,25,30
// print b row by row
  printf("b:\n");
   for(i=0;i<k;i++){
     for(j=0;j<n;j++){
       printf("%5.0f",b[IDX2C(i,j,k)]);
     }
    printf("\n");
   }
// define an mxn matrix c column by column
  ind=11;                                          // c:
  for(j=0;j<n;j++){                                // 11,17,23,29
    for(i=0;i<m;i++){                              // 12,18,24,30
      c[IDX2C(i,j,m)]=(float)ind++;                // 13,19,25,31
    }                                              // 14,20,26,32
  }                                                // 15,21,27,33
                                                   // 16,22,28,34
// print c row by row
  printf("c:\n");
   for(i=0;i<m;i++){
     for(j=0;j<n;j++){
       printf("%5.0f",c[IDX2C(i,j,m)]);
     }
    printf("\n");
   }
// on the device
  float* d_a;                          // d_a - a on the device
  float* d_b;                          // d_b - b on the device
  float* d_c;                          // d_c - c on the device
  cudaStat=cudaMalloc((void**)&d_a,m*k*sizeof(*a));  //device
                                       // memory alloc for a
  cudaStat=cudaMalloc((void**)&d_b,k*n*sizeof(*b));  //device
                                       // memory alloc for b
  cudaStat=cudaMalloc((void**)&d_c,m*n*sizeof(*c));  //device
                                       // memory alloc for c
  stat = cublasCreate(&handle);  // initialize CUBLAS context
// copy matrices from the host to the device
  stat = cublasSetMatrix(m,k,sizeof(*a),a,m,d_a,m);//a -> d_a
```

```
  stat = cublasSetMatrix(k,n,sizeof(*b),b,k,d_b,k);//b -> d_b
  stat = cublasSetMatrix(m,n,sizeof(*c),c,m,d_c,m);//c -> d_c
  float al=1.0f;                                        // al=1
  float bet=1.0f;                                       //bet=1
// matrix-matrix multiplication: d_c = al*d_a*d_b + bet*d_c
// d_a -mxk matrix, d_b -kxn matrix, d_c -mxn matrix;
// al,bet -scalars

  stat=cublasSgemm(handle,CUBLAS_OP_N,CUBLAS_OP_N,m,n,k,&al,d_a,
                                              m,d_b,k,&bet,d_c,m);

  stat=cublasGetMatrix(m,n,sizeof(*c),d_c,m,c,m); //cp d_c->c
  printf("c after Sgemm :\n");
  for(i=0;i<m;i++){
    for(j=0;j<n;j++){
      printf("%7.0f",c[IDX2C(i,j,m)]);  //print c after Sgemm
    }
    printf("\n");
  }
  cudaFree(d_a);                          // free device memory
  cudaFree(d_b);                          // free device memory
  cudaFree(d_c);                          // free device memory
  cublasDestroy(handle);                // destroy CUBLAS context
  free(a);                                  // free host memory
  free(b);                                  // free host memory
  free(c);                                  // free host memory
  return EXIT_SUCCESS;
}
// a:
//    11    17    23    29    35
//    12    18    24    30    36
//    13    19    25    31    37
//    14    20    26    32    38
//    15    21    27    33    39
//    16    22    28    34    40
// b:
//    11    16    21    26
//    12    17    22    27
//    13    18    23    28
//    14    19    24    29
//    15    20    25    30
// c:
//    11    17    23    29
//    12    18    24    30
//    13    19    25    31
//    14    20    26    32
//    15    21    27    33
//    16    22    28    34

// c after Sgemm :
//    1566    2147    2728    3309
//    1632    2238    2844    3450
```

```
//    1698    2329    2960    3591      //  c=al*a*b+bet*c
//    1764    2420    3076    3732
//    1830    2511    3192    3873
//    1896    2602    3308    4014
```

3.4.2 cublasSsymm - symmetric matrix-matrix multiplication

This function performs the left or right symmetric matrix-matrix multiplications

$$C = \alpha AB + \beta C \qquad \text{in CUBLAS\_SIDE\_LEFT case,}$$
$$C = \alpha BA + \beta C \qquad \text{in CUBLAS\_SIDE\_RIGHT case.}$$

The symmetric matrix A has dimension $m \times m$ in the first case and $n \times n$ in the second one. The general matrices B, C have dimensions $m \times n$ and α, β are scalars. The matrix A can be stored in lower (CUBLAS_FILL_MODE_LOWER) or upper (CUBLAS_FILL_MODE_UPPER) mode.

```
// nvcc 037ssymm.c -lcublas
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include "cublas_v2.h"
#define IDX2C(i,j,ld) (((j)*(ld))+( i ))
#define m 6                                    // a - mxm matrix
#define n 4                                 // b,c - mxn matrices
int main(void){
  cudaError_t cudaStat;                       // cudaMalloc status
  cublasStatus_t stat;                 // CUBLAS functions status
  cublasHandle_t handle;                      // CUBLAS context
  int i,j;                            // i-row ind., j-column ind.
  float* a;                           // mxm matrix a on the host
  float* b;                           // mxn matrix b on the host
  float* c;                           // mxn matrix c on the host
  a=(float*)malloc(m*m*sizeof(float));   // host memory for a
  b=(float*)malloc(m*n*sizeof(float));   // host memory for b
  c=(float*)malloc(m*n*sizeof(float));   // host memory for c
// define the lower triangle of an mxm symmetric matrix a in
// lower mode column by column
  int ind=11;                                 // a:
  for(j=0;j<m;j++){                           // 11
    for(i=0;i<m;i++){                         // 12,17
      if(i>=j){                               // 13,18,22
        a[IDX2C(i,j,m)]=(float)ind++;         // 14,19,23,26
      }                                       // 15,20,24,27,29
    }                                         // 16,21,25,28,30,31
  }
// print the lower triangle of a row by row
  printf("lower triangle of a:\n");
```

```
    for(i=0;i<m;i++){
      for(j=0;j<m;j++){
        if(i>=j)
        printf("%5.0f",a[IDX2C(i,j,m)]);
      }
     printf("\n");
    }
// define mxn matrices b,c column by column
  ind=11;                                          // b,c:
  for(j=0;j<n;j++){                                // 11,17,23,29
    for(i=0;i<m;i++){                              // 12,18,24,30
      b[IDX2C(i,j,m)]=(float)ind;                  // 13,19,25,31
      c[IDX2C(i,j,m)]=(float)ind;                  // 14,20,26,32
      ind++;                                       // 15,21,27,33
    }                                              // 16,22,28,34
  }
// print b(=c) row by row
  printf("b(=c):\n");
   for(i=0;i<m;i++){
     for(j=0;j<n;j++){
       printf("%5.0f",b[IDX2C(i,j,m)]);
     }
     printf("\n");
   }
// on the device
  float* d_a;                          // d_a - a on the device
  float* d_b;                          // d_b - b on the device
  float* d_c;                          // d_c - c on the device
  cudaStat=cudaMalloc((void**)&d_a,m*m*sizeof(*a));  //device
                                       // memory alloc for a
  cudaStat=cudaMalloc((void**)&d_b,m*n*sizeof(*b));  //device
                                       // memory alloc for b
  cudaStat=cudaMalloc((void**)&d_c,m*n*sizeof(*c));  //device
                                       // memory alloc for c
  stat = cublasCreate(&handle);  // initialize CUBLAS context
// copy matrices from the host to the device
  stat = cublasSetMatrix(m,m,sizeof(*a),a,m,d_a,m);//a -> d_a
  stat = cublasSetMatrix(m,n,sizeof(*b),b,m,d_b,m);//b -> d_b
  stat = cublasSetMatrix(m,n,sizeof(*c),c,m,d_c,m);//c -> d_c
  float al=1.0f;                                     // al=1
  float bet=1.0f;                                   // bet=1
// symmetric matrix-matrix multiplication:
//  d_c = al*d_a*d_b + bet*d_c; d_a - mxm symmetric matrix;
// d_b,d_c - mxn general matrices; al,bet - scalars

  stat=cublasSsymm(handle,CUBLAS_SIDE_LEFT,CUBLAS_FILL_MODE_LOWER,
                                  m,n,&al,d_a,m,d_b,m,&bet,d_c,m);

  stat=cublasGetMatrix(m,n,sizeof(*c),d_c,m,c,m); //d_c -> c
  printf("c after Ssymm :\n");          //print c after Ssymm
  for(i=0;i<m;i++){
    for(j=0;j<n;j++){
```

```
      printf("%7.0f",c[IDX2C(i,j,m)]);
    }
    printf("\n");
  }
  cudaFree(d_a);                          // free device memory
  cudaFree(d_b);                          // free device memory
  cudaFree(d_c);                          // free device memory
  cublasDestroy(handle);                // destroy CUBLAS context
  free(a);                                  // free host memory
  free(b);                                  // free host memory
  free(c);                                  // free host memory
  return EXIT_SUCCESS;
}
// lower triangle of a:
//    11
//    12    17
//    13    18    22
//    14    19    23    26
//    15    20    24    27    29
//    16    21    25    28    30    31
// b(=c):
//    11    17    23    29
//    12    18    24    30
//    13    19    25    31
//    14    20    26    32
//    15    21    27    33
//    16    22    28    34

// c after Ssymm :
//    1122    1614    2106    2598
//    1484    2132    2780    3428
//    1740    2496    3252    4008        // c=al*a*b+bet*c
//    1912    2740    3568    4396
//    2025    2901    3777    4653
//    2107    3019    3931    4843
```

3.4.3 `cublasSsyrk` - symmetric rank-k update

This function performs the symmetric rank-k update

$$C = \alpha\, op(A) op(A)^T + \beta C,$$

where $op(A)$ is an $n \times k$ matrix, C is a symmetric $n \times n$ matrix stored in lower (`CUBLAS_FILL_MODE_LOWER`) or upper (`CUBLAS_FILL_MODE_UPPER`) mode and α, β are scalars. The value of $op(A)$ can be equal to A in `CUBLAS_OP_N` case or A^T (transposition) in `CUBLAS_OP_T` case.

```
// nvcc 038ssyrk.c -lcublas
#include <stdio.h>
#include <stdlib.h>
```

```
#include <cuda_runtime.h>
#include "cublas_v2.h"
#define IDX2C(i,j,ld) (((j)*(ld))+( i ))
#define n 6                                     // a - nxk matrix
#define k 4                                     // c - nxn matrix
int main(void){
  cudaError_t cudaStat;                    // cudaMalloc status
  cublasStatus_t stat;              // CUBLAS functions status
  cublasHandle_t handle;                    // CUBLAS context
  int i,j;                        // i-row index, j-column index
  float* a;                          // nxk matrix a on the host
  float* c;                          // nxn matrix c on the host
  a=(float*)malloc(n*k*sizeof(float));   // host memory for a
  c=(float*)malloc(n*n*sizeof(float));   // host memory for c
// define the lower triangle of an nxn symmetric matrix c
// column by column
  int ind=11;                                   // c:
  for(j=0;j<n;j++){                             // 11
    for(i=0;i<n;i++){                           // 12,17
      if(i>=j){                                 // 13,18,22
        c[IDX2C(i,j,n)]=(float)ind++;           // 14,19,23,26
      }                                         // 15,20,24,27,29
    }                                           // 16,21,25,28,30,31
  }
// print the lower triangle of c row by row
  printf("lower triangle of c:\n");
   for(i=0;i<n;i++){
     for(j=0;j<n;j++){
       if(i>=j)
       printf("%5.0f",c[IDX2C(i,j,n)]);
     }
    printf("\n");
   }
// define an nxk matrix a column by column
  ind=11;                                           // a:
  for(j=0;j<k;j++){                                 // 11,17,23,29
    for(i=0;i<n;i++){                               // 12,18,24,30
      a[IDX2C(i,j,n)]=(float)ind;                   // 13,19,25,31
      ind++;                                        // 14,20,26,32
    }                                               // 15,21,27,33
  }                                                 // 16,22,28,34
  printf("a:\n");
   for(i=0;i<n;i++){
     for(j=0;j<k;j++){
       printf("%5.0f",a[IDX2C(i,j,n)]); // print a row by row
     }
    printf("\n");
   }

// on the device
  float* d_a;                          // d_a - a on the device
  float* d_c;                          // d_c - c on the device
```

```
  cudaStat=cudaMalloc((void**)&d_a,n*k*sizeof(*a));  //device
                                          // memory alloc for a
  cudaStat=cudaMalloc((void**)&d_c,n*n*sizeof(*c));  //device
                                          // memory alloc for c
  stat = cublasCreate(&handle);  // initialize CUBLAS context
// copy matrices from the host to the device
  stat = cublasSetMatrix(n,k,sizeof(*a),a,n,d_a,n);//a -> d_a
  stat = cublasSetMatrix(n,n,sizeof(*c),c,n,d_c,n);//c -> d_c
  float al=1.0f;                                        // al=1
  float bet=1.0f;                                       //bet=1
// symmetric rank-k update: c = al*d_a*d_a^T + bet*d_c;
// d_c - symmetric nxn matrix, d_a - general nxk matrix;
// al,bet - scalars

  stat=cublasSsyrk(handle,CUBLAS_FILL_MODE_LOWER,CUBLAS_OP_N,
                                  n,k,&al,d_a,n,&bet,d_c,n);

  stat=cublasGetMatrix(n,n,sizeof(*c),d_c,n,c,n);// d_c -> c
  printf("lower triangle of updated c after Ssyrk :\n");
  for(i=0;i<n;i++){
    for(j=0;j<n;j++){
      if(i>=j)                        //print the lower triangle
      printf("%7.0f",c[IDX2C(i,j,n)]);     //of c after Ssyrk
    }
    printf("\n");
  }
  cudaFree(d_a);                          // free device memory
  cudaFree(d_c);                          // free device memory
  cublasDestroy(handle);             // destroy CUBLAS context
  free(a);                                  // free host memory
  free(c);                                  // free host memory
  return EXIT_SUCCESS;
}
// lower triangle of c:
//    11
//    12    17
//    13    18    22
//    14    19    23    26
//    15    20    24    27    29
//    16    21    25    28    30    31

// a:
//    11    17    23    29
//    12    18    24    30
//    13    19    25    31
//    14    20    26    32
//    15    21    27    33
//    16    22    28    34

// lower triangle of updated c after Ssyrk: c=al*a*a^T+bet*c
//    1791
//    1872    1961
```

```
//    1953    2046    2138
//    2034    2131    2227    2322
//    2115    2216    2316    2415    2513
//    2196    2301    2405    2508    2610    2711
```

3.4.4 cublasSsyr2k - symmetric rank-2k update

This function performs the symmetric rank-2k update

$$C = \alpha(op(A)op(B)^T + op(B)op(A)^T) + \beta C,$$

where $op(A), op(B)$ are $n \times k$ matrices, C is a symmetric $n \times n$ matrix stored in lower (CUBLAS_FILL_MODE_LOWER) or upper (CUBLAS_FILL_MODE_UPPER) mode and α, β are scalars. The value of $op(A)$ can be equal to A in CUBLAS_OP_N case or A^T (transposition) in CUBLAS_OP_T case and similarly for $op(B)$.

```
// nvcc 039ssyrk.c -lcublas
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include "cublas_v2.h"
#define IDX2C(i,j,ld) (((j)*(ld))+( i ))
#define n 6                                     // c - nxn matrix
#define k 4                                  // a,b - nxk matrices
int main(void){
  cudaError_t cudaStat;                       // cudaMalloc status
  cublasStatus_t stat;                  // CUBLAS functions status
  cublasHandle_t handle;                        // CUBLAS context
  int i,j;                             // i-row index, j-col. index
  float* a;                              // nxk matrix on the host
  float* b;                              // nxk matrix on the host
  float* c;                              // nxn matrix on the host
  a=(float*)malloc(n*k*sizeof(float));   // host memory for a
  b=(float*)malloc(n*k*sizeof(float));   // host memory for b
  c=(float*)malloc(n*n*sizeof(float));   // host memory for c
// define the lower triangle of an nxn symmetric matrix c in
// lower mode  column by column
  int ind=11;                                // c:
  for(j=0;j<n;j++){                          // 11
    for(i=0;i<n;i++){                        // 12,17
      if(i>=j){                              // 13,18,22
        c[IDX2C(i,j,n)]=(float)ind++;        // 14,19,23,26,
      }                                      // 15,20,24,27,29
    }                                        // 16,21,25,28,30,31
  }
// print the lower triangle of c row by row
  printf("lower triangle of c:\n");
   for(i=0;i<n;i++){
```

```
      if(i>=j)
      for(j=0;j<n;j++){
        printf("%5.0f",c[IDX2C(i,j,n)]);
      }
     printf("\n");
    }
// define nxk matrices a,b column by column
  ind=11;                                          // a,b:
  for(j=0;j<k;j++){                                // 11,17,23,29
    for(i=0;i<n;i++){                              // 12,18,24,30
      a[IDX2C(i,j,n)]=(float)ind;                  // 13,19,25,31
      b[IDX2C(i,j,n)]=(float)ind;                  // 14,20,26,32
      ind++;                                       // 15,21,27,33
    }                                              // 16,22,28,34
  }
  printf("a(=b):\n");
   for(i=0;i<n;i++){
     for(j=0;j<k;j++){
       printf("%5.0f",a[IDX2C(i,j,n)]); // print a row by row
     }
     printf("\n");
    }
// on the device
  float* d_a;                          // d_a - a on the device
  float* d_b;                          // d_b - b on the device
  float* d_c;                          // d_c - c on the device
  cudaStat=cudaMalloc((void**)&d_a,n*k*sizeof(*a));  //device
                                       // memory alloc for a
  cudaStat=cudaMalloc((void**)&d_b,n*k*sizeof(*b));  //device
                                       // memory alloc for b
  cudaStat=cudaMalloc((void**)&d_c,n*n*sizeof(*c));  //device
                                       // memory alloc for c
  stat = cublasCreate(&handle);  // initialize CUBLAS context
// copy matrices from the host to the device
  stat = cublasSetMatrix(n,k,sizeof(*a),a,n,d_a,n);//a -> d_a
  stat = cublasSetMatrix(n,k,sizeof(*b),b,n,d_b,n);//b -> d_b
  stat = cublasSetMatrix(n,n,sizeof(*c),c,n,d_c,n);//c -> d_c
  float al=1.0f;                                          // al=1
  float bet=1.0f;                                         //bet=1
// symmetric rank-2k update:
// d_c=al*(d_a*d_b^T+d_b*d_a^T)+bet*d_c
// d_c - symmetric nxn matrix, d_a,d_b - general nxk matrices
// al,bet - scalars

  stat=cublasSsyr2k(handle,CUBLAS_FILL_MODE_LOWER,CUBLAS_OP_N,
                            n,k,&al,d_a,n,d_b,n,&bet,d_c,n);
  stat=cublasGetMatrix(n,n,sizeof(*c),d_c,n,c,n); //d_c -> c
  printf("lower triangle of updated c after Ssyr2k :\n");
  for(i=0;i<n;i++){
    for(j=0;j<n;j++){
      if(i>=j)                     //print the lower triangle
      printf("%7.0f",c[IDX2C(i,j,n)]);    //of c after Ssyr2k
```

```
    }
    printf("\n");
  }
  cudaFree(d_a);                          // free device memory
  cudaFree(d_b);                          // free device memory
  cudaFree(d_c);                          // free device memory
  cublasDestroy(handle);                // destroy CUBLAS context
  free(a);                                  // free host memory
  free(b);                                  // free host memory
  free(c);                                  // free host memory
  return EXIT_SUCCESS;
}
// lower triangle of c:
//   11
//   12   17
//   13   18   22
//   14   19   23   26
//   15   20   24   27   29
//   16   21   25   28   30   31
// a(=b):
//   11   17   23   29
//   12   18   24   30
//   13   19   25   31
//   14   20   26   32
//   15   21   27   33
//   16   22   28   34

// lower triangle of updated c after Ssyr2k :
//   3571
//   3732   3905
//   3893   4074   4254
//   4054   4243   4431   4618
//   4215   4412   4608   4803   4997
//   4376   4581   4785   4988   5190   5391

// c = al(a*b^T + b*a^T) + bet*c
```

3.4.5 `cublasStrmm` - triangular matrix-matrix multiplication

This function performs the left or right triangular matrix-matrix multiplications

$$C = \alpha\, op(A)\, B \qquad \text{in } \texttt{CUBLAS\_SIDE\_LEFT} \text{ case,}$$
$$C = \alpha\, B\, op(A) \qquad \text{in } \texttt{CUBLAS\_SIDE\_RIGHT} \text{ case,}$$

where A is a triangular matrix, C, B are $m \times n$ matrices and α is a scalar. The value of $op(A)$ can be equal to A in `CUBLAS_OP_N` case, A^T (transposition) in `CUBLAS_OP_T` case or A^H (conjugate transposition) in `CUBLAS_OP_C` case. A has dimension $m \times m$ in the first case and $n \times n$ in the second case. A can be stored in lower (`CUBLAS_FILL_MODE_LOWER`) or upper

(CUBLAS_FILL_MODE_UPPER) mode. If the diagonal of the matrix A has non-unit elements, then the parameter CUBLAS_DIAG_NON_UNIT should be used (in the opposite case - CUBLAS_DIAG_UNIT).

```
// nvcc 040strmm.c -lcublas
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include "cublas_v2.h"
#define IDX2C(i,j,ld) (((j)*(ld))+( i ))
#define m 6                                    // a - mxm matrix
#define n 5                                 // b,c - mxn matrices
int main(void){
  cudaError_t cudaStat;                      // cudaMalloc status
  cublasStatus_t stat;                 // CUBLAS functions status
  cublasHandle_t handle;                        // CUBLAS context
  int i,j;                            // i-row index, j-col. index
  float* a;                            // mxm matrix a on the host
  float* b;                            // mxn matrix b on the host
  float* c;                            // mxn matrix c on the host
  a=(float*)malloc(m*m*sizeof(float));    // host memory for a
  b=(float*)malloc(m*n*sizeof(float));    // host memory for b
  c=(float*)malloc(m*n*sizeof(float));    // host memory for c
// define the lower triangle of an mxm triangular matrix a in
// lower mode  column by column
  int ind=11;                                 // a:
  for(j=0;j<m;j++){                           // 11
    for(i=0;i<m;i++){                         // 12,17
      if(i>=j){                               // 13,18,22
        a[IDX2C(i,j,m)]=(float)ind++;         // 14,19,23,26
      }                                       // 15,20,24,27,29
    }                                         // 16,21,25,28,30,31
  }
// print the lower triangle of a row by row
  printf("lower triangle of a:\n");
   for(i=0;i<m;i++){
     for(j=0;j<m;j++){
       if(i>=j)
       printf("%5.0f",a[IDX2C(i,j,m)]);
     }
    printf("\n");
   }
// define an mxn matrix b column by column
  ind=11;                                        // b:
  for(j=0;j<n;j++){                              // 11,17,23,29,35
    for(i=0;i<m;i++){                            // 12,18,24,30,36
      b[IDX2C(i,j,m)]=(float)ind++;              // 13,19,25,31,37
    }                                            // 14,20,26,32,38
  }                                              // 15,21,27,33,39
                                                 // 16,22,28,34,40
  printf("b:\n");
```

```
    for(i=0;i<m;i++){
      for(j=0;j<n;j++){
        printf("%5.0f",b[IDX2C(i,j,m)]);//  print b row by row
      }
     printf("\n");
    }
// on the device
  float* d_a;                          // d_a - a on the device
  float* d_b;                          // d_b - b on the device
  float* d_c;                          // d_c - c on the device
  cudaStat=cudaMalloc((void**)&d_a,m*m*sizeof(*a));  //device
                                       // memory alloc for a
  cudaStat=cudaMalloc((void**)&d_b,m*n*sizeof(*b));  //device
                                       // memory alloc for b
  cudaStat=cudaMalloc((void**)&d_c,m*n*sizeof(*c));  //device
                                       // memory alloc for c
  stat = cublasCreate(&handle);  // initialize CUBLAS context
// copy matrices from the host to the device
  stat = cublasSetMatrix(m,m,sizeof(*a),a,m,d_a,m);//a -> d_a
  stat = cublasSetMatrix(m,n,sizeof(*b),b,m,d_b,m);//b -> d_b
  float al=1.0f;
// triangular matrix-matrix multiplication: d_c = al*d_a*d_b;
// d_a - mxm triangular matrix in lower mode,
// d_b,d_c -mxn general matrices; al- scalar

 stat=cublasStrmm(handle,CUBLAS_SIDE_LEFT,CUBLAS_FILL_MODE_LOWER,
     CUBLAS_OP_N,CUBLAS_DIAG_NON_UNIT,m,n,&al,d_a,m,d_b,m,d_c,m);

  stat=cublasGetMatrix(m,n,sizeof(*c),d_c,m,c,m); //d_c -> c
  printf("c after Strmm :\n");
  for(i=0;i<m;i++){
    for(j=0;j<n;j++){
      printf("%7.0f",c[IDX2C(i,j,m)]); //print c after Strmm
    }
    printf("\n");
  }
  cudaFree(d_a);                       // free device memory
  cudaFree(d_b);                       // free device memory
  cudaFree(d_c);                       // free device memory
  cublasDestroy(handle);             // destroy CUBLAS context
  free(a);                               // free host memory
  free(b);                               // free host memory
  free(c);                               // free host memory
  return EXIT_SUCCESS;
}
// lower triangle of a:
//    11
//    12    17
//    13    18    22
//    14    19    23    26
//    15    20    24    27    29
//    16    21    25    28    30    31
```

```
// b:
//    11    17    23    29    35
//    12    18    24    30    36
//    13    19    25    31    37
//    14    20    26    32    38
//    15    21    27    33    39
//    16    22    28    34    40

// c after Strmm :
//     121     187     253     319     385
//     336     510     684     858    1032
//     645     963    1281    1599    1917   // c = al*a*b
//    1045    1537    2029    2521    3013
//    1530    2220    2910    3600    4290
//    2091    2997    3903    4809    5715
```

3.4.6 cublasStrsm - solving the triangular linear system

This function solves the triangular system

$$op(A)\,X = \alpha\,B \qquad \text{in CUBLAS\_SIDE\_LEFT case,}$$
$$X\,op(A) = \alpha\,B \qquad \text{in CUBLAS\_SIDE\_RIGHT case,}$$

where A is a triangular matrix, X, B are $m \times n$ matrices and α is a scalar. The value of $op(A)$ can be equal to A in CUBLAS_OP_N case, A^T (transposition) in CUBLAS_OP_T case or A^H (conjugate transposition) in CUBLAS_OP_C case. A has dimension $m \times m$ in the first case and $n \times n$ in the second and third case. A can be stored in lower (CUBLAS_FILL_MODE_LOWER) or upper (CUBLAS_FILL_MODE_UPPER) mode. If the diagonal of the matrix A has non-unit elements, then the parameter CUBLAS_DIAG_NON_UNIT should be used (in the opposite case - CUBLAS_DIAG_UNIT).

```
// nvcc 041strsm.c -lcublas
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include "cublas_v2.h"
#define IDX2C(i,j,ld) (((j)*(ld))+( i ))
#define m 6                                     // a - mxm matrix
#define n 5                                  // b,x - mxn matrices
int main(void){
  cudaError_t cudaStat;                      // cudaMalloc status
  cublasStatus_t stat;                 // CUBLAS functions status
  cublasHandle_t handle;                       // CUBLAS context
  int i,j;                           // i-row index, j-col. index
  float* a;                           // mxm matrix a on the host
  float* b;                           // mxn matrix b on the host
  a=(float*)malloc(m*m*sizeof(float));   // host memory for a
```

```
  b=(float*)malloc(m*n*sizeof(float));    // host memory for b
// define the lower triangle of an mxm triangular matrix a in
// lower  mode  column by column
  int ind=11;                                  // a:
  for(j=0;j<m;j++){                            // 11
    for(i=0;i<m;i++){                          // 12,17
      if(i>=j){                                // 13,18,22
        a[IDX2C(i,j,m)]=(float)ind++;          // 14,19,23,26
      }                                        // 15,20,24,27,29
    }                                          // 16,21,25,28,30,31
  }
// print the lower triangle of a row by row
  printf("lower triangle of a:\n");
   for(i=0;i<m;i++){
     for(j=0;j<m;j++){
       if(i>=j)
       printf("%5.0f",a[IDX2C(i,j,m)]);
     }
    printf("\n");
   }
// define an mxn matrix b column by column
  ind=11;                                      // b:
  for(j=0;j<n;j++){                            // 11,17,23,29,35
    for(i=0;i<m;i++){                          // 12,18,24,30,36
      b[IDX2C(i,j,m)]=(float)ind;              // 13,19,25,31,37
      ind++;                                   // 14,20,26,32,38
    }                                          // 15,21,27,33,39
  }                                            // 16,22,28,34,40
  printf("b:\n");
   for(i=0;i<m;i++){
     for(j=0;j<n;j++){
       printf("%5.0f",b[IDX2C(i,j,m)]); // print b row by row
     }
    printf("\n");
   }
// on the device
  float* d_a;                          // d_a - a on the device
  float* d_b;                          // d_b - b on the device
  cudaStat=cudaMalloc((void**)&d_a,m*m*sizeof(*a));  //device
                                       // memory alloc for a
  cudaStat=cudaMalloc((void**)&d_b,m*n*sizeof(*b));  //device
                                       // memory alloc for b
  stat = cublasCreate(&handle);  // initialize CUBLAS context
// copy matrices from the host to the device
  stat = cublasSetMatrix(m,m,sizeof(*a),a,m,d_a,m);//a -> d_a
  stat = cublasSetMatrix(m,n,sizeof(*b),b,m,d_b,m);//b -> d_b
  float al=1.0f;                                     // al=1
// solve d_a*x=al*d_b; the solution x overwrites rhs d_b;
// d_a - mxm triangular matrix in lower mode;
// d_b,x - mxn general matrices; al - scalar
```

```
    stat=cublasStrsm(handle,CUBLAS_SIDE_LEFT,CUBLAS_FILL_MODE_LOWER,
           CUBLAS_OP_N,CUBLAS_DIAG_NON_UNIT,m,n,&al,d_a,m,d_b,m);

    stat=cublasGetMatrix(m,n,sizeof(*b),d_b,m,b,m); // d_b -> b
    printf("solution x from Strsm :\n");
    for(i=0;i<m;i++){
      for(j=0;j<n;j++){
        printf("%11.5f",b[IDX2C(i,j,m)]); //print b after Strsm
      }
      printf("\n");
    }
    cudaFree(d_a);                          // free device memory
    cudaFree(d_b);                          // free device memory
    cublasDestroy(handle);                // destroy CUBLAS context
    free(a);                                  // free host memory
    free(b);                                  // free host memory
    return EXIT_SUCCESS;
}
// lower triangle of a:
//    11
//    12    17
//    13    18    22
//    14    19    23    26
//    15    20    24    27    29
//    16    21    25    28    30    31
// b:
//    11    17    23    29    35
//    12    18    24    30    36
//    13    19    25    31    37
//    14    20    26    32    38
//    15    21    27    33    39
//    16    22    28    34    40

// solution x from Strsm :  a*x=b
//     1.00000     1.54545     2.09091     2.63636     3.18182
//     0.00000    -0.03209    -0.06417    -0.09626    -0.12834
//     0.00000    -0.02333    -0.04667    -0.07000    -0.09334
//     0.00000    -0.01885    -0.03769    -0.05654    -0.07539
//     0.00000    -0.01625    -0.03250    -0.04874    -0.06499
//     0.00000    -0.01468    -0.02935    -0.04403    -0.05870
```

3.4.7 `cublasChemm` - Hermitian matrix-matrix multiplication

This function performs the Hermitian matrix-matrix multiplication

$$C = \alpha AB + \beta C \qquad \text{in } \texttt{CUBLAS\_SIDE\_LEFT} \text{ case,}$$
$$C = \alpha BA + \beta C \qquad \text{in } \texttt{CUBLAS\_SIDE\_RIGHT} \text{ case,}$$

where A is a Hermitian $m \times m$ matrix in the first case and $n \times n$ Hermitian matrix in the second case, B, C are general $m \times n$ matrices and $\alpha.\beta$

are scalars. A can be stored in lower (`CUBLAS_FILL_MODE_LOWER`) or upper (`CUBLAS_FILL_MODE_UPPER`) mode.

```
// nvcc 042chemm.c -lcublas
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include "cublas_v2.h"
#define IDX2C(i,j,ld) (((j)*(ld))+( i ))
#define m 6                                     // a - mxm matrix
#define n 5                                  // b,c - mxn matrices
int main(void){
  cudaError_t cudaStat;                      // cudaMalloc status
  cublasStatus_t stat;                 // CUBLAS functions status
  cublasHandle_t handle;                       // CUBLAS context
  int i,j;                              // i-row index, j-col. ind.
// data preparation on the host
  cuComplex *a;              // mxm complex matrix a on the host
  cuComplex *b;              // mxn complex matrix b on the host
  cuComplex *c;              // mxn complex matrix c on the host
  a=(cuComplex*)malloc(m*m*sizeof(cuComplex)); // host memory
                                               // alloc for a
  b=(cuComplex*)malloc(m*n*sizeof(cuComplex)); // host memory
                                               // alloc for b
  c=(cuComplex*)malloc(m*n*sizeof(cuComplex)); // host memory
                                               // alloc for c
// define the lower triangle of an mxm Hermitian matrix a in
// lower  mode  column by column
  int ind=11;                                  // a:
  for(j=0;j<m;j++){                            // 11
    for(i=0;i<m;i++){                          // 12,17
      if(i>=j){                                // 13,18,22
        a[IDX2C(i,j,m)].x=(float)ind++;        // 14,19,23,26
        a[IDX2C(i,j,m)].y=0.0f;                // 15,20,24,27,29
      }                                        // 16,21,25,28,30,31
    }
  }

//print the lower triangle of a row by row
  printf("lower triangle of a:\n");
   for(i=0;i<m;i++){
     for(j=0;j<m;j++){
       if(i>=j)
       printf("%5.0f+%2.0f*I",a[IDX2C(i,j,m)].x,
                              a[IDX2C(i,j,m)].y);
     }
    printf("\n");
   }
// define mxn matrices b,c column by column
  ind=11;                                    // b,c:
  for(j=0;j<n;j++){                          // 11,17,23,29,35
```

```
    for(i=0;i<m;i++){                          // 12,18,24,30,36
      b[IDX2C(i,j,m)].x=(float)ind;            // 13,19,25,31,37
      b[IDX2C(i,j,m)].y=0.0f;                  // 14,20,26,32,38
      c[IDX2C(i,j,m)].x=(float)ind;            // 15,21,27,33,39
      c[IDX2C(i,j,m)].y=0.0f;                  // 16,22,28,34,40
      ind++;
    }
  }
// print b(=c) row by row
  printf("b,c:\n");
   for(i=0;i<m;i++){
     for(j=0;j<n;j++){
       printf("%5.0f+%2.0f*I",b[IDX2C(i,j,m)].x,
                              b[IDX2C(i,j,m)].y);
     }
    printf("\n");
   }
// on the device
  cuComplex* d_a;                      // d_a - a on the device
  cuComplex* d_b;                      // d_b - b on the device
  cuComplex* d_c;                      // d_c - c on the device
  cudaStat=cudaMalloc((void**)&d_a,m*m*sizeof(cuComplex));
                                      //device memory alloc for a
  cudaStat=cudaMalloc((void**)&d_b,n*m*sizeof(cuComplex));
                                      //device memory alloc for b
  cudaStat=cudaMalloc((void**)&d_c,n*m*sizeof(cuComplex));
                                      //device memory alloc for c
  stat = cublasCreate(&handle);  // initialize CUBLAS context
// copy matrices from the host to the device
  stat = cublasSetMatrix(m,m,sizeof(*a),a,m,d_a,m);//a -> d_a
  stat = cublasSetMatrix(m,n,sizeof(*b),b,m,d_b,m);//b -> d_b
  stat = cublasSetMatrix(m,n,sizeof(*c),c,m,d_c,m);//c -> d_c
  cuComplex al={1.0f,0.0f};                          // al=1
  cuComplex bet={1.0f,0.0f};                        // bet=1
// Hermitian matrix-matrix multiplication:
// d_c=al*d_a*d_b+bet*d_c;
// d_a - mxm hermitian matrix; d_b,d_c - mxn-general matices;
// al,bet - scalars

  stat=cublasChemm(handle,CUBLAS_SIDE_LEFT,CUBLAS_FILL_MODE_LOWER,
                          m,n,&al,d_a,m,d_b,m,&bet,d_c,m);

  stat=cublasGetMatrix(m,n,sizeof(*c),d_c,m,c,m); // d_c -> c
  printf("c after  Chemm :\n");
   for(i=0;i<m;i++){
     for(j=0;j<n;j++){                    //print c after Chemm
       printf("%5.0f+%1.0f*I",c[IDX2C(i,j,m)].x,
                              c[IDX2C(i,j,m)].y);
     }
    printf("\n");
   }
  cudaFree(d_a);                          // free device memory
```

```
  cudaFree(d_b);                          // free device memory
  cudaFree(d_c);                          // free device memory
  cublasDestroy(handle);                // destroy CUBLAS context
  free(a);                                  // free host memory
  free(b);                                  // free host memory
  free(c);                                  // free host memory
  return EXIT_SUCCESS;
}
//lower triangle of a:
//    11+ 0*I
//    12+ 0*I    17+ 0*I
//    13+ 0*I    18+ 0*I    22+ 0*I
//    14+ 0*I    19+ 0*I    23+ 0*I    26+ 0*I
//    15+ 0*I    20+ 0*I    24+ 0*I    27+ 0*I    29+ 0*I
//    16+ 0*I    21+ 0*I    25+ 0*I    28+ 0*I    30+ 0*I    31+ 0*I

// b,c:
//    11+ 0*I    17+ 0*I    23+ 0*I    29+ 0*I    35+ 0*I
//    12+ 0*I    18+ 0*I    24+ 0*I    30+ 0*I    36+ 0*I
//    13+ 0*I    19+ 0*I    25+ 0*I    31+ 0*I    37+ 0*I
//    14+ 0*I    20+ 0*I    26+ 0*I    32+ 0*I    38+ 0*I
//    15+ 0*I    21+ 0*I    27+ 0*I    33+ 0*I    39+ 0*I
//    16+ 0*I    22+ 0*I    28+ 0*I    34+ 0*I    40+ 0*I

// c after  Chemm :
// 1122+0*I 1614+0*I 2106+0*I 2598+0*I 3090+0*I //
// 1484+0*I 2132+0*I 2780+0*I 3428+0*I 4076+0*I //
// 1740+0*I 2496+0*I 3252+0*I 4008+0*I 4764+0*I //    c=a*b+c
// 1912+0*I 2740+0*I 3568+0*I 4396+0*I 5224+0*I //
// 2025+0*I 2901+0*I 3777+0*I 4653+0*I 5529+0*I //
// 2107+0*I 3019+0*I 3931+0*I 4843+0*I 5755+0*I //
```

3.4.8 `cublasCherk` - Hermitian rank-k update

This function performs the Hermitian rank-k update

$$C = \alpha\, op(A) op(A)^H + \beta\, C,$$

where C is a Hermitian $n \times n$ matrix, $op(A)$ is an $n \times k$ matrix and α, β are scalars. The value of $op(A)$ can be equal to A in `CUBLAS_OP_N` case or A^H in `CUBLAS_OP_C` case (conjugate transposition). C can be stored in lower (`CUBLAS_FILL_MODE_LOWER`) or upper (`CUBLAS_FILL_MODE_UPPER`) mode.

```
// nvcc 043cherk.c -lcublas
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include "cublas_v2.h"
#define IDX2C(i,j,ld) (((j)*(ld))+( i ))
#define n 6                                    // c - nxn matrix
```

```
#define k 5                                     // a - nxk matrix
int main(void){
  cudaError_t cudaStat;                      // cudaMalloc status
  cublasStatus_t stat;                 // CUBLAS functions status
  cublasHandle_t handle;                       // CUBLAS context
  int i,j;                            // i-row index, j-col. index
// data preparation on the host
  cuComplex *a;              // nxk complex matrix a on the host
  cuComplex *c;              // nxn complex matrix c on the host
  a=(cuComplex*)malloc(n*k*sizeof(cuComplex)); // host memory
                                                 // alloc for a
  c=(cuComplex*)malloc(n*n*sizeof(cuComplex)); // host memory
                                                 // alloc for c
// define the lower triangle of an nxn Hermitian matrix c in
// lower mode  column by column;
  int ind=11;                                 // c:
  for(j=0;j<n;j++){                           // 11
    for(i=0;i<n;i++){                         // 12,17
      if(i>=j){                               // 13,18,22
        c[IDX2C(i,j,n)].x=(float)ind++; // 14,19,23,26
        c[IDX2C(i,j,n)].y=0.0f;         // 15,20,24,27,29
      }                                       // 16,21,25,28,30,31
    }
  }
// print the lower triangle of c row by row
  printf("lower triangle of c:\n");
   for(i=0;i<n;i++){
     for(j=0;j<n;j++){
       if(i>=j)
       printf("%5.0f+%2.0f*I",c[IDX2C(i,j,n)].x,
                              c[IDX2C(i,j,n)].y);
     }
    printf("\n");
   }
// define an nxk matrix a column by column
  ind=11;                                       // a:
  for(j=0;j<k;j++){                             // 11,17,23,29,35
    for(i=0;i<n;i++){                           // 12,18,24,30,36
      a[IDX2C(i,j,n)].x=(float)ind;             // 13,19,25,31,37
      a[IDX2C(i,j,n)].y=0.0f;                   // 14,20,26,32,38
      ind++;                                    // 15,21,27,33,39
    }                                           // 16,22,28,34,40
  }
// print a row by row
  printf("a:\n");
   for(i=0;i<n;i++){
     for(j=0;j<k;j++){
       printf("%5.0f+%2.0f*I",a[IDX2C(i,j,n)].x,
                              a[IDX2C(i,j,n)].y);
     }
    printf("\n");
   }
```

```
// on the device
  cuComplex* d_a;                     // d_a - a on the device
  cuComplex* d_c;                     // d_c - c on the device
  cudaStat=cudaMalloc((void**)&d_a,n*k*sizeof(cuComplex));
                                      //device memory alloc for a
  cudaStat=cudaMalloc((void**)&d_c,n*n*sizeof(cuComplex));
                                      //device memory alloc for c
  stat = cublasCreate(&handle);  // initialize CUBLAS context
// copy matrices from the host to the device
  stat = cublasSetMatrix(n,k,sizeof(*a),a,n,d_a,n);//a -> d_a
  stat = cublasSetMatrix(n,n,sizeof(*c),c,n,d_c,n);//c -> d_c
  float al=1.0f;                                        // al=1
  float bet=1.0f;                                       //bet=1
// rank-k update of a Hermitian matrix:
// d_c=al*d_a*d_a^H +bet*d_c
// d_c - nxn, Hermitian matrix; d_a - nxk general matrix;
// al,bet - scalars

  stat=cublasCherk(handle,CUBLAS_FILL_MODE_LOWER,CUBLAS_OP_N,
                                    n,k,&al,d_a,n,&bet,d_c,n);

  stat=cublasGetMatrix(n,n,sizeof(*c),d_c,n,c,n); // d_c -> c
  printf("lower triangle of c after Cherk :\n");
   for(i=0;i<n;i++){
     for(j=0;j<n;j++){                  // print c after Cherk
       if(i>=j)
       printf("%5.0f+%1.0f*I",c[IDX2C(i,j,n)].x,
                                c[IDX2C(i,j,n)].y);
     }
    printf("\n");
   }
  cudaFree(d_a);                          // free device memory
  cudaFree(d_c);                          // free device memory
  cublasDestroy(handle);                // destroy CUBLAS context
  free(a);                                  // free host memory
  free(c);                                  // free host memory
  return EXIT_SUCCESS;
}
// lower triangle of c:
//   11+ 0*I
//   12+ 0*I   17+ 0*I
//   13+ 0*I   18+ 0*I   22+ 0*I
//   14+ 0*I   19+ 0*I   23+ 0*I   26+ 0*I
//   15+ 0*I   20+ 0*I   24+ 0*I   27+ 0*I   29+ 0*I
//   16+ 0*I   21+ 0*I   25+ 0*I   28+ 0*I   30+ 0*I   31+ 0*I
// a:
//   11+ 0*I   17+ 0*I   23+ 0*I   29+ 0*I   35+ 0*I
//   12+ 0*I   18+ 0*I   24+ 0*I   30+ 0*I   36+ 0*I
//   13+ 0*I   19+ 0*I   25+ 0*I   31+ 0*I   37+ 0*I
//   14+ 0*I   20+ 0*I   26+ 0*I   32+ 0*I   38+ 0*I
//   15+ 0*I   21+ 0*I   27+ 0*I   33+ 0*I   39+ 0*I
```

```
//   16+ 0*I   22+ 0*I   28+ 0*I   34+ 0*I   40+ 0*I

// lower triangle of c after Cherk :
// 3016+0*I
// 3132+0*I 3257+0*I
// 3248+0*I 3378+0*I 3507+0*I                    // c=a*a^H +c
// 3364+0*I 3499+0*I 3633+0*I 3766+0*I
// 3480+0*I 3620+0*I 3759+0*I 3897+0*I 4034+0*I
// 3596+0*I 3741+0*I 3885+0*I 4028+0*I 4170+0*I 4311+0*I
```

3.4.9 cublasCher2k - Hermitian rank-2k update

This function performs the Hermitian rank-2k update

$$C = \alpha\, op(A)\, op(B)^H + \bar{\alpha}\, op(B)\, op(A)^H + \beta\, C,$$

where C is a Hermitian $n \times n$ matrix, $op(A), op(B)$ are $n \times k$ matrices and α, β are scalars. The value of $op(A)$ can be equal to A in CUBLAS_OP_N case or A^H (conjugate transposition) in CUBLAS_OP_C case and similarly for $op(B)$. C can be stored in lower (CUBLAS_FILL_MODE_LOWER) or upper (CUBLAS_FILL_MODE_UPPER) mode.

```
// nvcc 044cher2k.c -lcublas
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include "cublas_v2.h"
#define IDX2C(i,j,ld) (((j)*(ld))+( i ))
#define n 6                                    // c - nxn matrix
#define k 5                                   // a,b - nxk matrix
int main(void){
  cudaError_t cudaStat;                      // cudaMalloc status
  cublasStatus_t stat;                // CUBLAS functions status
  cublasHandle_t handle;                       // CUBLAS context
  int i,j;                              // i-row index, j-col. ind.
// data preparation on the host
  cuComplex *a;             // nxk complex matrix a on the host
  cuComplex *b;             // nxk complex matrix a on the host
  cuComplex *c;             // nxn complex matrix c on the host
  a=(cuComplex*)malloc(n*k*sizeof(cuComplex))  // host memory
                                               // alloc for a
  b=(cuComplex*)malloc(n*k*sizeof(cuComplex)); // host memory
                                               // alloc for b
  c=(cuComplex*)malloc(n*n*sizeof(cuComplex)); // host memory
                                               // alloc for c
// define the lower triangle of an nxn Hermitian matrix c in
// lower mode  column by column
  int ind=11;                                  // c:
  for(j=0;j<n;j++){                            // 11
    for(i=0;i<n;i++){                          // 12 17
```

```
      if(i>=j){                              // 13,18,22
        c[IDX2C(i,j,n)].x=(float)ind;        // 14,19,23,26
        c[IDX2C(i,j,n)].y=0.0f;              // 15,20,24,27,29
        ind++;                               // 16,21,25,28,30,31
      }
    }
  }
// print the lower triangle of c row by row
  printf("lower triangle of c:\n");
   for(i=0;i<n;i++){
     for(j=0;j<n;j++){
       if(i>=j)
       printf("%5.0f+%2.0f*I",c[IDX2C(i,j,n)].x,
                              c[IDX2C(i,j,n)].y);
     }
     printf("\n");
   }
// define nxk matrices a,b column by column
  ind=11;                                    // a,b:
  for(j=0;j<k;j++){                          // 11,17,23,29,35
    for(i=0;i<n;i++){                        // 12,18,24,30,36
      a[IDX2C(i,j,n)].x=(float)ind;          // 13,19,25,31,37
      a[IDX2C(i,j,n)].y=0.0f;                // 14,20,26,32,38
      b[IDX2C(i,j,n)].x=(float)ind++;        // 15,21,27,33,39
      b[IDX2C(i,j,n)].y=0.0f;                // 16,22,28,34,40
    }
  }
// print a(=b) row by row
  printf("a,b:\n");
   for(i=0;i<n;i++){
     for(j=0;j<k;j++){
       printf("%5.0f+%2.0f*I",a[IDX2C(i,j,n)].x,
                              a[IDX2C(i,j,n)].y);
     }
     printf("\n");
   }
// on the device
  cuComplex* d_a;                    // d_a - a on the device
  cuComplex* d_b;                    // d_b - b on the device
  cuComplex* d_c;                    // d_c - c on the device
  cudaStat=cudaMalloc((void**)&d_a,n*k*sizeof(cuComplex));
                                     //device memory alloc for a
  cudaStat=cudaMalloc((void**)&d_b,n*k*sizeof(cuComplex));
                                     //device memory alloc for b
  cudaStat=cudaMalloc((void**)&d_c,n*n*sizeof(cuComplex));
                                     //device memory alloc for c
  stat = cublasCreate(&handle);  // initialize CUBLAS context
  stat = cublasSetMatrix(n,k,sizeof(*a),a,n,d_a,n);//a -> d_a
  stat = cublasSetMatrix(n,k,sizeof(*a),b,n,d_b,n);//b -> d_b
  stat = cublasSetMatrix(n,n,sizeof(*c),c,n,d_c,n);//c -> d_c
  cuComplex al={1.0f,0.0f};                          // al=1
  float bet=1.0f;                                    //bet=1
```

```
// Hermitian rank-2k update:
// d_c=al*d_a*d_b^H+\bar{al}*d_b*a^H + bet*d_c
// d_c - nxn, hermitian matrix; d_a,d_b -nxk general matrices;
// al,bet - scalars

  stat=cublasCher2k(handle,CUBLAS_FILL_MODE_LOWER,CUBLAS_OP_N,
                              n,k,&al,d_a,n,d_b,n,&bet,d_c,n);

  stat=cublasGetMatrix(n,n,sizeof(*c),d_c,n,c,n); //d_c -> c
// print the updated lower triangle of c row by row
  printf("lower triangle of c after Cher2k :\n");
   for(i=0;i<n;i++){
     for(j=0;j<n;j++){                  //print c after Cher2k
       if(i>=j)
       printf("%6.0f+%2.0f*I",c[IDX2C(i,j,n)].x,
                              c[IDX2C(i,j,n)].y);
     }
    printf("\n");
   }
  cudaFree(d_a);                          // free device memory
  cudaFree(d_b);                          // free device memory
  cudaFree(d_c);                          // free device memory
  cublasDestroy(handle);                // destroy CUBLAS context
  free(a);                                  // free host memory
  free(b);                                  // free host memory
  free(c);                                  // free host memory
  return EXIT_SUCCESS;
}
// lower triangle of c:
//    11+ 0*I
//    12+ 0*I    17+ 0*I
//    13+ 0*I    18+ 0*I    22+ 0*I
//    14+ 0*I    19+ 0*I    23+ 0*I    26+ 0*I
//    15+ 0*I    20+ 0*I    24+ 0*I    27+ 0*I    29+ 0*I
//    16+ 0*I    21+ 0*I    25+ 0*I    28+ 0*I    30+ 0*I    31+ 0*I
// a,b:
//    11+ 0*I    17+ 0*I    23+ 0*I    29+ 0*I    35+ 0*I
//    12+ 0*I    18+ 0*I    24+ 0*I    30+ 0*I    36+ 0*I
//    13+ 0*I    19+ 0*I    25+ 0*I    31+ 0*I    37+ 0*I
//    14+ 0*I    20+ 0*I    26+ 0*I    32+ 0*I    38+ 0*I
//    15+ 0*I    21+ 0*I    27+ 0*I    33+ 0*I    39+ 0*I
//    16+ 0*I    22+ 0*I    28+ 0*I    34+ 0*I    40+ 0*I

// lower triangle of c after Cher2k : c = a*b^H + b*a^H + c
//  6021+0*I
//  6252+0*I  6497+0*I
//  6483+0*I  6738+0*I  6992+0*I
//  6714+0*I  6979+0*I  7243+0*I  7506+0*I
//  6945+0*I  7220+0*I  7494+0*I  7767+0*I  8039+0*I
//  7176+0*I  7461+0*I  7745+0*I  8028+0*I  8310+0*I  8591+0*I
```

Chapter 4

MAGMA by example

4.1 General remarks on Magma

MAGMA is an abbreviation for Matrix Algebra for GPU and Multicore Architectures (http://icl.cs.utk.edu/magma/). It is a collection of dense linear algebra routines, a successor of Lapack and ScaLapack, specially developed for heterogeneous GPU-based architectures.
Magma is an open-source project developed by Innovative Computing Laboratory (ICL), University of Tennessee, Knoxville, USA.
It includes

- LU, QR and Cholesky factorization.
- Hessenberg reduction.
- Linear solvers based on LU, QR and Cholesky decompositions.
- Eigenvalue and singular value problem solvers.
- Generalized Hermitian-definite eigenproblem solver.
- Mixed-precision iterative refinement solvers based on LU, QR and Cholesky factorizations.

A more detailed (but not complete) information on procedures contained in Magma can be found in table of contents. A complete information can be found for example in `magma-X.Y.Z/src` directory. Let us notice that the source files in this directory contain a precise syntax description of Magma functions, so we do not repeat this information in our text (the syntax is also easily available on the Internet). Instead, we present a series of examples how to use the library.
All subprograms have four versions corresponding to four data types

- `s - float` – real single-precision
- `d - double` – real double-precision,
- `c - magmaFloatComplex` – complex single-precision,
- `z - magmaDoubleComplex` – complex double-precision.

For example `magma_i<t>amax` is a template which can represent `magma_isamax, magma_idamax, magma_icamax` or `magma_izamax`.

- We shall restrict our examples to the most popular real, single and double precision versions. The single precision versions are important because in users hands there are millions of inexpensive GPUs which have restricted double precision capabilities. Installing Magma on such devices can be a good starting point to more advanced studies. On the other hand in many applications the double precision is necessary, so we have decided to present our examples in both versions (in Magma BLAS case only in single precision). In most examples we measure the computations times, so one can compare the performance in both precisions.
- Ideally we should check for errors on every function call. Unfortunately such an approach doubles the length of our sample codes (which are as short as possible by design). Since our set of Magma sample code (without error checking) is almost 140 pages long we have decided to ignore the error checking and to focus on the explanations which cannot be found in the syntax description.
- To obtain more compact explanations in our examples we restrict the full generality of Magma to the special case where the leading dimension of matrices is equal to the number of rows and the stride between consecutive elements of vectors is equal to 1. Magma allows for more flexible approach giving the user the access to submatrices an subvectors. The corresponding generalizations can be found in syntax descritions in source files.

4.1.1 Remarks on installation and compilation

Magma can be downloaded from http://icl.cs.utk.edu/magma/software/index.html. In the Magma directory obtained after extraction of the downloaded `magma-X.Y.Z.tar.gz` file there is `README` file which contains installation instructions. The user must provide `make.inc` which specifies where CUDA, BLAS and Lapack are installed in the system. Some sample `make.inc` files are contained in Magma directory. After proper modification of the `make.inc` file, running

`$make`
creates `libmagma.a` in Magma `lib` subdirectory and testing drivers in `testing` directory.
An easy way of compiling examples from our text is to copy source file, for example `testing_001example.cpp` to `testing` directory, add appropriate name `testing_001example` at the end of `testing/Makefile.src` file and change directory to `testing`. Running
`$make`
in this directory should give a new executable `testing_001example`.

4.1.2 Remarks on hardware used in examples

In most examples we have measured the computations times. The times were obtained on the machine with Centos 6.4, CUDA 5.5, magma-1.4.0 compiled with MKL library and

- two socket Xeon CPU E5-2665, 2.40 GHz,
- two Tesla K20m GPUs.

4.2 Magma BLAS

Magma version of BLAS is not as exhaustive as CUBLAS. We restrict ourselves to presentation of the following subset of Magma BLAS single precision functions.

Leve1 1 BLAS : `magma_isamax`, `magma_sswap`,

Leve1 2 BLAS : `magma_sgemv`, `magma_ssymv`,

Leve1 3 BLAS : `magma_sgemm`, `magma_ssymm`, `magma_ssyrk`, `magma_ssyr2k`, `magma_strmm`, `magma_sgeadd`.

4.2.1 `magma_isamax` - find element with maximal absolute value

This functions finds the smallest index of the element of an array with the maximum magnitude.

```
#include <stdlib.h>
#include <stdio.h>
#include "magma.h"
int main( int argc, char** argv ){
  magma_init();                              // initialize Magma
  magma_int_t m = 1024;                          // length of a
  float *a;                          // a - m-vector on the host
  float *d_a;                // d_a - m-vector a on the device
  magma_err_t err;
```

```
// allocate the vector on the host
  err = magma_smalloc_cpu( &a , m );      // host memory for a
// allocate the vector on the device
  err = magma_smalloc( &d_a,  m );      // device memory for a
                              // a={sin(0),sin(1),...,sin(m-1)}
  for(int j=0;j<m;j++) a[j]=sin((float)j);
// copy data from host to device
  magma_ssetvector( m, a, 1, d_a, 1 );        // copy a -> d_a
// find the smallest index of the element of d_a with maximum
// absolute value

  int i = magma_isamax( m, d_a, 1 );

  printf("max |a[i]|: %f\n",fabs(a[i-1]));
  printf("fortran index: %d\n",i);
  free(a);                                    // free host memory
  magma_free(d_a);                           // free device memory
  magma_finalize();                             // finalize Magma
  return 0;
}
// max |a[i]|: 0.999990
//
// fortran index: 700
```

4.2.2 magma_sswap - vectors swapping

This function interchanges the elements of vectors a and b:

$$a \leftarrow b, \quad b \leftarrow a.$$

```
#include <stdlib.h>
#include <stdio.h>
#include "magma.h"
int main( int argc, char** argv ){
  magma_init();                                 // initialize Magma
  magma_int_t m = 1024;                              // length of a
  float *a;                           // a - m-vector on the host
  float *b;                           // b - m-vector on the host
  float *d_a;                 // d_a - m-vector a on the device
  float *d_b;                 // d_b - m-vector a on the device
  magma_err_t err;
// allocate the vectors on the host
  err = magma_smalloc_cpu( &a , m );        // host mem. for a
  err = magma_smalloc_cpu( &b , m );        // host mem. for b
// allocate the vector on the device
  err = magma_smalloc( &d_a,  m );      // device memory for a
  err = magma_smalloc( &d_b,  m );      // device memory for b
                              // a={sin(0),sin(1),...,sin(m-1)}
  for(int j=0;j<m;j++) a[j]=sin((float)j);
                              // b={cos(0),cos(1),...,cos(m-1)}
  for(int j=0;j<m;j++) b[j]=cos((float)j);
```

```
  printf("a: ");
  for(int j=0;j<4;j++) printf("%6.4f,",a[j]);printf("...\n");
  printf("b: ");
  for(int j=0;j<4;j++) printf("%6.4f,",b[j]);printf("...\n");
// copy data from host to device
  magma_ssetvector( m, a, 1, d_a, 1 );        // copy a -> d_a
  magma_ssetvector( m, b, 1, d_b, 1 );        // copy b -> d_b
// swap the vectors

  magma_sswap( m, d_a, 1, d_b, 1 );

  magma_sgetvector( m, d_a, 1, a, 1 );        // copy d_a -> a
  magma_sgetvector( m, d_b, 1, b, 1 );        // copy d_b -> b
  printf("after magma_sswap:\n");
  printf("a: ");
  for(int j=0;j<4;j++) printf("%6.4f,",a[j]);printf("...\n");
  printf("b: ");
  for(int j=0;j<4;j++) printf("%6.4f,",b[j]);printf("...\n");
  free(a);                                    // free host memory
  free(b);                                    // free host memory
  magma_free(d_a);                           // free device memory
  magma_free(d_b);                           // free device memory
  magma_finalize();                             // finalize Magma
  return 0;
}
// a: 0.0000,0.8415,0.9093,0.1411,...
// b: 1.0000,0.5403,-0.4161,-0.9900,...
//
// after magma_sswap:
//
// a: 1.0000,0.5403,-0.4161,-0.9900,...
// b: 0.0000,0.8415,0.9093,0.1411,...
```

4.2.3 magma_sgemv - matrix-vector multiplication

This function performs matrix-vector multiplication

$$c = \alpha\ op(A)b + \beta c,$$

where A is a matrix, b, c are vectors, α, β are scalars and $op(A)$ can be equal to A (`MagmaNoTrans,'N'` case), A^T (transposition) in `MagmaTrans,'T'` case or A^H (conjugate transposition) in `MagmaConjTrans,'C'` case.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
    magma_init();                             // initialize Magma
    magma_timestr_t  start, end;
```

```
    float   gpu_time;
    magma_int_t m = 4096;               // number of rows of a
    magma_int_t n = 2048;            // number of columns of a
    magma_int_t mn=m*n;                          // size of a
    float *a;                      // a- mxn matrix on the host
    float *b;                        // b- n-vector on the host
    float *c,*c2;                // c,c2- m-vectors on the host
    float *d_a;              // d_a- mxn matrix a on the device
    float *d_b;                // d_b- n-vector b on the device
    float *d_c;                  //d_c -m-vector on the device
    float alpha = MAGMA_S_MAKE( 1.0, 0.0 );         // alpha=1
    float beta  = MAGMA_S_MAKE( 1.0, 0.0 );          // beta=1
    magma_int_t ione = 1;
    magma_int_t ISEED[4] = { 0,1,2,3 };                // seed
    magma_err_t err;
// allocate matrix and vectors on the host
    err = magma_smalloc_pinned( &a , m*n ); // host mem. for a
    err = magma_smalloc_pinned( &b , n );   // host mem. for b
    err = magma_smalloc_pinned( &c , m );   // host mem. for c
    err = magma_smalloc_pinned( &c2, m );  // host mem. for c2
// allocate matrix and vectors on the device
    err = magma_smalloc( &d_a,  m*n );  // device memory for a
    err = magma_smalloc( &d_b,  n );    // device memory for b
    err = magma_smalloc( &d_c , m );    // device memory for c
// generate random matrix a and vectors b,c
    lapackf77_slarnv(&ione,ISEED,&mn,a);           // random a
    lapackf77_slarnv(&ione,ISEED,&n,b);            // random b
    lapackf77_slarnv(&ione,ISEED,&m,c);            // random c
// copy data from host to device
    magma_ssetmatrix( m, n, a,  m, d_a,  m ); // copy a -> d_a
    magma_ssetvector( n, b, 1, d_b,  1 );     // copy b -> d_b
    magma_ssetvector( m, c, 1, d_c, 1 );      // copy c -> d_c
// matrix-vector multiplication:
// d_c = alpha*d_a*d_b + beta*d_c;
// d_a- mxn matrix; b -n -vector; c -m -vector
    start = get_current_time();

  magma_sgemv(MagmaNoTrans,m,n,alpha,d_a,m,d_b,1,beta,d_c,1);

    end = get_current_time();
    gpu_time=GetTimerValue(start,end)/1e3;
    printf("magma_sgemv time: %7.5f sec.\n",gpu_time);
// copy data from device to host
    magma_sgetvector( m, d_c, 1, c2, 1 );     // copy d_c ->c2
    printf("after magma_sgemv:\n");
    printf("c2: ");
    for(int j=0;j<4;j++) printf("%9.4f,",c2[j]);
    printf("...\n");
    magma_free_pinned(a);                  // free host memory
    magma_free_pinned(b);                  // free host memory
    magma_free_pinned(c);                  // free host memory
    magma_free_pinned(c2);                 // free host memory
```

```
    magma_free(d_a);                        // free device memory
    magma_free(d_b);                        // free device memory
    magma_free(d_c);                        // free device memory
    magma_finalize();                           // finalize Magma
    return 0;
}
// magma_sgemv time: 0.00087 sec.
//
// after magma_sgemv:
// c2:  507.9389, 498.1867, 503.1055, 508.1643,...
```

4.2.4 magma_ssymv - symmetric matrix-vector multiplication

This function performs the symmetric matrix-vector multiplication.

$$c = \alpha Ab + \beta c,$$

where A is an $m \times m$ symmetric matrix, b, c are vectors and α, β are scalars. The matrix A can be stored in lower (`MagmaLower,'L'`) or upper (`MagmaUpper,'U'`) mode.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                                 // initialize Magma
  magma_timestr_t  start, end;
  float   gpu_time;
  magma_int_t m = 4096;     // number of rows and columns of a
  magma_int_t mm=m*m;                               // size of a
  float *a;                         // a- mxm matrix on the host
// lower triangular part of a contains the lower triangular
// part of some symmetric matrix
  float *b;                           // b- m-vector on the host
  float *c,*c2;                 // c,c2- m-vectors on the host
  float *d_a;               // d_a- mxm matrix a on the device
  float *d_b;                 // d_b- m-vector b on the device
  float *d_c;                   //d_c -m-vector on the device
  float alpha = MAGMA_S_MAKE( 1.0, 0.0 );           // alpha=1
  float beta  = MAGMA_S_MAKE( 1.0, 0.0 );            // beta=1
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = { 0,1,2,3 };                 // seed
  magma_err_t err;
// allocate matrix and vectors on the host
  err = magma_smalloc_pinned( &a , mm );    // host mem. for a
  err = magma_smalloc_pinned( &b , m );     // host mem. for b
  err = magma_smalloc_pinned( &c , m );     // host mem. for c
  err = magma_smalloc_pinned( &c2, m );    // host mem. for c2
// allocate matrix and vectors on the device
```

```
  err = magma_smalloc( &d_a,  mm );      // device memory for a
  err = magma_smalloc( &d_b,  m );       // device memory for b
  err = magma_smalloc( &d_c , m );       // device memory for c
// generate random matrix a and vectors b,c; only the lower
// triangular part of a is to be referenced
  lapackf77_slarnv(&ione,ISEED,&mm,a);              // random a
  lapackf77_slarnv(&ione,ISEED,&m,b);               // random b
  lapackf77_slarnv(&ione,ISEED,&m,c);               // random c
// copy data from host to device
  magma_ssetmatrix( m, m, a,  m, d_a,  m );   // copy a -> d_a
  magma_ssetvector( m, b, 1, d_b,  1 );       // copy b -> d_b
  magma_ssetvector( m, c, 1, d_c, 1 );        // copy c -> d_c
// matrix-vector multiplication:
// d_c = alpha*d_a*d_b + beta*d_c;
// d_a- mxm matrix; b -m -vector; c -m -vector
  start = get_current_time();

  magma_ssymv(MagmaLower,m,alpha,d_a,m,d_b,1,beta,d_c,1);

  end = get_current_time();
  gpu_time=GetTimerValue(start,end)/1e3;
  printf("magma_ssymv time: %7.5f sec.\n",gpu_time);
// copy data from device to host
  magma_sgetvector( m, d_c, 1, c2, 1 );       // copy d_c ->c2
  printf("after magma_ssymv:\n");
  printf("c2: ");
  for(int j=0;j<4;j++) printf("%10.4f,",c2[j]);
  printf("...\n");
  magma_free_pinned(a);                     // free host memory
  magma_free_pinned(b);                     // free host memory
  magma_free_pinned(c);                     // free host memory
  magma_free_pinned(c2);                    // free host memory
  magma_free(d_a);                         // free device memory
  magma_free(d_b);                         // free device memory
  magma_free(d_c);                         // free device memory
  magma_finalize();                           // finalize Magma
  return 0;
}
// magma_ssymv time: 0.00140 sec.
//
// after magma_ssymv:
// c2:  1003.9608, 1029.2787, 1008.7328, 1042.9585,...
```

4.2.5 magma_sgemm - matrix-matrix multiplication

This function performs the matrix-matrix multiplication

$$C = \alpha op(A) op(B) + \beta C,$$

where A, B, C are matrices and α, β are scalars. The value of $op(A)$ can be equal to A (`MagmaNoTrans,'N'` case), A^T (transposition) in `MagmaTrans,'T'`

case, or A^H (conjugate transposition) in `MagmaConjTrans,'C'` case and similarly for $op(B)$.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                                  // initialize Magma
  magma_timestr_t  start, end;
  float   gpu_time;
  magma_int_t m = 8192;                           // a - mxk matrix
  magma_int_t n = 4096;                           // b - kxn matrix
  magma_int_t k = 2048;                           // c - mxn matrix
  magma_int_t mk=m*k;                                  // size of a
  magma_int_t kn=k*n;                                  // size of b
  magma_int_t mn=m*n;                                  // size of c
  float *a;                          // a- mxk matrix on the host
  float *b;                          // b- kxn matrix on the host
  float *c;                          // c- mxn matrix on the host
  float *d_a;                // d_a- mxk matrix a on the device
  float *d_b;                // d_b- kxn matrix b on the device
  float *d_c;                // d_c- mxn matrix c on the device
  float alpha = MAGMA_S_MAKE( 1.0, 0.0 );            // alpha=1
  float beta  = MAGMA_S_MAKE( 1.0, 0.0 );             // beta=1
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = { 0,1,2,3 };                   // seed
  magma_err_t err;
// allocate matrices on the host
  err = magma_smalloc_pinned( &a , mk );     // host mem. for a
  err = magma_smalloc_pinned( &b , kn );     // host mem. for b
  err = magma_smalloc_pinned( &c , mn );     // host mem. for c
// allocate matrix and vectors on the device
  err = magma_smalloc( &d_a,  mk );      // device memory for a
  err = magma_smalloc( &d_b,  kn );      // device memory for b
  err = magma_smalloc( &d_c,  mn );      // device memory for c
// generate random matrices a, b, c;
  lapackf77_slarnv(&ione,ISEED,&mk,a);              // random a
  lapackf77_slarnv(&ione,ISEED,&kn,b);              // random b
  lapackf77_slarnv(&ione,ISEED,&mn,c);              // random c
// copy data from host to device
  magma_ssetmatrix( m, k, a,  m, d_a,  m );    // copy a -> d_a
  magma_ssetmatrix( k, n, b,  k, d_b,  k );    // copy b -> d_b
  magma_ssetmatrix( m, n, c,  m, d_c,  m );    // copy c -> d_c
// matrix-matrix multiplication: d_c = al*d_a*d_b + bet*d_c
// d_a -mxk matrix, d_b -kxn matrix, d_c -mxn matrix;
// al,bet - scalars
  start = get_current_time();

  magma_sgemm(MagmaNoTrans,MagmaNoTrans,m,n,k,alpha,d_a,m,d_b,k,
                                                    beta,d_c,m);
```

```
  end = get_current_time();
  gpu_time=GetTimerValue(start,end)/1e3;
  printf("magma_sgemm time: %7.5f sec.\n",gpu_time);
// copy data from device to host
  magma_sgetmatrix( m, n, d_c, m, c, m );      // copy d_c -> c
  printf("after magma_sgemm:\n");
  printf("c:\n");
  for(int i=0;i<4;i++){
  for(int j=0;j<4;j++) printf("%10.4f,",c[i*m+j]);
  printf("...\n");}
  printf(".....................................\n");
  magma_free_pinned(a);                        // free host memory
  magma_free_pinned(b);                        // free host memory
  magma_free_pinned(c);                        // free host memory
  magma_free(d_a);                            // free device memory
  magma_free(d_b);                            // free device memory
  magma_free(d_c);                            // free device memory
  magma_finalize();                               // finalize Magma
  return 0;
}
// magma_sgemm time: 0.05517 sec.
//
// after magma_sgemm:
// c:
//    498.3723,   521.3933,   507.0844,   515.5119,...
//    504.1406,   517.1718,   509.3519,   511.3415,...
//    511.1694,   530.6165,   517.5001,   524.9462,...
//    505.5946,   522.4631,   511.7729,   516.2770,...
//    ........................................
```

4.2.6 magma_ssymm - symmetric matrix-matrix multiplication

This function performs the left or right symmetric matrix-matrix multiplications

$$C = \alpha AB + \beta C \qquad \text{in } \texttt{MagmaLeft,'L'} \text{ case,}$$
$$C = \alpha BA + \beta C \qquad \text{in } \texttt{MagmaRight,'R'} \text{ case.}$$

The symmetric matrix A has dimension $m \times m$ in the first case and $n \times n$ in the second one. The general matrices B, C have dimensions $m \times n$ and α, β are scalars. The matrix A can be stored in lower (`MagmaLower,'L'`) or upper (`MagmaUpper,'U'`) mode.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                               // initialize Magma
```

```
  magma_timestr_t  start, end;
  float   gpu_time;
  magma_int_t info;
  magma_int_t m = 8192;                       // a - mxm matrix
  magma_int_t n = 4096;                    // b,c - mxn matrices
  magma_int_t mm=m*m;                             // size of a
  magma_int_t mn=m*n;                          // size of b,c
  float *a;                        // a- mxm matrix on the host
  float *b;                        // b- mxn matrix on the host
  float *c;                        // c- mxn matrix on the host
  float *d_a;                // d_a- mxm matrix a on the device
  float *d_b;                // d_b- mxn matrix b on the device
  float *d_c;                // d_c- mxn matrix c on the device
  float alpha = MAGMA_S_MAKE( 1.0, 0.0 );           // alpha=1
  float beta  = MAGMA_S_MAKE( 1.0, 0.0 );            // beta=1
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = { 0,1,2,3 };                  // seed
  magma_err_t err;
// allocate matrices on the host
  err = magma_smalloc_pinned( &a , mm );  // host memory for a
  err = magma_smalloc_pinned( &b , mn );  // host memory for b
  err = magma_smalloc_pinned( &c , mn );  // host memory for c
// allocate matrix and vectors on the device
  err = magma_smalloc( &d_a,  mm );     // device memory for a
  err = magma_smalloc( &d_b,  mn );     // device memory for b
  err = magma_smalloc( &d_c,  mn );     // device memory for c
// generate random matrices a, b, c;
  lapackf77_slarnv(&ione,ISEED,&mm,a);             // random a
// lower triangular part of a is the lower triangular part
// of some symmetric matrix, the strictly upper triangular
// part of a is not referenced
  lapackf77_slarnv(&ione,ISEED,&mn,b);             // random b
  lapackf77_slarnv(&ione,ISEED,&mn,c);             // random c
// copy data from host to device
  magma_ssetmatrix( m, m, a,  m, d_a,  m );   // copy a -> d_a
  magma_ssetmatrix( m, n, b,  m, d_b,  m );   // copy b -> d_b
  magma_ssetmatrix( m, n, c,  m, d_c,  m );   // copy c -> d_c
// symmetric matrix-matrix multiplication:
// d_c = al*d_a*d_b + bet*d_c
// d_a -mxm symmetric matrix, d_b, d_c -mxn matrices;
// al,bet - scalars
  start = get_current_time();

  magma_ssymm(MagmaLeft,MagmaLower,m,n,alpha,d_a,m,d_b,m,beta,
                                                    d_c,m);

  end = get_current_time();
  gpu_time=GetTimerValue(start,end)/1e3;
  printf("magma_ssymm time: %7.5f sec.\n",gpu_time);
// copy data from device to host
  magma_sgetmatrix( m, n, d_c, m, c, m );     // copy d_c -> c
  printf("after magma_ssymm:\n");
```

```
  printf("c:\n");
  for(int i=0;i<4;i++){
  for(int j=0;j<4;j++) printf("%10.4f,",c[i*m+j]);
  printf("...\n");}
  printf("..........................................\n");
  magma_free_pinned(a);                    // free host memory
  magma_free_pinned(b);                    // free host memory
  magma_free_pinned(c);                    // free host memory
  magma_free(d_a);                        // free device memory
  magma_free(d_b);                        // free device memory
  magma_free(d_c);                        // free device memory
  magma_finalize();                          // finalize Magma
  return 0;
}
// magma_ssymm time: 0.30387 sec.
//
// after magma_ssymm:
// c:
//  2021.3813, 2045.4391, 2048.6992, 2019.2108,...
//  2037.0027, 2050.8364, 2047.5414, 2031.6824,...
//  2053.6797, 2084.0029, 2077.5017, 2068.3191,...
//  2023.3381, 2045.9785, 2051.4314, 2013.8231,...
//  ..........................................
```

4.2.7 magma_ssyrk - symmetric rank-k update

This function performs the symmetric rank-k update

$$C = \alpha\, op(A) op(A)^T + \beta C,$$

where $op(A)$ is an $m \times k$ matrix, C is a symmetric $m \times m$ matrix stored in lower (`MagmaLower,'L'`) or upper (`MagmaUpper,'U'`) mode and α, β are scalars. The value of $op(A)$ can be equal to A in `MagmaNoTrans,'N'` case or A^T (transposition) in `MagmaTrans,'T'` case.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                            // initialize Magma
  magma_timestr_t  start, end;
  float   gpu_time;
  magma_int_t info;
  magma_int_t m = 8192;                      // a - mxk matrix
  magma_int_t k = 4096;                      // c - mxm matrix
  magma_int_t mm=m*m;                           // size of c
  magma_int_t mk=m*k;                           // size of a
  float *a;                       // a- mxk matrix on the host
  float *c;                       // c- mxm matrix on the host
```

```
  float *d_a;               // d_a- mxk matrix a on the device
  float *d_c;               // d_c- mxm matrix c on the device
  float alpha = 1.0;                                  // alpha=1
  float beta  = 1.0;                                   // beta=1
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = { 0,1,2,3 };                    // seed
  magma_err_t err;
// allocate matrices on the host
  err = magma_smalloc_pinned( &a , mk );  // host memory for a
  err = magma_smalloc_pinned( &c , mm );  // host memory for c
// allocate matrix and vectors on the device
  err = magma_smalloc( &d_a,  mk );     // device memory for a
  err = magma_smalloc( &d_c,  mm );     // device memory for c
// generate random matrices a, c;
  lapackf77_slarnv(&ione,ISEED,&mk,a);              // random a
  lapackf77_slarnv(&ione,ISEED,&mm,c);              // random c
// lower triangular part of c is the lower triangular part
// of some symmetric matrix, the strictly upper triangular
// part of c is not referenced
// copy data from host to device
  magma_ssetmatrix( m, k, a,  m, d_a,  m );   // copy a -> d_a
  magma_ssetmatrix( m, m, c,  m, d_c,  m );   // copy c -> d_c
// symmetric rank-k update:  d_c=alpha*d_a*d_a^T+beta*d_c
// d_c -mxm symmetric matrix,  d_a -mxk matrix;
// alpha,beta - scalars
  start = get_current_time();

  magma_ssyrk(MagmaUpper,MagmaNoTrans,m,k,alpha,d_a,m,beta,d_c,m);

  end = get_current_time();
  gpu_time=GetTimerValue(start,end)/1e3;
  printf("magma_ssyrk time: %7.5f sec.\n",gpu_time);
// copy data from device to host
  magma_sgetmatrix( m, m, d_c, m, c, m );     // copy d_c -> c
  printf("after magma_ssyrk:\n");
  printf("c:\n");
  for(int i=0;i<4;i++){
  for(int j=0;j<4;j++) if(i>=j) printf("%10.4f,",c[i*m+j]);
  printf("...\n");}
  printf(".....................................\n");
  magma_free_pinned(a);                     // free host memory
  magma_free_pinned(c);                     // free host memory
  magma_free(d_a);                        // free device memory
  magma_free(d_c);                        // free device memory
  magma_finalize();                          // finalize Magma
  return 0;
}
// magma_ssyrk time: 0.10996 sec.
//
// after magma_ssyrk:
// c:
//  1358.9562,...
```

```
//   1027.0094, 1382.1946,...
//   1011.2416, 1022.4153, 1351.7262,...
//   1021.8580, 1037.6437, 1025.0333, 1376.4917,...
//   ........................................
```

4.2.8 magma_ssyr2k - symmetric rank-2k update

This function performs the symmetric rank-2k update

$$C = \alpha(op(A)op(B)^T + op(B)op(A)^T) + \beta C,$$

where $op(A), op(B)$ are $m \times k$ matrices, C is a symmetric $m \times m$ matrix stored in lower (`MagmaLower,'L'`) or upper (`MagmaUpper,'U'`) mode and α, β are scalars. The value of $op(A)$ can be equal to A in `MagmaNoTrans,'N'` case or A^T (transposition) in `MagmaTrans,'T'` case and similarly for $op(B)$.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                              // initialize Magma
  magma_timestr_t  start, end;
  float   gpu_time;
  magma_int_t info;
  magma_int_t m = 8192;                    // a,b - mxk matrices
  magma_int_t k = 4096;                        // c - mxm matrix
  magma_int_t mm=m*m;                               // size of c
  magma_int_t mk=m*k;                               // size of a
  float *a;                          // a- mxk matrix on the host
  float *b;                          // b- mxk matrix on the host
  float *c;                          // c- mxm matrix on the host
  float *d_a;                // d_a- mxk matrix a on the device
  float *d_b;                // d_b- mxk matrix a on the device
  float *d_c;                // d_c- mxm matrix c on the device
  float alpha = 1.0;                                 // alpha=1
  float beta  = 1.0;                                  // beta=1
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = { 0,1,2,3 };                   // seed
  magma_err_t err;
// allocate matrices on the host
  err = magma_smalloc_pinned( &a , mk );  // host memory for a
  err = magma_smalloc_pinned( &b , mk );  // host memory for b
  err = magma_smalloc_pinned( &c , mm );  // host memory for c
// allocate matrix and vectors on the device
  err = magma_smalloc( &d_a,  mk );     // device memory for a
  err = magma_smalloc( &d_b,  mk );     // device memory for b
  err = magma_smalloc( &d_c,  mm );     // device memory for c
// generate random matrices a,b,c;
```

```
  lapackf77_slarnv(&ione,ISEED,&mk,a);              // random a
  lapackf77_slarnv(&ione,ISEED,&mk,b);              // random b
  lapackf77_slarnv(&ione,ISEED,&mm,c);              // random c
// lower triangular part of c is the lower triangular part
// of some symmetric matrix, the strictly upper triangular
// part of c is not referenced
// copy data from host to device
  magma_ssetmatrix( m, k, a,  m, d_a,  m );    // copy a -> d_a
  magma_ssetmatrix( m, k, a,  m, d_b,  m );    // copy b -> d_b
  magma_ssetmatrix( m, m, c,  m, d_c,  m );    // copy c -> d_c
// symmetric rank-2k update:
// d_c=alpha*d_a*d_b^T+\bar alpha d_b*d_a^T+beta*d_c
// d_c -mxm symmetric matrix,  d_a,d_b -mxk matrices;
// alpha,beta - scalars
  start = get_current_time();

  magma_ssyr2k(MagmaUpper,MagmaNoTrans,m,k,alpha,d_a,m,d_b,m,
                                              beta,d_c,m);

  end = get_current_time();
  gpu_time=GetTimerValue(start,end)/1e3;
  printf("magma_ssyr2k time: %7.5f sec.\n",gpu_time);
// copy data from device to host
  magma_sgetmatrix( m, m, d_c, m, c, m );      // copy d_c -> c
  printf("after magma_ssyr2k:\n");
  printf("c:\n");
  for(int i=0;i<4;i++){
  for(int j=0;j<4;j++) if(i>=j) printf("%10.4f,",c[i*m+j]);
  printf("...\n");}
  printf("..........................................\n");
  magma_free_pinned(a);                       // free host memory
  magma_free_pinned(c);                       // free host memory
  magma_free(d_a);                           // free device memory
  magma_free(d_c);                           // free device memory
  magma_finalize();                             // finalize Magma
  return 0;
}
// magma_ssyr2k time: 0.22002 sec.
//
// after magma_ssyr2k:
// c:
//  2718.7930,...
//  2054.1855, 2763.3325,...
//  2022.0312, 2043.4248, 2702.5745,...
//  2043.3660, 2075.6743, 2048.9951, 2753.3296,...
//  ..........................................
```

4.2.9 `magma_strmm` - triangular matrix-matrix multiplication

This function performs the left or right triangular matrix-matrix multiplications

$$C = \alpha\, op(A)\, B \qquad \text{in } \texttt{MagmaLeft,'L'} \text{ case,}$$
$$C = \alpha\, B\, op(A) \qquad \text{in } \texttt{MagmaRight,'R'} \text{ case,}$$

where A is a triangular matrix, C, B are $m \times n$ matrices and α is a scalar. The value of $op(A)$ can be equal to A in `MagmaNoTrans,'N'` case, A^T (transposition) in `MagmaTrans,'T'` case or A^H (conjugate transposition) in `MagmaConjTrans,'C'` case. A has dimension $m \times m$ in the first case and $n \times n$ in the second case. A can be stored in lower (`MagmaLower,'L'`) or upper (`MagmaUpper,'U'`) mode. If the diagonal of the matrix A has non-unit elements, then the parameter `MagmaNonUnit,'N'` should be used (in the opposite case - `MagmaUnit,'U'`).

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                              // initialize Magma
  magma_timestr_t  start, end;
  float   gpu_time;
  magma_int_t info;
  magma_int_t m = 8192;                        // a - mxm matrix
  magma_int_t n = 4096;                        // c - mxn matrix
  magma_int_t mm=m*m;                               // size of a
  magma_int_t mn=m*n;                               // size of c
  float *a;                          // a- mxm matrix on the host
  float *c;                          // c- mxn matrix on the host
  float *d_a;                // d_a- mxm matrix a on the device
  float *d_c;                // d_c- mxn matrix c on the device
  float alpha = 1.0;                                 // alpha=1
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = { 0,1,2,3 };                   // seed
  magma_err_t err;
// allocate matrices on the host
  err = magma_smalloc_pinned( &a , mm );  // host memory for a
  err = magma_smalloc_pinned( &c , mn );  // host memory for c
// allocate matrix and vectors on the device
  err = magma_smalloc( &d_a,  mm );     // device memory for a
  err = magma_smalloc( &d_c,  mn );     // device memory for c
// generate random matrices a, c;
  lapackf77_slarnv(&ione,ISEED,&mm,a);              // random a
  lapackf77_slarnv(&ione,ISEED,&mn,c);              // random c
// lower triangular part of a is the lower triangular part
// of some lower triangular matrix, the strictly upper
// triangular part of c is not referenced
```

```
// copy data from host to device
  magma_ssetmatrix( m, m, a,  m, d_a,  m );    // copy a -> d_a
  magma_ssetmatrix( m, n, c,  m, d_c,  m );    // copy c -> d_c
// triangular matrix-matrix multiplication
// d_c=alpha*d_a*d_c
// d_c -mxn matrix,  d_a -mxm triangular matrix;
// alpha - scalar
  start = get_current_time();

  magma_strmm(MagmaLeft,MagmaUpper,MagmaNoTrans,MagmaNonUnit,
                                        m,n,alpha,d_a,m,d_c,m);

  end = get_current_time();
  gpu_time=GetTimerValue(start,end)/1e3;
  printf("magma_strmm time: %7.5f sec.\n",gpu_time);
// copy data from device to host
  magma_sgetmatrix( m, n, d_c, m, c, m );      // copy d_c -> c
  printf("after magma_strmm:\n");
  printf("c:\n");
  for(int i=0;i<4;i++){
  for(int j=0;j<4;j++) if(i>=j) printf("%10.4f,",c[i*m+j]);
  printf("...\n");}
  printf(".....\n");
  magma_free_pinned(a);                       // free host memory
  magma_free_pinned(c);                       // free host memory
  magma_free(d_a);                           // free device memory
  magma_free(d_c);                           // free device memory
  magma_finalize();                              // finalize Magma
  return 0;
}
// magma_strmm time: 1.28922 sec.
//
// after magma_strmm:
// c:
//  2051.0044,...
//  2040.4779, 2027.2761,...
//  2077.4158, 2052.2385, 2050.4998,...
//  2028.7089, 2034.3583, 2003.8667, 2031.4482,...
//  .....
```

4.2.10 `magmablas_sgeadd` - matrix-matrix addition

This function performs the addition of matrices

$$C = \alpha A + C,$$

where A, C are $m \times n$ matrices and α is a scalar.

```
#include <stdio.h>
#include <cuda.h>
```

```
#include "magma.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                              // initialize Magma
  magma_timestr_t  start, end;
  float   gpu_time;
  magma_int_t m = 8192;                        // a - mxn matrix
  magma_int_t n = 4096;                        // c - mxn matrix
  magma_int_t mn=m*n;                               // size of c
  float *a;                          // a- mxn matrix on the host
  float *c;                          // c- mxn matrix on the host
  float *d_a;                // d_a- mxn matrix a on the device
  float *d_c;                // d_c- mxn matrix c on the device
  float alpha = 2.0;                                 // alpha=2
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = { 0,1,2,3 };                    // seed
  magma_err_t err;
// allocate matrices on the host
  err = magma_smalloc_pinned( &a , mn );  // host memory for a
  err = magma_smalloc_pinned( &c , mn );  // host memory for c
// allocate matrix and vectors on the device
  err = magma_smalloc( &d_a,  mn );     // device memory for a
  err = magma_smalloc( &d_c,  mn );     // device memory for c
// generate random matrices a, c;
  lapackf77_slarnv(&ione,ISEED,&mn,a);              // random a
  lapackf77_slarnv(&ione,ISEED,&mn,c);              // random c
  printf("a:\n");
  for(int i=0;i<4;i++){
  for(int j=0;j<4;j++)  printf("%10.4f,",a[i*m+j]);
  printf("...\n");}
  printf(".....\n");
  printf("c:\n");
  for(int i=0;i<4;i++){
  for(int j=0;j<4;j++)  printf("%10.4f,",c[i*m+j]);
  printf("...\n");}
  printf(".....\n");
// copy data from host to device
  magma_ssetmatrix( m, n, a,  m, d_a,  m );   // copy a -> d_a
  magma_ssetmatrix( m, n, c,  m, d_c,  m );   // copy c -> d_c
//
// d_c=alpha*d_a+d_c
// d_a, d_c -mxn matrices;
// alpha - scalar
  start = get_current_time();

  magmablas_sgeadd(m,n,alpha,d_a,m,d_c,m);

  end = get_current_time();
  gpu_time=GetTimerValue(start,end)/1e3;
  printf("magmablas_sgeadd time: %7.5f sec.\n",gpu_time);
// copy data from device to host
  magma_sgetmatrix( m, n, d_c, m, c, m );     // copy d_c -> c
```

```
  printf("after magmablas_sgeadd:\n");
  printf("c:\n");
//  for(int i=0;i<4;i++){
//  for(int j=0;j<4;j++)  printf("%10.4f,",c[i*m+j]);
//  printf("...\n");}
//  printf("..........................................\n");
magma_sprint( 4, 4, c, m );
  magma_free_pinned(a);                    // free host memory
  magma_free_pinned(c);                    // free host memory
  magma_free(d_a);                        // free device memory
  magma_free(d_c);                        // free device memory
  magma_finalize();                          // finalize Magma
  return 0;
}
// a:
//    0.1319,    0.2338,    0.3216,    0.7105,...
//    0.6137,    0.0571,    0.4461,    0.8876,...
//    0.5486,    0.9655,    0.8833,    0.8968,...
//    0.5615,    0.0839,    0.2581,    0.8629,...
//.........................................
// c:
//    0.0443,    0.4490,    0.8054,    0.1554,...
//    0.1356,    0.5692,    0.6642,    0.2544,...
//    0.6798,    0.7744,    0.8358,    0.1854,...
//    0.3021,    0.1897,    0.9450,    0.0734,...
//.........................................
// magmablas_sgeadd time: 0.00348 sec.
//
// after magmablas_sgeadd (c=2a+c):
// c:
//    0.3080,    0.9166,    1.4487,    1.5765,...
//    1.3630,    0.6835,    1.5565,    2.0297,...
//    1.7771,    2.7055,    2.6023,    1.9789,...
//    1.4252,    0.3575,    1.4612,    1.7992,...
//.........................................
```

4.3 LU decomposition and solving general linear systems

4.3.1 magma_sgesv - solve a general linear system in single precision, CPU interface

This function solves in single precision a general real linear system

$$A\,X = B,$$

where A is an $m \times m$ matrix and X, B are $m \times n$ matrices. A, B are defined on the host. In the solution, the LU decomposition of A with partial pivoting and row interchanges is used. See magma-X.Y.Z/src/sgesv.cpp for more details.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                              // initialize Magma
  magma_timestr_t  start, end;
  float   gpu_time, cpu_time;
  magma_int_t *piv, info;  // piv - array of indices of inter-
  magma_int_t m = 2*8192;// changed rows;   a,r - mxm matrices
  magma_int_t n = 100;                 // b,c,c1 - mxn matrices
  magma_int_t mm=m*m;                             // size of a,r
  magma_int_t mn=m*n;                           // size of b,c,c1
  float *a;                         // a- mxm matrix on the host
  float *r;                         // r- mxm matrix on the host
  float *b;                         // b- mxn matrix on the host
  float *c;                         // c- mxn matrix on the host
  float *c1;                       // c1- mxn matrix on the host
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = {0,0,0,1};                    // seed
  magma_err_t err;
  const float alpha = 1.0;                            // alpha=1
  const float beta = 0.0;                              // beta=0
// allocate matrices  on the host
  err = magma_smalloc_pinned( &a , mm );  // host memory for a
  err = magma_smalloc_pinned( &r , mm );  // host memory for r
  err = magma_smalloc_pinned( &b , mn );  // host memory for b
  err = magma_smalloc_pinned( &c , mn );  // host memory for c
  err = magma_smalloc_pinned( &c1 , mn );// host memory for c1
  piv=(magma_int_t*)malloc(m*sizeof(magma_int_t));// host mem.
// generate  matrices                                 // for piv
  lapackf77_slarnv(&ione,ISEED,&mm,a);              // random a
  lapackf77_slaset( MagmaUpperLowerStr,&m,&n,&alpha,&alpha,
                            b,&m);  // b -mxn  matrix of ones
  lapackf77_slacpy(MagmaUpperLowerStr,&m,&m,a,&m,r,&m);// a->r
  printf("upper left corner of the expected solution:\n");
  magma_sprint( 4, 4, b, m ); // part of the expected solution
// right hand side c=a*b
  blasf77_sgemm("N","N",&m,&n,&m,&alpha,a,&m,b,&m,&beta,c,&m);
  lapackf77_slacpy(MagmaUpperLowerStr,&m,&n,c,&m,c1,&m);//c->c1
// MAGMA
// solve the linear system a*x=c, a -mxm matrix, c -mxn matrix,
// c is overwritten by the solution; LU decomposition with par-
// tial pivoting and row interchanges is used, row i of a is
// interchanged with row piv(i)
  start = get_current_time();

  magma_sgesv(m,n,a,m,piv,c,m,&info);

  end = get_current_time();
  gpu_time=GetTimerValue(start,end)/1e3;              // Magma
  printf("magma_sgesv time: %7.5f sec.\n",gpu_time);  // time
```

```
  printf("upper left corner of the magma solution:\n");
  magma_sprint( 4, 4, c, m );   // part of the Magma solution
// LAPACK
  start = get_current_time();
  lapackf77_sgesv(&m,&n,r,&m,piv,c1,&m,&info);
  end = get_current_time();
  cpu_time=GetTimerValue(start,end)/1e3;      // Lapack  time
  printf("lapackf77_sgesv time:  %7.5f sec.\n",cpu_time);
  printf("upper left corner of the lapack solution:\n");
  magma_sprint( 4, 4, c1, m );  // part of the Lapack solution
  magma_free_pinned(a);                     // free host memory
  magma_free_pinned(r);                     // free host memory
  magma_free_pinned(b);                     // free host memory
  magma_free_pinned(c);                     // free host memory
  magma_free_pinned(c1);                    // free host memory
  free(piv);                                // free host memory
  magma_finalize();                           // finalize Magma
  return 0;
}
// upper left corner of the expected solution:
//[
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//];
// magma_sgesv time: 2.48341  sec.
//
// upper left corner of the magma solution:
//[
//   0.9999   0.9999   0.9999   0.9999
//   1.0223   1.0223   1.0223   1.0223
//   1.0001   1.0001   1.0001   1.0001
//   0.9871   0.9871   0.9871   0.9871
//];
// lapackf77_sgesv time: 6.82807 sec.
//
// upper left corner of the lapack solution:
//[
//   0.9868   0.9868   0.9868   0.9868
//   1.0137   1.0137   1.0137   1.0137
//   1.0071   1.0071   1.0071   1.0071
//   0.9986   0.9986   0.9986   0.9986
//];
```

4.3.2 magma_dgesv - solve a general linear system in double precision, CPU interface

This function solves in double precision a general real linear system

$$A\,X = B,$$

where A is an $m \times m$ matrix and X, B are $m \times n$ matrices. A, B are defined on the host. In the solution, the LU decomposition of A with partial pivoting and row interchanges is used. See `magma-X.Y.Z/src/dgesv.cpp` for more details.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                              // initialize Magma
  magma_timestr_t  start, end;
  double   gpu_time, cpu_time;
  magma_int_t *piv, info;  // piv - array of indices of inter-
  magma_int_t m = 2*8192;    // changed rows; a,r - mxm matrices
  magma_int_t n = 100;                 // b,c,c1 - mxn matrices
  magma_int_t mm=m*m;                             // size of a,r
  magma_int_t mn=m*n;                          // size of b,c,c1
  double *a;                        // a- mxm matrix on the host
  double *r;                        // r- mxm matrix on the host
  double *b;                        // b- mxn matrix on the host
  double *c;                        // c- mxn matrix on the host
  double *c1;                      // c1- mxn matrix on the host
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = { 0,0,0,1 };                  // seed
  magma_err_t err;
  const double alpha = 1.0;                           // alpha=1
  const double beta = 0.0;                             // beta=0
// allocate matrices on the host
  err = magma_dmalloc_pinned( &a , mm );  // host memory for a
  err = magma_dmalloc_pinned( &r , mm );  // host memory for a
  err = magma_dmalloc_pinned( &b , mn );  // host memory for b
  err = magma_dmalloc_pinned( &c , mn );  // host memory for c
  err = magma_dmalloc_pinned( &c1 , mn );// host memory for c1
  piv=(magma_int_t*)malloc(m*sizeof(magma_int_t));// host mem.
// generate  matrices a, b;                            // for piv
  lapackf77_dlarnv(&ione,ISEED,&mm,a);              // random a
// b - mxn matrix of ones
  lapackf77_dlaset(MagmaUpperLowerStr,&m,&n,&alpha,&alpha,b,&m);
  lapackf77_dlacpy(MagmaUpperLowerStr,&m,&m,a,&m,r,&m); //a->r
  printf("upper left corner of the expected solution:\n");
  magma_dprint( 4, 4, b, m ); // part of the expected solution
// right hand side c=a*b
  blasf77_dgemm("N","N",&m,&n,&m,&alpha,a,&m,b,&m,&beta,c,&m);
  lapackf77_dlacpy(MagmaUpperLowerStr,&m,&n,c,&m,c1,&m);//c->c1
// MAGMA
// solve the linear system a*x=c, a -mxm matrix, c -mxn matrix,
// c is overwritten by the solution; LU decomposition with par-
// tial pivoting and row interchanges is used, row i of a is
// interchanged with row piv(i)
  start = get_current_time();
```

```
  magma_dgesv(m,n,a,m,piv,c,m,&info);

  end = get_current_time();
  gpu_time=GetTimerValue(start,end)/1e3;              // Magma
  printf("magma_dgesv time: %7.5f sec.\n",gpu_time);  // time
  printf("upper left corner of the magma solution:\n");
  magma_dprint( 4, 4, c, m );   // part of the Magma solution
// LAPACK
  start = get_current_time();
  lapackf77_dgesv(&m,&n,r,&m,piv,c1,&m,&info);
  end = get_current_time();
  cpu_time=GetTimerValue(start,end)/1e3;         // Lapack time
  printf("lapackf77_dgesv time:  %7.5f sec.\n",cpu_time);
  printf("upper left corner of the lapack solution:\n");
  magma_dprint( 4, 4, c1, m );  // part of the Lapack solution
  magma_free_pinned(a);                        // free host memory
  magma_free_pinned(r);                        // free host memory
  magma_free_pinned(b);                        // free host memory
  magma_free_pinned(c);                        // free host memory
  magma_free_pinned(c1);                       // free host memory
  free(piv);                                   // free host memory
  magma_finalize();                             // finalize Magma
  return 0;
}
// upper left corner of the expected solution:
//[
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//];
// magma_dgesv time: 4.73224 sec.
//
// upper left corner of the magma solution:
//[
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//];
// lapackf77_dgesv time: 13.53960 sec.
//
// upper left corner of the lapack solution:
//[
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//];
```

4.3.3 magma_sgesv_gpu - solve a general linear system in single precision, GPU interface

This function solves in single precision a general real linear system

$$A\,X = B,$$

where A is an $m \times m$ matrix and X, B are $m \times n$ matrices. A, B are defined on the device. In the solution, the LU decomposition of A with partial pivoting and row interchanges is used. See magma-X.Y.Z/src/sgesv_gpu.cpp for more details.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                              // initialize Magma
  magma_timestr_t  start, end;
  float   gpu_time;
  magma_int_t *piv, info;  // piv - array of indices of inter-
  magma_int_t m = 2*8192;  // changed rows; a,d_a - mxm matrices
  magma_int_t n = 100;               // b,c,d_c - mxn matrices
  magma_int_t mm=m*m;                          // size of a,a_d
  magma_int_t mn=m*n;                         // size of b,c,d_c
  float *a;                       // a- mxm matrix on the host
  float *b;                       // b- mxn matrix on the host
  float *c;                       // c- mxn matrix on the host
  float *d_a;               // d_a- mxm matrix a on the device
  float *d_c;               // d_c- mxn matrix c on the device
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = {0,0,0,1};                    // seed
  magma_err_t err;
  const float alpha = 1.0;                           // alpha=1
  const float beta = 0.0;                             // beta=0
// allocate matrices
  err = magma_smalloc_cpu( &a , mm );     // host memory for a
  err = magma_smalloc_cpu( &b , mn );     // host memory for b
  err = magma_smalloc_cpu( &c , mn );     // host memory for c
  err = magma_smalloc( &d_a,  mm );     // device memory for a
  err = magma_smalloc( &d_c,  mn );     // device memory for c
  piv=(magma_int_t*)malloc(m*sizeof(magma_int_t));// host mem.
// generate  matrices a, b;                          // for piv
  lapackf77_slarnv(&ione,ISEED,&mm,a);             // random a
// b - mxn matrix of ones
  lapackf77_slaset(MagmaUpperLowerStr,&m,&n,&alpha,&alpha,b,&m);
  printf("upper left corner of the expected solution:\n");
  magma_sprint( 4, 4, b, m ); // part of the expected solution
// right hand side c=a*b
  blasf77_sgemm("N","N",&m,&n,&m,&alpha,a,&m,b,&m,&beta,c,&m);
  magma_ssetmatrix( m, m, a,  m, d_a,  m );   // copy a -> d_a
```

```
  magma_ssetmatrix( m, n, c,  m, d_c,  m );   // copy c -> d_c
// MAGMA
// solve the linear system d_a*x=d_c, d_a -mxm matrix,
// d_c -mxn matrix, d_c is overwritten by the solution;
// LU decomposition with partial pivoting and row
// interchanges is used, row i is interchanged with row piv(i)
  start = get_current_time();

  magma_sgesv_gpu(m,n,d_a,m,piv,d_c,m,&info);

  end = get_current_time();
  gpu_time=GetTimerValue(start,end)/1e3;         // Magma time
  printf("magma_sgesv_gpu time: %7.5f sec.\n",gpu_time);
  magma_sgetmatrix( m, n, d_c, m, c, m );
  printf("upper left corner of the magma solution:\n");
  magma_sprint( 4, 4, c, m );    // part of the Magma solution
  free(a);                                  // free host memory
  free(b);                                  // free host memory
  free(c);                                  // free host memory
  free(piv);                                // free host memory
  magma_free(d_a);                         // free device memory
  magma_free(d_c);                         // free device memory
  magma_finalize();                           // finalize Magma
  return 0;
}
// upper left corner of the expected solution:
//[
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//];
// magma_sgesv_gpu time: 1.99060 sec.
//
// upper left corner of the solution:
//[
//   0.9999   0.9999   0.9999   0.9999
//   1.0223   1.0223   1.0223   1.0223
//   1.0001   1.0001   1.0001   1.0001
//   0.9871   0.9871   0.9871   0.9871
//];
```

4.3.4 magma_dgesv_gpu - solve a general linear system in double precision, GPU interface

This function solves in double precision a general real linear system

$$A\,X = B,$$

where A is an $m \times m$ matrix and X, B are $m \times n$ matrices. A, B are defined on the device. In the solution, the LU decomposition of A with partial pivoting

and row interchanges is used. See magma-X.Y.Z/src/dgesv_gpu.cpp for more details.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                              // initialize Magma
  magma_timestr_t  start, end;
  float   gpu_time;
  magma_int_t *piv, info;  // piv - array of indices of inter-
  magma_int_t m = 8192;  // changed rows; a,d_a - mxm matrices
  magma_int_t n = 100;                // b,c,d_c - mxn matrices
  magma_int_t mm=m*m;                          // size of a,a_d
  magma_int_t mn=m*n;                        // size of b,c,d_c
  double *a;                       // a- mxm matrix on the host
  double *b;                       // b- mxn matrix on the host
  double *c;                       // c- mxn matrix on the host
  double *d_a;               // d_a- mxm matrix a on the device
  double *d_c;               // d_c- mxn matrix c on the device
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = { 0,0,0,1 };                  // seed
  magma_err_t err;
  const double alpha = 1.0;                           // alpha=1
  const double beta = 0.0;                             // beta=0
// allocate matrices
  err = magma_dmalloc_cpu( &a , mm );     // host memory for a
  err = magma_dmalloc_cpu( &b , mn );     // host memory for b
  err = magma_dmalloc_cpu( &c , mn );     // host memory for c
  err = magma_dmalloc( &d_a,  mm );     // device memory for a
  err = magma_dmalloc( &d_c,  mn );     // device memory for c
  piv=(magma_int_t*)malloc(m*sizeof(magma_int_t));// host mem.
// generate matrices                                  // for piv
  lapackf77_dlarnv(&ione,ISEED,&mm,a);              // random a
// b - mxn matrix of ones
  lapackf77_dlaset(MagmaUpperLowerStr,&m,&n,&alpha,&alpha,b,&m);
  printf("upper left corner of the expected solution:\n");
  magma_dprint( 4, 4, b, m ); // part of the expected solution
// right hand side c=a*b
  blasf77_dgemm("N","N",&m,&n,&m,&alpha,a,&m,b,&m,&beta,c,&m);
  magma_dsetmatrix( m, m, a,  m, d_a,  m );   // copy a -> d_a
  magma_dsetmatrix( m, n, c,  m, d_c,  m );   // copy c -> d_c
// MAGMA
// solve the linear system d_a*x=d_c, d_a -mxm matrix,
// d_c -mxn matrix, d_c is overwritten by the solution;
// LU decomposition with partial pivoting and row
// interchanges is used, row i is interchanged with row piv(i)
  start = get_current_time();

  magma_dgesv_gpu(m,n,d_a,m,piv,d_c,m,&info);
```

```
  end = get_current_time();
  gpu_time=GetTimerValue(start,end)/1e3;          // Magma time
  printf("magma_dgesv_gpu time: %7.5f sec.\n",gpu_time);
  magma_dgetmatrix( m, n, d_c, m, c, m );
  printf("upper left corner of the solution:\n");
  magma_dprint( 4, 4, c, m );     // part of the Magma solution
  free(a);                                    // free host memory
  free(b);                                    // free host memory
  free(c);                                    // free host memory
  free(piv);                                  // free host memory
  magma_free(d_a);                           // free device memory
  magma_free(d_c);                           // free device memory
  magma_finalize();                             // finalize Magma
  return 0;
}
// upper left corner of the expected solution:
//[
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//];
// magma_dgesv_gpu time:  3.90760 sec.
//
// upper left corner of the solution:
//[
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//];
```

4.3.5 `magma_sgetrf`, `lapackf77_sgetrs` - LU factorization and solving factorized systems in single precision, CPU interface

The first function using single precision computes an LU factorization of a general $m \times n$ matrix A using partial pivoting with row interchanges:

$$A = P\ L\ U,$$

where P is a permutation matrix, L is lower triangular with unit diagonal, and U is upper diagonal. The matrix A to be factored is defined on the host. On exit A contains the factors L, U. The information on the interchanged rows is contained in *piv*. See `magma-X.Y.Z/src/sgetrf.cpp` for more details.
Using the obtained factorization one can replace the problem of solving a general linear system by solving two triangular systems with matrices L and U respectively. The Lapack function `sgetrs` uses the LU factorization to solve a general linear system (it is faster to use Lapack `sgetrs` than to copy A to the device).

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                              // initialize Magma
  magma_timestr_t  start, end;
  float   gpu_time;
  magma_int_t *piv, info;  // piv - array of indices of inter-
  magma_int_t m = 2*8192,n=m;  // changed rows; a - m*n matrix
  magma_int_t nrhs = 100;   // b - n*nrhs, c - m*nrhs matrices
  magma_int_t mn=m*n;                              // size of a
  magma_int_t nnrhs=n*nrhs;                        // size of b
  magma_int_t mnrhs=m*nrhs;                        // size of c
  float *a;                       // a- m*n matrix on the host
  float *b;                     // b- n*nrhs matrix on the host
  float *c;                     // c- m*nrhs matrix on the host
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = {0,0,0,1};                   // seed
  magma_err_t err;
  const float alpha = 1.0;                          // alpha=1
  const float beta = 0.0;                           // beta=0
// allocate matrices on the host
  err = magma_smalloc_pinned(&a , mn );   // host memory for a
  err = magma_smalloc_pinned(&b, nnrhs ); // host memory for b
  err = magma_smalloc_pinned(&c, mnrhs ); // host memory for c
  piv=(magma_int_t*)malloc(m*sizeof(magma_int_t));// host mem.
// generate  matrices                               // for piv
  lapackf77_slarnv(&ione,ISEED,&mn,a);             // random a
  lapackf77_slaset(MagmaUpperLowerStr,&n,&nrhs,&alpha,&alpha,
                          b,&n);// b - n*nrhs matrix of ones
  printf("upper left corner of the expected solution:\n");
  magma_sprint( 4, 4, b, m );// part of the  expected solution
  blasf77_sgemm("N","N",&m,&nrhs,&n,&alpha,a,&m,b,&m,&beta,c,
                           &m);     // right hand side c=a*b
// MAGMA
// solve the linear system a*x=c, a -m*n matrix, c -m*nrhs ma-
// trix, c is overwritten by the solution; LU decomposition
// with partial pivoting and row interchanges is used,
// row i is interchanged with row piv(i)
  start = get_current_time();

  magma_sgetrf( m, n, a, m, piv, &info);
  lapackf77_sgetrs("N",&m,&nrhs,a,&m,piv,c,&m,&info);

  end = get_current_time();
  gpu_time=GetTimerValue(start,end)/1e3;
  printf("magma_sgetrf + lapackf77_sgetrs time: %7.5f sec.\n",
                           gpu_time); // Magma + Lapack time
  printf("upper left corner of the Magma/Lapack solution:\n");
  magma_sprint( 4, 4, c, m ); // part of the Magma/Lapack sol.
  magma_free_pinned(a);                    // free host memory
```

```
  magma_free_pinned(b);                    // free host memory
  magma_free_pinned(c);                    // free host memory
  free(piv);                               // free host memory
  magma_finalize();                          // finalize Magma
  return 0;
}
// upper left corner of the expected solution:
//[
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//];
// magma_sgetrf + lapackf77_sgetrs time: 3.13849 sec.
//
// upper left corner of the Magma/Lapack solution:
//[
//   0.9965   0.9965   0.9965   0.9965
//   1.0065   1.0065   1.0065   1.0065
//   0.9968   0.9968   0.9968   0.9968
//   0.9839   0.9839   0.9839   0.9839
//];
```

4.3.6 magma_dgetrf, lapackf77_dgetrs - LU factorization and solving factorized systems in double precision, CPU interface

The first function using double precision computes an LU factorization of a general $m \times n$ matrix A using partial pivoting with row interchanges:

$$A = P\,L\,U,$$

where P is a permutation matrix, L is lower triangular with unit diagonal, and U is upper diagonal. The matrix A to be factored is defined on the host. On exit A contains the factors L, U. The information on the interchanged rows is contained in *piv*. See magma-X.Y.Z/src/sgetrf.cpp for more details.
Using the obtained factorization one can replace the problem of solving a general linear system by solving two triangular systems with matrices L and U respectively. The Lapack function `dgetrs` uses the LU factorization to solve a general linear system (it is faster to use Lapack `dgetrs` than to copy A to the device).

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                            // initialize Magma
  magma_timestr_t  start, end;
```

```
  float   gpu_time;
  magma_int_t *piv, info;  // piv - array of indices of inter-
  magma_int_t m = 8192,n=8192; // changed rows; a - m*n matrix
  magma_int_t nrhs = 100;   // b - n*nrhs, c - m*nrhs matrices
  magma_int_t mn=m*n;                              // size of a
  magma_int_t nnrhs=n*nrhs;                        // size of b
  magma_int_t mnrhs=m*nrhs;                        // size of c
  double *a;                      // a- m*n matrix on the host
  double *b;                    // b- n*nrhs matrix on the host
  double *c;                    // c- m*nrhs matrix on the host
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = {0,0,0,1};                   // seed
  magma_err_t err;
  const double alpha = 1.0;                          // alpha=1
  const double beta = 0.0;                           // beta=0
// allocate matrices on the host
  err = magma_dmalloc_pinned(&a,mn);      // host memory for a
  err = magma_dmalloc_pinned(&b,nnrhs);   // host memory for b
  err = magma_dmalloc_pinned(&c,mnrhs);   // host memory for c
  piv=(magma_int_t*)malloc(m*sizeof(magma_int_t));// host mem.
// generate  matrices                                 // for piv
  lapackf77_dlarnv(&ione,ISEED,&mn,a);              // random a
  lapackf77_dlaset(MagmaUpperLowerStr,&n,&nrhs,&alpha,&alpha,
                          b,&n);// b - n*nrhs matrix of ones
  printf("upper left corner of the expected solution:\n");
  magma_dprint( 4, 4, b, m ); // part of the expected solution
// right hand side c=a*b
  blasf77_dgemm("N","N",&m,&nrhs,&n,&alpha,a,&m,b,&m,&beta,
                          c,&m);    // right hand side c=a*b
// MAGMA
// solve the linear system a*x=c, a -m*n matrix, c -m*nrhs ma-
// trix, c is overwritten by the solution; LU decomposition
// with partial pivoting and row interchanges is used,
// row i is interchanged with row piv(i)
  start = get_current_time();

  magma_dgetrf(m,n,a,m,piv,&info);
  lapackf77_dgetrs("N",&m,&nrhs,a,&m,piv,c,&m,&info);

  lapackf77_dgetrs("N",&m,&nrhs,a,&m,piv,c,&m,&info);
  end = get_current_time();
  gpu_time=GetTimerValue(start,end)/1e3;
  printf("magma_dgetrf + lapackf77_dgetrs time: %7.5f sec.\n",
                           gpu_time); // Magma + Lapack time
  printf("upper left corner of the Magma/Lapack solution:\n");
  magma_dprint( 4, 4, c, m ); // part of the Magma/Lapack sol.
  magma_free_pinned(a);                       // free host memory
  magma_free_pinned(b);                       // free host memory
  magma_free_pinned(c);                       // free host memory
  free(piv);                                  // free host memory
  magma_finalize();                             // finalize Magma
  return 0;
```

```
}
// upper left corner of the expected solution:
//[
//    1.0000    1.0000    1.0000    1.0000
//    1.0000    1.0000    1.0000    1.0000
//    1.0000    1.0000    1.0000    1.0000
//    1.0000    1.0000    1.0000    1.0000
//];
// magma_dgetrf + lapackf77_dgetrs time: 6.04550 sec.
//
// upper left corner of the Magma/Lapack solution:
//[
//    1.0000    1.0000    1.0000    1.0000
//    1.0000    1.0000    1.0000    1.0000
//    1.0000    1.0000    1.0000    1.0000
//    1.0000    1.0000    1.0000    1.0000
//];
```

4.3.7 magma_sgetrf_gpu, magma_sgetrs_gpu - LU factorization and solving factorized systems in single precision, GPU interface

The function magma_sgetrf_gpu computes in single precision an LU factorization of a general $m \times n$ matrix A using partial pivoting with row interchanges:

$$A = P\, L\, U,$$

where P is a permutation matrix, L is lower triangular with unit diagonal, and U is upper diagonal. The matrix A to be factored and the factors L, U are defined on the device. The information on the interchanged rows is contained in *piv*. See magma-X.Y.Z/src/sgetrf_gpu.cpp for more details. Using the obtained factorization one can replace the problem of solving a general linear system by solving two triangular systems with matrices L and U respectively. The function magma_sgetrs_gpu uses the L, U factors defined on the device by magma_sgetrf_gpu to solve in single precision a general linear system

$$A\, X = B.$$

The right hand side B is a matrix defined on the device. On exit it is replaced by the solution. See magma-X.Y.Z/src/sgetrs_gpu.cpp for more details.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                                  // initialize Magma
```

```
  magma_timestr_t  start, end;
  float   gpu_time;
  magma_int_t *piv, info;  // piv - array of indices of inter-
  magma_int_t m = 8192,n=8192; // changed rows; a - m*n matrix
  magma_int_t nrhs = 100;   // b - n*nrhs, c - m*nrhs matrices
  magma_int_t mn=m*n;                              // size of a
  magma_int_t nnrhs=n*nrhs;                        // size of b
  magma_int_t mnrhs=m*nrhs;                        // size of c
  float *a;                       // a- m*n matrix on the host
  float *b;                    // b- n*nrhs matrix on the host
  float *c;                    // c- m*nrhs matrix on the host
  float *d_a;                // d_a- m*n matrix a on the device
  float *d_c;             // d_c- m*nrhs matrix c on the device
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = {0,0,0,1};                   // seed
  magma_err_t err;
  const float alpha = 1.0;                           // alpha=1
  const float beta = 0.0;                             // beta=0
// allocate matrices
  err = magma_smalloc_cpu( &a , mn );     // host memory for a
  err = magma_smalloc_cpu( &b , nnrhs );  // host memory for b
  err = magma_smalloc_cpu( &c , mnrhs );  // host memory for c
  err = magma_smalloc( &d_a,  mn );     // device memory for a
  err = magma_smalloc( &d_c,  mnrhs );  // device memory for c
  piv=(magma_int_t*)malloc(m*sizeof(magma_int_t));// host mem.
// generate matrices                                 // for piv
  lapackf77_slarnv(&ione,ISEED,&mn,a);             // random a
  lapackf77_slaset(MagmaUpperLowerStr,&n,&nrhs,&alpha,&alpha,
                         b,&n);// b - n*nrhs matrix of ones
  printf("upper left corner of the expected solution:\n");
  magma_sprint( 4, 4, b, m ); // part of the expected solution
// right hand side c=a*b
  blasf77_sgemm("N","N",&m,&nrhs,&n,&alpha,a,&m,b,&m,&beta,c,&m);
  magma_ssetmatrix( m, n, a,  m, d_a,  m );   // copy a -> d_a
  magma_ssetmatrix( m, nrhs, c,  m, d_c,  m );// copy c -> d_c
// MAGMA
// solve the linear system d_a*x=d_c, d_a -m*n matrix,
// d_c -m*nrhs matrix, d_c is overwritten by the solution;
// LU decomposition with partial pivoting and row interchanges
// is used, row i is interchanged with row piv(i)
  start = get_current_time();

  magma_sgetrf_gpu( m, n, d_a, m, piv, &info);
  magma_sgetrs_gpu(MagmaNoTrans,m,nrhs,d_a,m,piv,d_c,m,&info);

  end = get_current_time();
  gpu_time=GetTimerValue(start,end)/1e3;
  printf("magma_sgetrf_gpu+magma_sgetrs_gpu time: %7.5f sec.\n",
                                    gpu_time);  // Magma time
  magma_sgetmatrix( m, nrhs, d_c, m, c, m );
  printf("upper left corner of the Magma solution:\n");
  magma_sprint( 4, 4, c, m );    // part of the Magma solution
```

```
    free(a);                                    // free host memory
    free(b);                                    // free host memory
    free(c);                                    // free host memory
    free(piv);                                  // free host memory
    magma_free(d_a);                           // free device memory
    magma_free(d_c);                           // free device memory
    magma_finalize();                              // finalize Magma
    return 0;
}
// upper left corner of the expected solution:
//[
//    1.0000    1.0000    1.0000    1.0000
//    1.0000    1.0000    1.0000    1.0000
//    1.0000    1.0000    1.0000    1.0000
//    1.0000    1.0000    1.0000    1.0000
//];
// magma_sgetrf_gpu + magma_sgetrs_gpu time: 1.98868 sec.
//
// upper left corner of the Magma solution:
//[
//    0.9999    0.9999    0.9999    0.9999
//    1.0223    1.0223    1.0223    1.0223
//    1.0001    1.0001    1.0001    1.0001
//    0.9871    0.9871    0.9871    0.9871
//];
```

4.3.8 `magma_dgetrf_gpu`, `magma_dgetrs_gpu` - LU factorization and solving factorized systems in double precision , GPU interface

The function `magma_dgetrf_gpu` computes in double precision an LU factorization of a general $m \times n$ matrix A using partial pivoting with row interchanges:

$$A = P\, L\, U,$$

where P is a permutation matrix, L is lower triangular with unit diagonal, and U is upper diagonal. The matrix A to be factored and the factors L, U are defined on the device. The information on the interchanged rows is contained in *piv*. See `magma-X.Y.Z/src/dgetrf_gpu.cpp` for more details. Using the obtained factorization one can replace the problem of solving a general linear system by solving two triangular systems with matrices L and U respectively. The function `magma_dgetrs_gpu` uses the L, U factors defined on the device by `magma_dgetrf_gpu` to solve in double precision a general linear system

$$A\, X = B.$$

The right hand side B is a matrix defined on the device. On exit it is replaced by the solution. See `magma-X.Y.Z/src/dgetrs_gpu.cpp` for more details.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                              // initialize Magma
  magma_timestr_t  start, end;
  float   gpu_time;
  magma_int_t *piv, info;  // piv - array of indices of inter-
  magma_int_t m = 8192,n=8192; // changed rows; a - m*n matrix
  magma_int_t nrhs = 100;   // b - n*nrhs, c - m*nrhs matrices
  magma_int_t mn=m*n;                              // size of a
  magma_int_t nnrhs=n*nrhs;                        // size of b
  magma_int_t mnrhs=m*nrhs;                       // size of b,c
  double *a;                       // a- m*n matrix on the host
  double *b;                    // b- n*nrhs matrix on the host
  double *c;                    // c- m*nrhs matrix on the host
  double *d_a;               // d_a- m*n matrix a on the device
  double *d_c;            // d_c- m*nrhs matrix c on the device
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = {0,0,0,1};                   // seed
  magma_err_t err;
  const double alpha = 1.0;                          // alpha=1
  const double beta = 0.0;                            // beta=0
// allocate matrices
  err = magma_dmalloc_cpu( &a , mn );     // host memory for a
  err = magma_dmalloc_cpu( &b , nnrhs );  // host memory for b
  err = magma_dmalloc_cpu( &c , mnrhs );  // host memory for c
  err = magma_dmalloc( &d_a,  mn );     // device memory for a
  err = magma_dmalloc( &d_c,  mnrhs );  // device memory for c
  piv=(magma_int_t*)malloc(m*sizeof(magma_int_t));
// generate  matrices a, b;
  lapackf77_dlarnv(&ione,ISEED,&mn,a);             // random a
  lapackf77_dlaset(MagmaUpperLowerStr,&n,&nrhs,&alpha,&alpha,
                           b,&n);// b - n*nrhs matrix of ones
  printf("upper left corner of the expected solution:\n");
  magma_dprint( 4, 4, b, m ); // part of the expected solution
// right hand side c=a*b
  blasf77_dgemm("N","N",&m,&nrhs,&n,&alpha,a,&m,b,&m,&beta,c,&m);
  magma_dsetmatrix( m, n, a,  m, d_a,  m );   // copy a -> d_a
  magma_dsetmatrix( m, nrhs, c,  m, d_c,  m );// copy c -> d_c
// MAGMA
// solve the linear system d_a*x=d_c, d_a -m*n matrix,
// d_c -m*nrhs matrix, d_c is overwritten by the solution;
// LU decomposition with partial pivoting and row interchanges
// is used, row i is interchanged with row piv(i)
  start = get_current_time();

  magma_dgetrf_gpu( m, n, d_a, m, piv, &info);
  magma_dgetrs_gpu(MagmaNoTrans,m,nrhs,d_a,m,piv,d_c,m,&info);

  end = get_current_time();
```

```
  gpu_time=GetTimerValue(start,end)/1e3;
  printf("magma_dgetrf_gpu+magma_dgetrs_gpu time: %7.5f sec.\n",
                                    gpu_time);  // Magma time
  magma_dgetmatrix( m, nrhs, d_c, m, c, m );
  printf("upper left corner of the Magma  solution:\n");
  magma_dprint( 4, 4, c, m );    // part of the Magma solution
  free(a);                                    // free host memory
  free(b);                                    // free host memory
  free(c);                                    // free host memory
  free(piv);                                  // free host memory
  magma_free(d_a);                           // free device memory
  magma_free(d_c);                           // free device memory
  magma_finalize();                              // finalize Magma
  return 0;
}
// upper left corner of the expected solution:
//[
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//];
// magma_dgetrf_gpu+magma_dgetrs_gpu time: 3.91408  sec.
//
// upper left corner of the Magma solution:
//[
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//];
```

4.3.9 `magma_sgetrf_mgpu` - LU factorization in single precision on multiple GPU-s

The function `magma_sgetrf_mgpu` computes in single precision an LU factorization of a general $m \times n$ matrix A using partial pivoting with row interchanges:

$$A = P\, L\, U,$$

where P is a permutation matrix, L is lower triangular with unit diagonal, and U is upper diagonal. The blocks of matrix A to be factored and the blocks of factors L, U are distributed on `num_gpus` devices. The information on the interchanged rows is contained in *ipiv*. See `magma-X.Y.Z/src/sgetrf_mgpu.cpp` for more details.
Using the obtained factorization one can replace the problem of solving a general linear system by solving two triangular systems with matrices L and U respectively. The Lapack function `lapackf77_sgetrs` uses the L, U factors copied from `num_gpus` devices to solve in single precision a general

linear system

$$A\,X = B.$$

The right hand side B is a matrix defined on the host. On exit it is replaced by the solution.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
#define min(a,b)  (((a)<(b))?(a):(b))
extern "C" magma_int_t
magma_sgetrf_mgpu(magma_int_t num_gpus, magma_int_t m1,
                  magma_int_t n,float **d_la, magma_int_t m,
                  magma_int_t *ipiv, magma_int_t *info);
int main( int argc, char** argv)
{
  magma_init();                              // initialize Magma
  magma_err_t err;
  float   cpu_time, gpu_time;
  magma_int_t  m = 2*8192, n = m;         // a,r - m*n matrices
  magma_int_t nrhs=100;      // b -n*nrhs,  c - m*nrhs matrices
  magma_int_t *ipiv;  // array of indices of interchanged rows
  magma_int_t n2=m*n;                             // size of a,r
  magma_int_t nnrhs=n*nrhs;                         // size of b
  magma_int_t mnrhs=m*nrhs;                         // size of c
  magma_int_t ione = 1, info;
  magma_timestr_t start, end;
  float *a, *r;               // a,r - m*n matrices on the host
  float *b, *c; // b - n*nrhs, c - m*nrhs matrices on the host
  float *d_la[4];// d_la[i] - block of matrix a on i-th device
  float alpha=1.0, beta=0.0;                   // alpha=1,beta=0
  magma_int_t num_gpus= 2, n_local;       // number of gpus = 2
  magma_int_t i, k, min_mn=min(m,n), nb, nk;
  magma_int_t ldn_local;// m*ldn_local - size of the part of a
  magma_int_t ISEED[4] = {0,0,0,1};           // on i-th device
  nb =magma_get_sgetrf_nb(m);  // optimal blocksize for sgetrf
// allocate memory on cpu
  ipiv=(magma_int_t*)malloc(min_mn*sizeof(magma_int_t));
                                        // host memory for ipiv
  err = magma_smalloc_pinned(&a,n2);       // host memory for a
  err = magma_smalloc_pinned(&r,n2);       // host memory for r
  err = magma_smalloc_pinned(&b,nnrhs);    // host memory for b
  err = magma_smalloc_pinned(&c,mnrhs);    // host memory for c
// allocate device memory on num_gpus devices
  for(i=0; i<num_gpus; i++){
    n_local = ((n/nb)/num_gpus)*nb;
    if (i < (n/nb)%num_gpus)
       n_local += nb;
    else if (i == (n/nb)%num_gpus)
       n_local += n%nb;
    ldn_local = ((n_local+31)/32)*32;
```

```
    cudaSetDevice(i);
    err = magma_smalloc(&d_la[i],m*ldn_local); //device memory
  }                                            // on i-th device
  cudaSetDevice(0);
// generate matrices
  lapackf77_slarnv( &ione, ISEED, &n2, a );        // random a
  lapackf77_slaset(MagmaUpperLowerStr,&n,&nrhs,&alpha,&alpha,
                           b,&n);// b - n*nrhs matrix of ones
  lapackf77_slacpy( MagmaUpperLowerStr,&m,&n,a,&m,r,&m);//a->r
  printf("upper left corner of the expected solution:\n");
  magma_sprint(4,4,b,m);      // part of the expected solution
  blasf77_sgemm("N","N",&m,&nrhs,&n,&alpha,a,&m,b,&m,
                &beta,c,&m);         // right hand side c=a*b
// LAPACK version of LU decomposition
  start = get_current_time();
  lapackf77_sgetrf(&m, &n, a, &m, ipiv, &info);
  end = get_current_time();
  cpu_time=GetTimerValue(start,end)/1e3;        // Lapack time
  printf("lapackf77_sgetrf time: %7.5f sec.\n",cpu_time);
// copy the corresponding parts of the matrix r to num_gpus
  for(int j=0; j<n; j+=nb){
    k = (j/nb)%num_gpus;
    cudaSetDevice(k);
    nk = min(nb, n-j);
    magma_ssetmatrix( m, nk,r+j*m,m,
                              d_la[k]+j/(nb*num_gpus)*nb*m,m);
    }
  cudaSetDevice(0);
// MAGMA
// LU decomposition on num_gpus devices with partial  pivoting
// and row interchanges, row i is interchanged with row ipiv(i)
  start = get_current_time();

  magma_sgetrf_mgpu( num_gpus, m, n, d_la, m, ipiv, &info);


  end = get_current_time();
  gpu_time=GetTimerValue(start,end)/1e3;         // Magma time
  printf("magma_sgetrf_mgpu time: %7.5f sec.\n",gpu_time);
// copy the decomposition from num_gpus devices to r on the
// host
  for(int j=0; j<n; j+=nb){
    k = (j/nb)%num_gpus;
    cudaSetDevice(k);
    nk = min(nb, n-j);
    magma_sgetmatrix( m, nk,d_la[k]+j/(nb*num_gpus)*nb*m, m,
                                                  r+j*m,m);
  }
  cudaSetDevice(0);
// solve on the host the system r*x=c;  x overwrites c,
// using the LU decomposition obtained on num_gpus devices
  lapackf77_sgetrs("N",&m,&nrhs,r,&m,ipiv,c,&m,&info);
```

```
// print part of the solution from sgetrf_mgpu and sgetrs
  printf("upper left corner of the solution \n\
  from sgetrf_mgpu+sgetrs:\n");   // part of the solution from
  magma_sprint( 4, 4, c, m);     // magma_sgetrf_mgpu + sgetrs
  free(ipiv);                                  // free host memory
  magma_free_pinned(a);                        // free host memory
  magma_free_pinned(r);                        // free host memory
  magma_free_pinned(b);                        // free host memory
  magma_free_pinned(c);                        // free host memory
  for(i=0; i<num_gpus; i++){
    magma_free(d_la[i] );                 // free device memory
  }
  magma_finalize();                              // finalize Magma
}
// upper left corner of the expected solution:
//[
//    1.0000    1.0000    1.0000    1.0000
//    1.0000    1.0000    1.0000    1.0000
//    1.0000    1.0000    1.0000    1.0000
//    1.0000    1.0000    1.0000    1.0000
//];
// lapackf77_sgetrf time: 6.01532  sec.
//
// magma_sgetrf_mgpu time: 1.11910 sec.
//
// upper left corner of the solution
// from sgetrf_mgpu+sgetrs:
//[
//    0.9965    0.9965    0.9965    0.9965
//    1.0065    1.0065    1.0065    1.0065
//    0.9968    0.9968    0.9968    0.9968
//    0.9839    0.9839    0.9839    0.9839
//];
```

4.3.10 `magma_dgetrf_mgpu` - LU factorization in double precision on multiple GPU-s

The function `magma_dgetrf_mgpu` computes in double precision an LU factorization of a general $m \times n$ matrix A using partial pivoting with row interchanges:

$$A = P\ L\ U,$$

where P is a permutation matrix, L is lower triangular with unit diagonal, and U is upper diagonal. The blocks of matrix A to be factored and the blocks of factors L, U are distributed on `num_gpus` devices. The information on the interchanged rows is contained in *ipiv*. See magma-X.Y.Z/src/dgetrf_mgpu.cpp for more details.
Using the obtained factorization one can replace the problem of solving a general linear system by solving two triangular systems with matrices L

and U respectively. The Lapack function `lapackf77_dgetrs` uses the L, U factors copied from `num_gpus` devices to solve in double precision a general linear system

$$A\,X = B.$$

The right hand side B is a matrix defined on the host. On exit it is replaced by the solution.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
#define min(a,b)  (((a)<(b))?(a):(b))
extern "C" magma_int_t
magma_dgetrf_mgpu(magma_int_t num_gpus, magma_int_t m1,
                  magma_int_t n,double **d_la, magma_int_t m,
                  magma_int_t *ipiv, magma_int_t *info);
int main( int argc, char** argv)
{
  magma_init();                              // initialize Magma
  magma_err_t err;
  double   cpu_time, gpu_time;
  magma_int_t  m = 2*8192, n = m;         // a,r - m*n matrices
  magma_int_t  nrhs =100;   // b - n*nrhs, c - m*nrhs matrices
  magma_int_t *ipiv;  // array of indices of interchanged rows
  magma_int_t n2=m*n;                             // size of a,r
  magma_int_t nnrhs=n*nrhs;                         // size of b
  magma_int_t mnrhs=m*nrhs;                         // size of c
  magma_timestr_t  start, end;
  double *a, *r;              // a,r - mxn matrices on the host
  double *b, *c;// b - n*nrhs, c - m*nrhs matrices on the host
  double *d_la[4];//d_la[i] - block of matrix a on i-th device
  double alpha=1.0, beta=0.0;                  // alpha=1,beta=0
  magma_int_t num_gpus= 2, n_local;       // number of gpus = 2
  magma_int_t ione = 1, info;
  magma_int_t i, k, min_mn=min(m,n), nb, nk;
  magma_int_t ldn_local;// m*ldn_local - size of the part of a
  magma_int_t ISEED[4] = {0,0,0,1};           // on i-th device
  nb =magma_get_dgetrf_nb(m);  // optimal blocksize for dgetrf
// allocate memory on cpu
  ipiv=(magma_int_t*)malloc(min_mn*sizeof(magma_int_t));
                                        // host memory for ipiv
  err = magma_dmalloc_cpu(&a,n2);          // host memory for a
  err = magma_dmalloc_pinned(&r,n2);       // host memory for r
  err = magma_dmalloc_pinned(&b,nnrhs);    // host memory for b
  err = magma_dmalloc_pinned(&c,mnrhs);    // host memory for c
// allocate device memory on num_gpus devices
  for(i=0; i<num_gpus; i++){
    n_local = ((n/nb)/num_gpus)*nb;
    if (i < (n/nb)%num_gpus)
       n_local += nb;
```

```
    else if (i == (n/nb)%num_gpus)
       n_local += n%nb;
    ldn_local = ((n_local+31)/32)*32;
    cudaSetDevice(i);
    err = magma_dmalloc(&d_la[i],m*ldn_local); //device memory
  }                                            // on i-th device
  cudaSetDevice(0);
// generate matrices
  lapackf77_dlarnv( &ione, ISEED, &n2, a );        // random a
  lapackf77_dlaset(MagmaUpperLowerStr,&n,&nrhs,&alpha,&alpha,
                           b,&n);// b - n*nrhs matrix of ones
  lapackf77_dlacpy( MagmaUpperLowerStr,&m,&n,a,&m,r,&m);//a->r
  printf("upper left corner of the expected solution:\n");
  magma_dprint(4,4,b,m);      // part of the expected solution
  blasf77_dgemm("N","N",&m,&nrhs,&n,&alpha,a,&m,b,&m,
                &beta,c,&m);          // right hand side c=a*b
// LAPACK version of LU decomposition
  start = get_current_time();
  lapackf77_dgetrf(&m, &n, a, &m, ipiv, &info);
  end = get_current_time();
  cpu_time=GetTimerValue(start,end)/1e3;         // Lapack time
  printf("lapackf77_dgetrf time: %7.5f sec.\n",cpu_time);
// copy the corresponding parts of the matrix r to num_gpus
  for(int j=0; j<n; j+=nb){
    k = (j/nb)%num_gpus;
    cudaSetDevice(k);
    nk = min(nb, n-j);
    magma_dsetmatrix( m, nk,r+j*m,m,
                           d_la[k]+j/(nb*num_gpus)*nb*m, m );
    }
  cudaSetDevice(0);
// MAGMA
// LU decomposition on num_gpus devices with partial  pivoting
// and row interchanges, row i is interchanged with row ipiv(i)
  start = get_current_time();

  magma_dgetrf_mgpu( num_gpus, m, n, d_la, m, ipiv, &info);

  end = get_current_time();
  gpu_time=GetTimerValue(start,end)/1e3;          // Magma time
  printf("magma_dgetrf_mgpu time: %7.5f sec.\n",gpu_time);
// copy the decomposition from num_gpus devices to r on the
// host
  for(int j=0; j<n; j+=nb){
    k = (j/nb)%num_gpus;
    cudaSetDevice(k);
    nk = min(nb, n-j);
    magma_dgetmatrix( m, nk,d_la[k]+j/(nb*num_gpus)*nb*m, m,
                                                  r+j*m,m);
  }
  cudaSetDevice(0);
// solve on the host the system r*x=c;  x overwrites c,
```

```
// using the LU decomposition obtained on num_gpus devices
  lapackf77_dgetrs("N",&m,&nrhs,r,&m,ipiv,c,&m,&info);
// print part of the solution from dgetrf_mgpu and dgetrs
  printf("upper left corner of the solution \n\
  from dgetrf_mgpu+dgetrs:\n");   // part of the solution from
  magma_dprint( 4, 4, c, m);     // magma_dgetrf_mgpu + dgetrs
  free(ipiv);                                  // free host memory
  free(a);                                     // free host memory
  magma_free_pinned(r);                        // free host memory
  magma_free_pinned(b);                        // free host memory
  magma_free_pinned(c);                        // free host memory
  for(i=0; i<num_gpus; i++){
    magma_free(d_la[i] );                  // free device memory
  }
  magma_finalize();                              // finalize Magma
}
// upper left corner of the expected solution:
//[
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//];
// lapackf77_dgetrf time: 11.67350 sec.
//
// magma_dgetrf_mgpu time: 2.33553 sec.
//
// upper left corner of the solution
// from dgetrf_mgpu+dgetrs:
//[
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//];
```

4.3.11 `magma_sgetri_gpu` - inverse matrix in single precision, GPU interface

This function computes in single precision the inverse A^{-1} of a $m \times m$ matrix A:

$$A\,A^{-1} = A^{-1}\,A = I.$$

It uses the LU decomposition with partial pivoting and row interchanges computed by `magma_sgetrf_gpu`. The information on pivots is contained in an array `piv`. The function uses also a workspace array `dwork` of size `ldwork`. The matrix A is defined on the device and on exit it is replaced by its inverse. See `magma-X.Y.Z/src/sgetri_gpu.cpp` for more details.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                              // initialize Magma
  magma_timestr_t  start, end;
  float   gpu_time, *dwork ;                // dwork - workspace
  magma_int_t ldwork;                         //  size of dwork
  magma_int_t *piv, info;  // piv - array of indices of inter-
  magma_int_t m = 8192;         // changed rows; a - mxm matrix
  magma_int_t mm=m*m;                          // size of a, r, c
  float *a;                         // a- mxm matrix on the host
  float *d_a;                 // d_a- mxm matrix a on the device
  float *d_r;                 // d_r- mxm matrix r on the device
  float *d_c;                 // d_c- mxm matrix c on the device
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = {0,0,0,1};                    // seed
  magma_err_t err;
  const float alpha = 1.0;                            // alpha=1
  const float beta = 0.0;                              // beta=0
  ldwork = m * magma_get_sgetri_nb( m );     // workspace size
// allocate matrices
  err = magma_smalloc_cpu( &a , mm );      // host memory for a
  err = magma_smalloc( &d_a,  mm );      // device memory for a
  err = magma_smalloc( &d_r,  mm );      // device memory for r
  err = magma_smalloc( &d_c,  mm );      // device memory for c
  err = magma_smalloc( &dwork, ldwork);// dev. mem. for ldwork
  piv=(magma_int_t*)malloc(m*sizeof(magma_int_t));// host mem.
// generate random matrix a                             // for piv
  lapackf77_slarnv(&ione,ISEED,&mm,a);               // random a
  magma_ssetmatrix( m, m, a,  m, d_a,  m );    // copy a -> d_a
  magmablas_slacpy('A',m,m,d_a,m,d_r,m);     // copy d_a -> d_r
// find the inverse matrix: a_d*X=I using the LU factorization
// with partial pivoting and row interchanges computed by
// magma_sgetrf_gpu; row i is interchanged with row piv(i);
// d_a -mxm  matrix; d_a is overwritten by the inverse
  start = get_current_time();

  magma_sgetrf_gpu( m, m, d_a, m, piv, &info);
  magma_sgetri_gpu(m,d_a,m,piv,dwork,ldwork,&info);

  end = get_current_time();
  gpu_time=GetTimerValue(start,end)/1e3;          // Magma time
  magma_sgemm('N','N',m,m,m,alpha,d_a,m,d_r,m,beta,d_c,m);
  printf("magma_sgetrf_gpu + magma_sgetri_gpu time: %7.5f sec.\
                                                 \n",gpu_time);
  magma_sgetmatrix( m, m, d_c, m, a, m );
  printf("upper left corner of a^-1*a:\n");
  magma_sprint( 4, 4, a, m );                  // part of a^-1*a
  free(a);                                    // free host memory
  free(piv);                                  // free host memory
```

```
  magma_free(d_a);                          // free device memory
  magma_free(d_r);                          // free device memory
  magma_free(d_c);                          // free device memory
  magma_finalize();                             // finalize Magma
  return 0;
}
// magma_sgetrf_gpu + magma_sgetri_gpu time: 2.13416 sec.
//
// upper left corner of a^-1*a:
//[
//    1.0000    0.0000   -0.0000   -0.0000
//    0.0000    1.0000   -0.0000   -0.0000
//    0.0000    0.0000    1.0000    0.0000
//   -0.0000   -0.0000    0.0000    1.0000
//];
```

4.3.12 `magma_dgetri_gpu` - inverse matrix in double precision, GPU interface

This function computes in double precision the inverse A^{-1} of a $m \times m$ matrix A:

$$A\,A^{-1} = A^{-1}\,A = I.$$

It uses the LU decomposition with partial pivoting and row interchanges computed by `magma_dgetrf_gpu`. The information on pivots is contained in an array `piv`. The function uses also a workspace array `dwork` of size `ldwork`. The matrix A is defined on the device and on exit it is replaced by its inverse. See `magma-X.Y.Z/src/dgetri_gpu.cpp` for more details.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                              // initialize magma
  magma_timestr_t  start, end;
  double   gpu_time, *dwork;                // dwork - workspace
  magma_int_t ldwork;                          //  size of dwork
  magma_int_t *piv, info;  // piv - array of indices of inter-
  magma_int_t m = 8192;       // changed rows;  a - mxm matrix
  magma_int_t mm=m*m;                          // size of a, r, c
  double *a;                        // a- mxm matrix on the host
  double *d_a;               // d_a- mxm matrix a on the device
  double *d_r;               // d_r- mxm matrix r on the device
  double *d_c;               // d_c- mxm matrix c on the device
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = { 0,0,0,1 };                  // seed
  magma_err_t err;
  const double alpha = 1.0;                          // alpha=1
  const double beta = 0.0;                            // beta=0
```

```
  ldwork = m * magma_get_dgetri_nb( m );     // workspace size
// allocate matrices
  err = magma_dmalloc_cpu( &a , mm );     // host memory for a
  err = magma_dmalloc( &d_a,  mm );     // device memory for a
  err = magma_dmalloc( &d_r,  mm );     // device memory for c
  err = magma_dmalloc( &d_c,  mm );     // device memory for c
  err = magma_dmalloc( &dwork, ldwork);// dev. mem. for ldwork
  piv=(magma_int_t*)malloc(m*sizeof(magma_int_t));// host mem.
// generate random matrix a                          // for piv
  lapackf77_dlarnv(&ione,ISEED,&mm,a);             // random a
  magma_dsetmatrix( m, m, a,  m, d_a,  m );   // copy a -> d_a
  magmablas_dlacpy('A',m,m,d_a,m,d_r,m);    // copy d_a -> d_r
// find the inverse matrix: a_d*X=I using the LU factorization
// with partial pivoting and row interchanges computed by
// magma_sgetrf_gpu; row i is interchanged with row piv(i);
// d_a -mxm  matrix; d_a is overwritten by the inverse
  start = get_current_time();

  magma_dgetrf_gpu( m, m, d_a, m, piv, &info);
  magma_dgetri_gpu(m,d_a,m,piv,dwork,ldwork,&info);

  end = get_current_time();
  gpu_time=GetTimerValue(start,end)/1e3;         // Magma time
  magma_dgemm('N','N',m,m,m,alpha,d_a,m,d_r,m,beta,d_c,m);
  printf("magma_dgetrf_gpu + magma_dgetri_gpu time: %7.5f sec.\
                                                 \n",gpu_time);
  magma_dgetmatrix( m, m, d_c, m, a, m );
  printf("upper left corner of a^-1*a:\n");
  magma_dprint( 4, 4, a, m );                 // part of a^-1*a
  free(a);                                  // free host memory
  free(piv);                                // free host memory
  magma_free(d_a);                         // free device memory
  magma_free(d_r);                         // free device memory
  magma_free(d_c);                         // free device memory
  magma_finalize();                           // finalize magma
  return 0;
}
// magma_dgetrf_gpu + magma_dgetri_gpu time: 3.63810 sec.
//
// upper left corner of a^-1*a:
//[
//    1.0000   -0.0000   -0.0000   -0.0000
//   -0.0000    1.0000   -0.0000   -0.0000
//    0.0000    0.0000    1.0000   -0.0000
//   -0.0000    0.0000    0.0000    1.0000
//];
```

4.4 Cholesky decomposition and solving systems with positive definite matrices

4.4.1 `magma_sposv` - solve a system with a positive definite matrix in single precision, CPU interface

This function computes in single precision the solution of a real linear system

$$A\,X = B,$$

where A is an $m \times m$ symmetric positive definite matrix and B, X are general $m \times n$ matrices. The Cholesky factorization

$$A = \begin{cases} U^T\,U & \text{in } \texttt{MagmaUpper,'U'} \text{ case,} \\ L\,L^T & \text{in } \texttt{MagmaLower,'L'} \text{ case} \end{cases}$$

is used, where U is an upper triangular matrix and L is a lower triangular matrix. The matrices A, B and the solution X are defined on the host. See magma-X.Y.Z/src/sposv.cpp for more details.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                                  // initialize Magma
  magma_timestr_t  start, end;
  float   gpu_time;
  magma_int_t info, i, j;
  magma_int_t m = 2*8192;                          // a - mxm matrix
  magma_int_t n = 100;                          // b,c - mxn matrices
  magma_int_t mm=m*m;                                 // size of a
  magma_int_t mn=m*n;                              // size of b,c
  float *a;                          // a- mxm matrix on the host
  float *b;                          // b- mxn matrix on the host
  float *c;                          // c- mxn matrix on the host
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = {0,0,0,1};                       // seed
  magma_err_t err;
  const float alpha = 1.0;                               // alpha=1
  const float beta = 0.0;                                 // beta=0
// allocate matrices on the host
  err = magma_smalloc_cpu( &a , mm );      // host memory for a
  err = magma_smalloc_cpu( &b , mn );      // host memory for b
  err = magma_smalloc_cpu( &c , mn );      // host memory for c
// generate  matrices
  lapackf77_slarnv(&ione,ISEED,&mm,a);                // random a
// b - mxn matrix of ones
  lapackf77_slaset(MagmaUpperLowerStr,&m,&n,&alpha,&alpha,b,&m);
// symmetrize a and increase its diagonal
```

```
  for(i=0; i<m; i++) {
    MAGMA_S_SET2REAL(a[i*m+i],(MAGMA_S_REAL(a[i*m+i])+1.*m ) );
                for(j=0; j<i; j++)
                    a[i*m+j] = (a[j*m+i]);
  }
  printf("upper left corner of the expected solution:\n");
  magma_sprint( 4, 4, b, m ); // part of the expected solution
// right hand side c=a*b
  blasf77_sgemm("N","N",&m,&n,&m,&alpha,a,&m,b,&m,&beta,c,&m);
// solve  the linear system a*x=c
// c -mxn matrix, a -mxm  symmetric, positive def. matrix;
// c is overwritten by the solution,
// use the Cholesky factorization  a=L*L^T
  start = get_current_time();

  magma_sposv(MagmaLower,m,n,a,m,c,m,&info);

  end = get_current_time();
  gpu_time=GetTimerValue(start,end)/1e3;
  printf("magma_sposv time: %7.5f sec.\n",gpu_time);  // Magma
  printf("upper left corner of the Magma solution:\n"); //time
  magma_sprint( 4, 4, c, m );    // part of the Magma solution
  free(a);                                  // free host memory
  free(b);                                  // free host memory
  free(c);                                  // free host memory
  magma_finalize();                           // finalize Magma
  return 0;
}
// upper left corner of the expected solution:
//[
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//];
// magma_sposv time: 1.69051 sec.
//
// upper left corner of the Magma solution:
//[
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//];
```

4.4.2 `magma_dposv` - solve a system with a positive definite matrix in double precision, CPU interface

This function computes in double precision the solution of a real linear system

$$A\,X = B,$$

where A is an $m \times m$ symmetric positive definite matrix and B, X are general $m \times n$ matrices. The Cholesky factorization

$$A = \begin{cases} U^T\,U & \text{in } \texttt{MagmaUpper,'U'} \text{ case,} \\ L\,L^T & \text{in } \texttt{MagmaLower,'L'} \text{ case} \end{cases}$$

is used, where U is an upper triangular matrix and L is a lower triangular matrix. The matrices A, B and the solution X are defined on the host. See magma-X.Y.Z/src/dposv.cpp for more details.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                                  // initialize Magma
  magma_timestr_t  start, end;
  double   gpu_time;
  magma_int_t info, i, j;
  magma_int_t m = 2*8192;                          // a - mxm matrix
  magma_int_t n = 100;                          // b,c - mxn matrices
  magma_int_t mm=m*m;                                // size of a
  magma_int_t mn=m*n;                             // size of b,c
  double *a;                          // a- mxm matrix on the host
  double *b;                          // b- mxn matrix on the host
  double *c;                          // c- mxn matrix on the host
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = { 0,0,0,1 };                    // seed
  magma_err_t err;
  const double alpha = 1.0;                             // alpha=1
  const double beta = 0.0;                               // beta=0
// allocate matrices on the host
  err = magma_dmalloc_cpu( &a , mm );      // host memory for a
  err = magma_dmalloc_cpu( &b , mn );      // host memory for b
  err = magma_dmalloc_cpu( &c , mn );      // host memory for c
// generate  matrices a, b;
  lapackf77_dlarnv(&ione,ISEED,&mm,a);               // random a
// b - matrix of ones
  lapackf77_dlaset(MagmaUpperLowerStr,&m,&n,&alpha,&alpha,b,&m);
// symmetrize a and increase its diagonal
  for(i=0; i<m; i++) {
    MAGMA_D_SET2REAL(a[i*m+i],(MAGMA_D_REAL(a[i*m+i])+1.*m ) );
                for(j=0; j<i; j++)
                    a[i*m+j] = (a[j*m+i]);
```

```
  }
  printf("upper left corner of the expected solution:\n");
  magma_dprint( 4, 4, b, m ); // part of the expected solution
// right hand side c=a*b
  blasf77_dgemm("N","N",&m,&n,&m,&alpha,a,&m,b,&m,&beta,c,&m);
// solve  the linear system a*x=c
// c -mxn matrix, a -mxm  symmetric, positive def. matrix;
// c is overwritten by the solution,
// use the Cholesky factorization  a=L*L^T
  start = get_current_time();

  magma_dposv(MagmaLower,m,n,a,m,c,m,&info);

  end = get_current_time();
  gpu_time=GetTimerValue(start,end)/1e3;
  printf("magma_dposv time: %7.5f sec.\n",gpu_time);  // Magma
  printf("upper left corner of the Magma solution:\n"); //time
  magma_dprint( 4, 4, c, m );    // part of the Magma solution
  free(a);                                  // free host memory
  free(b);                                  // free host memory
  free(c);                                  // free host memory
  magma_finalize();                           // finalize Magma
  return 0;
}
// upper left corner of the expected solution:
//[
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//];
// magma_dposv time: 3.47437  sec.
//
// upper left corner of the Magma solution:
//[
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//];
```

4.4.3 magma_sposv_gpu - solve a system with a positive definite matrix in single precision, GPU interface

This function computes in single precision the solution of a real linear system

$$A\,X = B,$$

where A is an $m \times m$ symmetric positive definite matrix and B, X are general $m \times n$ matrices. The Cholesky factorization

$$A = \begin{cases} U^T U & \text{in MagmaUpper,'U' case,} \\ L\, L^T & \text{in MagmaLower,'L' case} \end{cases}$$

is used, where U is an upper triangular matrix and L is a lower triangular matrix. The matrices A, B and the solution X are defined on the device. See magma-X.Y.Z/src/sposv_gpu.cpp for more details.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                                  // initialize Magma
  magma_timestr_t  start, end;
  float   gpu_time;
  magma_int_t info, i, j;
  magma_int_t m = 2*8192;                          // a - mxm matrix
  magma_int_t n = 100;                          // b,c - mxn matrices
  magma_int_t mm=m*m;                                 // size of a
  magma_int_t mn=m*n;                              // size of b,c
  float *a;                           // a- mxm matrix on the host
  float *b;                           // b- mxn matrix on the host
  float *c;                           // c- mxn matrix on the host
  float *d_a;                  // d_a- mxm matrix a on the device
  float *d_c;                  // d_c- mxn matrix c on the device
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = { 0,0,0,1 };                    // seed
  magma_err_t err;
  const float alpha = 1.0;                              // alpha=1
  const float beta = 0.0;                                // beta=0
// allocate matrices
  err = magma_smalloc_cpu( &a , mm );      // host memory for a
  err = magma_smalloc_cpu( &b , mn );      // host memory for b
  err = magma_smalloc_cpu( &c , mn );      // host memory for c
  err = magma_smalloc( &d_a,  mm );      // device memory for a
  err = magma_smalloc( &d_c,  mn );      // device memory for c
// generate  matrices
  lapackf77_slarnv(&ione,ISEED,&mm,a);              // random a
// b - mxn matrix of ones
  lapackf77_slaset(MagmaUpperLowerStr,&m,&n,&alpha,&alpha,b,&m);
// symmetrize a and increase its diagonal
  for(i=0; i<m; i++) {
    MAGMA_S_SET2REAL(a[i*m+i],(MAGMA_S_REAL(a[i*m+i])+1.*m ) );
                for(j=0; j<i; j++)
                    a[i*m+j] = (a[j*m+i]);
  }
  printf("upper left corner of the expected solution:\n");
  magma_sprint( 4, 4, b, m ); // part of the expected solution
```

```
// right hand side c=a*b
  blasf77_sgemm("N","N",&m,&n,&m,&alpha,a,&m,b,&m,&beta,c,&m);
  magma_ssetmatrix( m, m, a,  m, d_a,  m );   // copy a -> d_a
  magma_ssetmatrix( m, n, c,  m, d_c,  m );   // copy c -> d_c
// solve  the linear system d_a*x=d_c
// d_c -mxn matrix,  d_a -mxm  symmetric, positive def. matrix;
// d_c is overwritten by the solution
// use the Cholesky factorization  d_a=L*L^T
  start = get_current_time();

  magma_sposv_gpu(MagmaLower,m,n,d_a,m,d_c,m,&info);

  end = get_current_time();
  gpu_time=GetTimerValue(start,end)/1e3;         // Magma time
  printf("magma_sposv_gpu time: %7.5f sec.\n",gpu_time);
  magma_sgetmatrix( m, n, d_c, m, c, m );
  printf("upper left corner of the Magma solution:\n");
  magma_sprint( 4, 4, c, m );    // part of the Magma solution
  free(a);                                  // free host memory
  free(b);                                  // free host memory
  free(c);                                  // free host memory
  magma_free(d_a);                         // free device memory
  magma_free(d_c);                         // free device memory
  magma_finalize();                            // finalize Magma
  return 0;
}
// upper left corner of the expected solution:
//[
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//];
// magma_sposv_gpu time: 0.97214 sec.
//
// upper left corner of the Magma solution:
//[
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//];
```

4.4.4 magma_dposv_gpu - solve a system with a positive definite matrix in double precision, GPU interface

This function computes in double precision the solution of a real linear system

$$A\,X = B,$$

where A is an $m \times m$ symmetric positive definite matrix and B, X are general $m \times n$ matrices. The Cholesky factorization

$$A = \begin{cases} U^T U & \text{in } \texttt{MagmaUpper,'U'} \text{ case,} \\ L\, L^T & \text{in } \texttt{MagmaLower,'L'} \text{ case} \end{cases}$$

is used, where U is an upper triangular matrix and L is a lower triangular matrix. The matrices A, B and the solution X are defined on the device. See magma-X.Y.Z/src/dposv_gpu.cpp for more details.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                              // initialize Magma
  magma_timestr_t  start, end;
  float gpu_time;
  magma_int_t info, i, j;
  magma_int_t m = 2*8192;                      // a - mxm matrix
  magma_int_t n = 100;                      // b,c - mxn matrices
  magma_int_t mm=m*m;                              // size of a
  magma_int_t mn=m*n;                           // size of b,c
  double *a;                          // a- mxm matrix on the host
  double *b;                          // b- mxn matrix on the host
  double *c;                          // c- mxn matrix on the host
  double *d_a;                // d_a- mxm matrix a on the device
  double *d_c;                // d_c- mxn matrix c on the device
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = { 0,0,0,1 };                   // seed
  magma_err_t err;
  const double alpha = 1.0;                            // alpha=1
  const double beta = 0.0;                              // beta=0
// allocate matrices
  err = magma_dmalloc_cpu( &a , mm );      // host memory for a
  err = magma_dmalloc_cpu( &b , mn );      // host memory for b
  err = magma_dmalloc_cpu( &c , mn );      // host memory for c
  err = magma_dmalloc( &d_a,  mm );      // device memory for a
  err = magma_dmalloc( &d_c,  mn );      // device memory for c
// generate  matrices
  lapackf77_dlarnv(&ione,ISEED,&mm,a);              // random a
// b - mxn matrix of ones
  lapackf77_dlaset(MagmaUpperLowerStr,&m,&n,&alpha,&alpha,b,&m);
// symmetrize a and increase its diagonal
  for(i=0; i<m; i++) {
    MAGMA_D_SET2REAL(a[i*m+i],(MAGMA_D_REAL(a[i*m+i])+1.*m ) );
                for(j=0; j<i; j++)
                    a[i*m+j] = (a[j*m+i]);
  }
  printf("upper left corner of the expected solution:\n");
  magma_dprint( 4, 4, b, m ); // part of the expected solution
```

```
// right hand side c=a*b
  blasf77_dgemm("N","N",&m,&n,&m,&alpha,a,&m,b,&m,&beta,c,&m);
  magma_dsetmatrix( m, m, a,  m, d_a,  m );   // copy a -> d_a
  magma_dsetmatrix( m, n, c,  m, d_c,  m );   // copy c -> d_c
// solve  the linear system d_a*x=d_c
// d_c -mxn matrix,  d_a -mxm  symmetric, positive def. matrix;
// d_c is overwritten by the solution
// use the Cholesky factorization  d_a=L*L^T
  start = get_current_time();

  magma_dposv_gpu(MagmaLower,m,n,d_a,m,d_c,m,&info);

  end = get_current_time();
  gpu_time=GetTimerValue(start,end)/1e3;         // Magma time
  printf("magma_dposv_gpu time: %7.5f sec.\n",gpu_time);
  magma_dgetmatrix( m, n, d_c, m, c, m );
  printf("upper left corner of the solution:\n");
  magma_dprint( 4, 4, c, m );    // part of the Magma solution
  free(a);                                  // free host memory
  free(b);                                  // free host memory
  free(c);                                  // free host memory
  magma_free(d_a);                         // free device memory
  magma_free(d_c);                         // free device memory
  magma_finalize();                           // finalize Magma
  return 0;
}
// upper left corner of the expected solution:
//[
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//];
// magma_dposv_gpu time: 1.94481 sec.
//
// upper left corner of the solution:
//[
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//];
```

4.4.5 `magma_spotrf`, `lapackf77_spotrs` - Cholesky decomposition and solving a system with a positive definite matrix in single precision, CPU interface

The function `magma_spotrf` computes in single precision the Cholesky factorization for a symmetric, positive definite $m \times m$ matrix A:

$$A = \begin{cases} U^T U & \text{in } \texttt{MagmaUpper,'U'} \text{ case,} \\ L\, L^T & \text{in } \texttt{MagmaLower,'L'} \text{ case,} \end{cases}$$

where U is an upper triangular matrix and L is a lower triangular matrix. The matrix A and the factors are defined on the host. See magma-X.Y.Z/src/spotrf.cpp for more details. Using the obtained factorization the function `lapackf77_spotrs` computes on the host in single precision the solution of the linear system

$$A\, X = B,$$

where B, X are general $m \times n$ matrices defined on the host. The solution X overwrites B.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                                  // initialize Magma
  magma_timestr_t  start, end;
  float   gpu_time;
  magma_int_t  info, i, j;
  magma_int_t m = 2*8192;                          // a - mxm matrix
  magma_int_t n = 100;                          // b,c - mxn matrices
  magma_int_t mm=m*m;                                 // size of a
  magma_int_t mn=m*n;                              // size of b,c
  float *a;                          // a- mxm matrix on the host
  float *b;                          // b- mxn matrix on the host
  float *c;                          // c- mxn matrix on the host
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = { 0,0,0,1 };                    // seed
  magma_err_t err;
  const float alpha = 1.0;                              // alpha=1
  const float beta = 0.0;                                // beta=0
// allocate matrices on the host
  err = magma_smalloc_cpu( &a , mm );      // host memory for a
  err = magma_smalloc_cpu( &b , mn );      // host memory for b
  err = magma_smalloc_cpu( &c , mn );      // host memory for c
// generate  matrices
  lapackf77_slarnv(&ione,ISEED,&mm,a);               // random a
// b - m*n matrix of ones
  lapackf77_slaset(MagmaUpperLowerStr,&m,&n,&alpha,&alpha,b,&m);
```

```
// symmetrize a and increase its diagonal
  for(i=0; i<m; i++) {
    MAGMA_S_SET2REAL(a[i*m+i],(MAGMA_S_REAL(a[i*m+i])+1.*m ) );
                for(j=0; j<i; j++)
                    a[i*m+j] = (a[j*m+i]);
  }
  printf("upper left corner of of the expected solution:\n");
  magma_sprint( 4, 4, b, m ); // part of the expected solution
// right hand side c=a*b
  blasf77_sgemm("N","N",&m,&n,&m,&alpha,a,&m,b,&m,&beta,c,&m);
// compute the Cholesky factorization a=L*L^T for a real
// symmetric, positive definite mxm matrix a;
// using this factorization solve  the linear system a*x=c
// for a general mxn matrix c, c is overwritten by the
// solution
  start = get_current_time();

  magma_spotrf(MagmaLower, m, a, m, &info);
  lapackf77_spotrs("L",&m,&n,a,&m,c,&m,&info);

  end = get_current_time();
  gpu_time=GetTimerValue(start,end)/1e3;  // Magma+Lapack time
  printf("magma_spotrf+spotrs time: %7.5f sec.\n",gpu_time);
  printf("upper left corner of the Magma/Lapack solution:\n");
  magma_sprint( 4, 4, c, m ); // part of the Magma/Lapack sol.
  free(a);                                  // free host memory
  free(b);                                  // free host memory
  free(c);                                  // free host memory
  magma_finalize();                           // finalize Magma
  return 0;
}
// upper left corner of of the expected solution:
//[
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//];
// magma_spotrf+spotrs time: 1.98372 sec.
//
// upper left corner of the Magma/Lapack solution:
//[
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//];
```

4.4.6 `magma_dpotrf`, `lapackf77_dpotrs` - Cholesky decomposition and solving a system with a positive definite matrix in double precision, CPU interface

The function `magma_dpotrf` computes in double precision the Cholesky factorization for a symmetric, positive definite $m \times m$ matrix A:

$$A = \begin{cases} U^T U & \text{in } \texttt{MagmaUpper,'U'} \text{ case,} \\ L\, L^T & \text{in } \texttt{MagmaLower,'L'} \text{ case,} \end{cases}$$

where U is an upper triangular matrix and L is a lower triangular matrix. The matrix A and the factors are defined on the host. See magma-X.Y.Z/src/dpotrf.cpp for more details. Using the obtained factorization the function `lapackf77_dpotrs` computes on the host in double precision the solution of the linear system

$$A\, X = B,$$

where B, X are general $m \times n$ matrices defined on the host. The solution X overwrites B.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                                  // initialize Magma
  magma_timestr_t  start, end;
  double   gpu_time;
  magma_int_t info, i, j;
  magma_int_t m = 2*8192;                          // a - mxm matrix
  magma_int_t n = 100;                          // b,c - mxn matrices
  magma_int_t mm=m*m;                                 // size of a
  magma_int_t mn=m*n;                              // size of b,c
  double *a;                          // a- mxm matrix on the host
  double *b;                          // b- mxn matrix on the host
  double *c;                          // c- mxn matrix on the host
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = { 0,0,0,1 };                    // seed
  magma_err_t err;
  const double alpha = 1.0;                             // alpha=1
  const double beta = 0.0;                               // beta=0
// allocate matrices on the host
  err = magma_dmalloc_cpu( &a , mm );      // host memory for a
  err = magma_dmalloc_cpu( &b , mn );      // host memory for b
  err = magma_dmalloc_cpu( &c , mn );      // host memory for c
// generate  matrices a, b;
  lapackf77_dlarnv(&ione,ISEED,&mm,a);                // random a
// b - m*n matrix of ones
  lapackf77_dlaset(MagmaUpperLowerStr,&m,&n,&alpha,&alpha,b,&m);
```

```
// symmetrize a and increase its diagonal
  for(i=0; i<m; i++) {
    MAGMA_D_SET2REAL(a[i*m+i],(MAGMA_D_REAL(a[i*m+i])+1.*m ) );
                for(j=0; j<i; j++)
                    a[i*m+j] = (a[j*m+i]);
  }
  printf("upper left corner of of the expected solution:\n");
  magma_dprint( 4, 4, b, m );// part of the expected solution
// right hand side c=a*b
  blasf77_dgemm("N","N",&m,&n,&m,&alpha,a,&m,b,&m,&beta,c,&m);
// compute the Cholesky factorization a=L*L^T for a real
// symmetric, positive definite mxm matrix a;
// using this factorization solve  the linear system a*x=c
// for a general mxn matrix c, c is overwritten by the
// solution
  start = get_current_time();

  magma_dpotrf(MagmaLower, m, a, m, &info);
  lapackf77_dpotrs("L",&m,&n,a,&m,c,&m,&info);

  end = get_current_time();
  gpu_time=GetTimerValue(start,end)/1e3;  // Magma+Lapack time
  printf("magma_dpotrf+dpotrs time: %7.5f sec.\n",gpu_time);
  printf("upper left corner of the Magma/Lapack solution:\n");
  magma_dprint( 4, 4, c, m ); // part of the Magma/Lapack sol.
  free(a);                                  // free host memory
  free(b);                                  // free host memory
  free(c);                                  // free host memory
  magma_finalize();                           // finalize Magma
  return 0;
}
// upper left corner of of the expected solution:
//[
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//];
// magma_dpotrf+dpotrs time: 3.94396 sec.
//
// upper left corner of the Magma/Lapack solution:
//[
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//];
```

4.4.7 `magma_spotrf_gpu`, `magma_spotrs_gpu` - Cholesky decomposition and solving a system with a positive definite matrix in single precision, GPU interface

The function `magma_spotrf_gpu` computes in single precision the Cholesky factorization for a symmetric, positive definite $m \times m$ matrix A:

$$A = \begin{cases} U^T U & \text{in } \texttt{MagmaUpper,'U'} \text{ case,} \\ L\, L^T & \text{in } \texttt{MagmaLower,'L'} \text{ case,} \end{cases}$$

where U is an upper triangular matrix and L is a lower triangular matrix. The matrix A and the factors are defined on the device. See magma-X.Y.Z/src/spotrf_gpu.cpp for more details. Using the obtained factorization the function `magma_spotrs_gpu` computes on the device in single precision the solution of the linear system

$$A\, X = B,$$

where B, X are general $m \times n$ matrices defined on the device. The solution X overwrites B.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                                  // initialize Magma
  magma_timestr_t  start, end;
  float   gpu_time;
  magma_int_t info, i, j;
  magma_int_t m = 2*8192;                          // a - mxm matrix
  magma_int_t n = 100;                          // b,c - mxn matrices
  magma_int_t mm=m*m;                                 // size of a
  magma_int_t mn=m*n;                              // size of b,c
  float *a;                          // a- mxm matrix on the host
  float *b;                          // b- mxn matrix on the host
  float *c;                          // c- mxn matrix on the host
  float *d_a;                 // d_a- mxm matrix a on the device
  float *d_c;                 // d_c- mxn matrix c on the device
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = {0,0,0,1};                       // seed
  magma_err_t err;
  const float alpha = 1.0;                               // alpha=1
  const float beta = 0.0;                                 // beta=0
// allocate matrices
  err = magma_smalloc_cpu( &a , mm );      // host memory for a
  err = magma_smalloc_cpu( &b , mn );      // host memory for b
  err = magma_smalloc_cpu( &c , mn );      // host memory for c
  err = magma_smalloc( &d_a,  mm );      // device memory for a
  err = magma_smalloc( &d_c,  mn );      // device memory for c
```

```
// generate  matrices
  lapackf77_slarnv(&ione,ISEED,&mm,a);              // random a
  lapackf77_slaset(MagmaUpperLowerStr,&m,&n,&alpha,&alpha,
                           b,&m);   // b - m*n matrix of ones
// symmetrize a and increase its diagonal
  for(i=0; i<m; i++) {
    MAGMA_S_SET2REAL(a[i*m+i],(MAGMA_S_REAL(a[i*m+i])+1.*m ) );
                for(j=0; j<i; j++)
                    a[i*m+j] = (a[j*m+i]);
  }
  printf("upper left corner of of the expected solution:\n");
  magma_sprint( 4, 4, b, m ); // part of the expected solution
// right hand side c=a*b
  blasf77_sgemm("N","N",&m,&n,&m,&alpha,a,&m,b,&m,&beta,c,&m);
  magma_ssetmatrix( m, m, a,  m, d_a,  m );   // copy a -> d_a
  magma_ssetmatrix( m, n, c,  m, d_c,  m );   // copy c -> d_c
// compute the Cholesky factorization d_a=L*L^T for a real
// symmetric, positive definite mxm matrix d_a;
// using this factorization solve  the linear system d_a*x=d_c
// for a general mxn matrix d_c, d_c is overwritten by the
// solution
  start = get_current_time();

  magma_spotrf_gpu(MagmaLower, m, d_a, m, &info);
  magma_spotrs_gpu(MagmaLower,m,n,d_a,m,d_c,m,&info);

  end = get_current_time();
  gpu_time=GetTimerValue(start,end)/1e3;         // Magma time
  printf("magma_spotrf_gpu+magma_spotrs_gpu time: %7.5f sec.\n",
                                                   gpu_time);
  magma_sgetmatrix( m, n, d_c, m, c, m );
  printf("upper left corner of the Magma solution:\n");
  magma_sprint( 4, 4, c, m );    // part of the Magma solution
  free(a);                                  // free host memory
  free(b);                                  // free host memory
  free(c);                                  // free host memory
  magma_free(d_a);                         // free device memory
  magma_free(d_c);                         // free device memory
  magma_finalize();                            // finalize Magma
  return 0;
}
// upper left corner of of the expected solution:
//[
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//];
// magma_spotrf_gpu+magma_spotrs_gpu time: 0.96925 sec.
//
// upper left corner of the Magma solution:
//[
```

```
//    1.0000    1.0000    1.0000    1.0000
//    1.0000    1.0000    1.0000    1.0000
//    1.0000    1.0000    1.0000    1.0000
//    1.0000    1.0000    1.0000    1.0000
//];
```

4.4.8 magma_dpotrf_gpu, magma_dpotrs_gpu - Cholesky decomposition and solving a system with a positive definite matrix in double precision, GPU interface

The function magma_dpotrf_gpu computes in double precision the Cholesky factorization for a symmetric, positive definite $m \times m$ matrix A:

$$A = \begin{cases} U^T\, U & \text{in MagmaUpper,'U' case,} \\ L\, L^T & \text{in MagmaLower,'L' case,} \end{cases}$$

where U is an upper triangular matrix and L is a lower triangular matrix. The matrix A and the factors are defined on the device. See magma-X.Y.Z/src/dpotrf_gpu.cpp for more details. Using the obtained factorization the function magma_dpotrs_gpu computes on the device in double precision the solution of the linear system

$$A\, X = B,$$

where B, X are general $m \times n$ matrices defined on the device. The solution X overwrites B.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                                  // initialize Magma
  magma_timestr_t  start, end;
  double   gpu_time;
  magma_int_t  info, i, j;
  magma_int_t m = 2*8192;                          // a - mxm matrix
  magma_int_t n = 100;                          // b,c - mxn matrices
  magma_int_t mm=m*m;                                 // size of a
  magma_int_t mn=m*n;                               // size of b,c
  double *a;                          // a- mxm matrix on the host
  double *b;                          // b- mxn matrix on the host
  double *c;                          // c- mxn matrix on the host
  double *d_a;                 // d_a- mxm matrix a on the device
  double *d_c;                 // d_c- mxn matrix c on the device
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = {0,0,0,1};                      // seed
  magma_err_t err;
```

```
  const double alpha = 1.0;                          // alpha=1
  const double beta = 0.0;                           // beta=0
// allocate matrices
  err = magma_dmalloc_cpu( &a , mm );      // host memory for a
  err = magma_dmalloc_cpu( &b , mn );      // host memory for b
  err = magma_dmalloc_cpu( &c , mn );      // host memory for c
  err = magma_dmalloc( &d_a,  mm );      // device memory for a
  err = magma_dmalloc( &d_c,  mn );      // device memory for c
// generate  matrices
  lapackf77_dlarnv(&ione,ISEED,&mm,a);              // random a
  lapackf77_dlaset(MagmaUpperLowerStr,&m,&n,&alpha,&alpha,
                           b,&m);   // b - m*n matrix of ones
// symmetrize a and increase its diagonal
  for(i=0; i<m; i++) {
    MAGMA_D_SET2REAL(a[i*m+i],(MAGMA_D_REAL(a[i*m+i])+1.*m ) );
                for(j=0; j<i; j++)
                    a[i*m+j] = (a[j*m+i]);
  }
  printf("upper left corner of of the expected solution:\n");
  magma_dprint( 4, 4, b, m ); // part of the expected solution
// right hand side c=a*b
  blasf77_dgemm("N","N",&m,&n,&m,&alpha,a,&m,b,&m,&beta,c,&m);
  magma_dsetmatrix( m, m, a,  m, d_a,  m );   // copy a -> d_a
  magma_dsetmatrix( m, n, c,  m, d_c,  m );   // copy c -> d_c
// compute the Cholesky factorization d_a=L*L^T for a real
// symmetric, positive definite mxm matrix d_a;
// using this factorization solve  the linear system d_a*x=d_c
// for a general mxn matrix d_c, d_c is overwritten by the
// solution
  start = get_current_time();

  magma_dpotrf_gpu(MagmaLower, m, d_a, m, &info);
  magma_dpotrs_gpu(MagmaLower,m,n,d_a,m,d_c,m,&info);

  end = get_current_time();
  gpu_time=GetTimerValue(start,end)/1e3;         // Magma time
  printf("magma_dpotrf_gpu+magma_dpotrs_gpu time: %7.5f sec.\n",
                                                  gpu_time);
  magma_dgetmatrix( m, n, d_c, m, c, m );     // copy d_c -> c
  printf("upper left corner of the solution:\n");
  magma_dprint( 4, 4, c, m );    // part of the Magma solution
  free(a);                                    // free host memory
  free(b);                                    // free host memory
  free(c);                                    // free host memory
  magma_free(d_a);                           // free device memory
  magma_free(d_c);                           // free device memory
  magma_finalize();                              // finalize Magma
  return 0;
}
// upper left corner of of the expected solution:
//[
//   1.0000   1.0000   1.0000   1.0000
```

```
//    1.0000    1.0000    1.0000    1.0000
//    1.0000    1.0000    1.0000    1.0000
//    1.0000    1.0000    1.0000    1.0000
//];
// magma_dpotrf_gpu+magma_dpotrs_gpu time: 1.94403 sec.
//
// upper left corner of the solution:
//[
//    1.0000    1.0000    1.0000    1.0000
//    1.0000    1.0000    1.0000    1.0000
//    1.0000    1.0000    1.0000    1.0000
//    1.0000    1.0000    1.0000    1.0000
//];
```

4.4.9 `magma_spotrf_mgpu`, `lapackf77_spotrs` - Cholesky decomposition on multiple GPUs and solving a system with a positive definite matrix in single precision

The function `magma_spotrf_mgpu` computes in single precision the Cholesky factorization for a symmetric, positive definite $m \times m$ matrix A:

$$A = \begin{cases} U^T U & \text{in } \texttt{MagmaUpper,'U'} \text{ case,} \\ L\, L^T & \text{in } \texttt{MagmaLower,'L'} \text{ case,} \end{cases}$$

where U is an upper triangular matrix and L is a lower triangular matrix. The matrix A and the factors are distributed to `num_gpus` devices. See magma-X.Y.Z/src/spotrf_mgpu.cpp for more details. Using the obtained factorization, after gathering the factors to some common matrix on the host, the function `lapackf77_spotrs` computes in single precision on the host the solution of the linear system

$$A\, X = B,$$

where B, X are general $m \times n$ matrices defined on the host. The solution X overwrites B.

```
#include <stdio.h>
#include <cublas.h>
#include "magma.h"
#include "magma_lapack.h"
#define min(a,b)  (((a)<(b))?(a):(b))
extern "C" magma_int_t
magma_spotrf_mgpu(int num_gpus, char uplo, magma_int_t n,
           float **d_la, magma_int_t ldda, magma_int_t *info);
int main( int argc, char** argv) {
  magma_init();                              // initialize Magma
  magma_setdevice(0);                                // device 0
  float   cpu_time,gpu_time;
```

```
  magma_err_t err;
  magma_timestr_t  start, end;
  magma_int_t m = 2*8192;                 // a,r - m*m matrices
  magma_int_t nrhs =100;               // b,c - m*nrhs matrices
  magma_int_t mm=m*m;                           // size of a,r
  magma_int_t mnrhs=m*nrhs;                     // size of b,c
  magma_int_t num_gpus=2;                    // number of GPUs
  float *a, *r;              // a,r - m*n matrices on the host
  float *b, *c;           // b,c - m*nrhs matrices on the host
  float *d_la[4]; // d_la[i] - part of matrix a on i-th device
  float alpha=1.0, beta=0.0;
  magma_int_t  mb, nb, nk;
  magma_int_t lda=m, ldda, n_local, ldn_local;
  magma_int_t i, j, k, info;
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = {0,0,0,1};
  nb = magma_get_spotrf_nb(m); // optimal blocksize for spotrf
  mb = nb;
  n_local = nb*(1+m/(nb*num_gpus)) * mb*((m+mb-1)/mb);
  ldda=n_local;  //size of the part d_l[i] of a on i-th device
// allocate host memory for matrices
  err = magma_smalloc_pinned(&a,mm);      // host memory for a
  err = magma_smalloc_pinned(&r,mm);      // host memory for r
  err = magma_smalloc_pinned(&b,mnrhs);   // host memory for b
  err = magma_smalloc_pinned(&c,mnrhs);   // host memory for c
// allocate blocks of  matrix on num_gpus devices
  for(i=0; i<num_gpus; i++){
    magma_setdevice(i);
    err = magma_smalloc(&d_la[i],ldda);        //device memory
  }                                          // on i-th device
  magma_setdevice(0);
  lapackf77_slarnv( &ione, ISEED, &mm, a );        // random a
  lapackf77_slaset(MagmaUpperLowerStr,&m,&nrhs,&alpha,&alpha,
                           b,&m);// b - m*nrhs matrix of ones
// Symmetrize a and increase its diagonal
  for(i=0; i<m; i++) {
  MAGMA_S_SET2REAL(a[i*lda+i],(MAGMA_S_REAL(a[i*lda+i])+1.*m));
    for(j=0; j<i; j++)
      a[i*lda+j] = (a[j*lda+i]);
  }
// copy a -> r
  lapackf77_slacpy( MagmaUpperLowerStr,&m,&m,a,&lda,r,&lda);
  printf("upper left corner of the expected solution:\n");
  magma_sprint(4,4,b,m);                   // expected solution
  blasf77_sgemm("N","N",&m,&nrhs,&m,&alpha,a,&m,b,&m,&beta,
                           c,&m);   // right hand side c=a*b
// MAGMA
// distribute the matrix a to num_gpus devices
// going through each block-row
  ldda = (1+m/(nb*num_gpus))*nb;
  for(j=0; j<m; j+=nb){
    k = (j/nb)%num_gpus;
```

```
    magma_setdevice(k);
    nk = min(nb, m-j);
    magma_ssetmatrix( nk, m, a+j,lda,
                          d_la[k]+j/(nb*num_gpus)*nb, ldda );
  }
  magma_setdevice(0);
  start = get_current_time();
// compute the Cholesky factorization a=L*L^T on num_gpus
// devices, blocks of a and blocks of factors are  distributed
// to num_gpus devices

  magma_spotrf_mgpu(num_gpus, MagmaLower, m, d_la, ldda, &info);

  end = get_current_time();
  gpu_time=GetTimerValue(start,end)/1e3;          // Magma time
  printf("magma_spotrf_mgpu time: %7.5f sec.\n", gpu_time);
// gather the resulting matrix from num_gpus devices to r
  for(j=0; j<m; j+=nb){
    k = (j/nb)%num_gpus;
    magma_setdevice(k);
    nk = min(nb, m-j);
    magma_sgetmatrix( nk, m,d_la[k]+j/(nb*num_gpus)*nb,ldda,
                                                 r+j,lda );
  }
  magma_setdevice(0);
// use LAPACK to obtain the solution of a*x=c
  lapackf77_spotrs("L",&m,&nrhs,r,&m,c,&m,&info);
  printf("upper left corner of the Magma/Lapack solution \n\
  from spotrf_mgpu+spotrs:\n");
  magma_sprint( 4, 4, c, m);          // Magma/Lapack solution
// LAPACK version of spotrf  for time comparison
  start = get_current_time();
  lapackf77_spotrf("L", &m, a, &lda, &info);
  end = get_current_time();
  cpu_time=GetTimerValue(start,end)/1e3;        // Lapack time
  printf("Lapack spotrf time: %7.5f sec.\n",cpu_time);
  magma_free_pinned(a);                    // free host memory
  magma_free_pinned(r);                    // free host memory
  magma_free_pinned(b);                    // free host memory
  magma_free_pinned(c);                    // free host memory
  for(i=0; i<num_gpus; i++){
    magma_setdevice(i);
    magma_free(d_la[i] );                // free device memory
  }
  magma_finalize();                          // finalize Magma
}
// upper left corner of the expected solution:
//[
//    1.0000    1.0000    1.0000    1.0000
//    1.0000    1.0000    1.0000    1.0000
//    1.0000    1.0000    1.0000    1.0000
//    1.0000    1.0000    1.0000    1.0000
```

```
//];
// magma_spotrf_mgpu time: 0.52645 sec.
//
// upper left corner of the Magma/Lapack solution
//  from spotrf_mgpu+spotrs:
//[
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//];
// Lapack spotrf time: 3.33889 sec.
```

4.4.10 `magma_dpotrf_mgpu, lapackf77_dpotrs` - Cholesky decomposition and solving a system with a positive definite matrix in double precision on multiple GPUs

The function `magma_dpotrf_mgpu, lapackf77_dpotrs` computes in double precision the Cholesky factorization for a symmetric, positive definite $m \times m$ matrix A:

$$A = \begin{cases} U^T\, U & \text{in } \texttt{MagmaUpper,'U'} \text{ case,} \\ L\, L^T & \text{in } \texttt{MagmaLower,'L'} \text{ case,} \end{cases}$$

where U is an upper triangular matrix and L is a lower triangular matrix. The matrix A and the factors are distributed to `num_gpus` devices. See magma-X.Y.Z/src/dpotrf_mgpu.cpp for more details. Using the obtained factorization, after gathering the factors to some common matrix on the host, the function `lapackf77_dpotrs` computes in double precision on the host the solution of the linear system

$$A\, X = B,$$

where B, X are general $m \times n$ matrices defined on the host. The solution X overwrites B.

```
#include <stdio.h>
#include <cublas.h>
#include "magma.h"
#include "magma_lapack.h"
#define min(a,b)  (((a)<(b))?(a):(b))
extern "C" magma_int_t
magma_dpotrf_mgpu(int num_gpus, char uplo, magma_int_t n,
           double **d_la, magma_int_t ldda, magma_int_t *info);
int main( int argc, char** argv) {
  magma_init();                              // initialize Magma
  magma_setdevice(0);                                // device 0
  double   cpu_time,gpu_time;
  magma_err_t err;
```

```
  magma_timestr_t  start, end;
  magma_int_t m = 2*8192;                  // a,r - m*m matrices
  magma_int_t nrhs =100;                // b,c - m*nrhs matrices
  magma_int_t mm=m*m;                              // size of a,r
  magma_int_t mnrhs=m*nrhs;                        // size of b,c
  magma_int_t num_gpus=2;                       // number of GPUs
  double *a, *r;              // a,r - m*n matrices on the host
  double *b, *c;           // b,c - m*nrhs matrices on the host
  double *d_la[4];// d_la[i] - part of matrix a on i-th device
  double alpha=1.0, beta=0.0;
  magma_int_t  mb, nb, nk;
  magma_int_t lda=m, ldda, n_local, ldn_local;
  magma_int_t i, j, k, info;
  magma_int_t ione = 1 ;
  magma_int_t ISEED[4] = {0,0,0,1};
  nb = magma_get_dpotrf_nb(m); // optimal blocksize for dpotrf
  mb = nb;
  n_local = nb*(1+m/(nb*num_gpus)) * mb*((m+mb-1)/mb);
  ldda = n_local;
// allocate host memory for matrices
  err = magma_dmalloc_pinned(&a,mm);       // host memory for a
  err = magma_dmalloc_pinned(&r,mm);       // host memory for r
  err = magma_dmalloc_pinned(&b,mnrhs);    // host memory for b
  err = magma_dmalloc_pinned(&c,mnrhs);    // host memory for c
// allocate local matrix on the devices
  for(i=0; i<num_gpus; i++){
    magma_setdevice(i);
    err = magma_dmalloc(&d_la[i],ldda);         //device memory
  }                                          // on i-th device
  magma_setdevice(0);
  lapackf77_dlarnv( &ione, ISEED, &mm, a );         // random a
  lapackf77_dlaset(MagmaUpperLowerStr,&m,&nrhs,&alpha,&alpha,
                           b,&m);// b - m*nrhs matrix of ones
// Symmetrize a and increase its diagonal
  for(i=0; i<m; i++) {
  MAGMA_D_SET2REAL(a[i*lda+i],(MAGMA_D_REAL(a[i*lda+i])+1.*m));
    for(j=0; j<i; j++)
      a[i*lda+j] = (a[j*lda+i]);
  }
// copy a -> r
  lapackf77_dlacpy( MagmaUpperLowerStr,&m,&m,a,&lda,r,&lda);
  printf("upper left corner of the expected solution:\n");
  magma_dprint(4,4,b,m);                    // expected solution
  blasf77_dgemm("N","N",&m,&nrhs,&m,&alpha,a,&m,b,&m,&beta,
                                  c,&m);  // right hand c=a*b
// MAGMA
// distribute the matrix a to num_gpus devices
// going through each block-row
  ldda = (1+m/(nb*num_gpus))*nb;
  for(j=0; j<m; j+=nb){
    k = (j/nb)%num_gpus;
    magma_setdevice(k);
```

```
    nk = min(nb, m-j);
    magma_dsetmatrix( nk, m, a+j,lda,
                      d_la[k]+j/(nb*num_gpus)*nb, ldda );
  }
  magma_setdevice(0);
  start = get_current_time();
// compute the Cholesky factorization a=L*L^T on num_gpus
// devices, blocks of a and blocks of factors are  distributed
// to num_gpus devices

  magma_dpotrf_mgpu(num_gpus, MagmaLower, m, d_la, ldda, &info);

  end = get_current_time();
  gpu_time=GetTimerValue(start,end)/1e3;          // Magma time
  printf("magma_dpotrf_mgpu time: %7.5f sec.\n", gpu_time);
// gather the resulting matrix from num_gpus devices to r
  for(j=0; j<m; j+=nb){
    k = (j/nb)%num_gpus;
    magma_setdevice(k);
    nk = min(nb, m-j);
    magma_dgetmatrix( nk, m,d_la[k]+j/(nb*num_gpus)*nb,ldda,
                                                r+j,lda );
  }
  magma_setdevice(0);
// use LAPACK to obtain the solution of a*x=c
  lapackf77_dpotrs("L",&m,&nrhs,r,&m,c,&m,&info);
  printf("upper left corner of the solution \n\
  from dpotrf_mgpu+dpotrs:\n");
  magma_dprint( 4, 4, c, m);          // Magma/Lapack solution
// LAPACK version of dpotrf for time comparison
  start = get_current_time();
  lapackf77_dpotrf("L", &m, a, &lda, &info);
  end = get_current_time();
  cpu_time=GetTimerValue(start,end)/1e3;        // Lapack time
  printf("Lapack dpotrf time: %7.5f sec.\n",cpu_time);
  magma_free_pinned(a);                     // free host memory
  magma_free_pinned(r);                     // free host memory
  magma_free_pinned(b);                     // free host memory
  magma_free_pinned(c);                     // free host memory
  for(i=0; i<num_gpus; i++){
    magma_setdevice(i);
    magma_free(d_la[i] );                // free device memory
  }
  magma_finalize();                          // finalize Magma
}
// upper left corner of the expected solution:
//[
//    1.0000    1.0000    1.0000    1.0000
//    1.0000    1.0000    1.0000    1.0000
//    1.0000    1.0000    1.0000    1.0000
//    1.0000    1.0000    1.0000    1.0000
//];
```

```
// magma_dpotrf_mgpu time: 1.10033 sec.
//
// upper left corner of the solution
//  from dpotrf_mgpu+dpotrs:
//[
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//];
// Lapack dpotrf time: 6.39562 sec.
```

4.4.11 magma_spotri - invert a symmetric positive definite matrix in single precision, CPU interface

This function computes in single precision the inverse A^{-1} of a $m \times m$ symmetric, positive definite matrix A:

$$A\,A^{-1} = A^{-1}\,A = I.$$

It uses the Cholesky decomposition:

$$A = \begin{cases} U^T\,U & \text{in MagmaUpper,'U' case,} \\ L\,L^T & \text{in MagmaLower,'L' case,} \end{cases}$$

computed by magma_spotrf. The matrix A is defined on the host and on exit it is replaced by its inverse. See magma-X.Y.Z/src/spotri.cpp for more details.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                              // initialize Magma
  magma_timestr_t  start, end;
  float   gpu_time ;
  magma_int_t  info, i, j;
  magma_int_t m = 8192;                       // a - mxm matrix
  magma_int_t mm=m*m;                        // size of a, r, c
  float *a;                        // a- mxm matrix on the host
  float *r;                        // r- mxm matrix on the host
  float *c;                        // c- mxm matrix on the host
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = {0,0,0,1};                    // seed
  magma_err_t err;
  const float alpha = 1.0;                           // alpha=1
  const float beta = 0.0;                             // beta=0
// allocate matrices on the host
```

```
  err = magma_smalloc_cpu( &a , mm );      // host memory for a
  err = magma_smalloc_cpu( &r , mm );      // host memory for r
  err = magma_smalloc_cpu( &c , mm );      // host memory for c
// generate random matrix a
  lapackf77_slarnv(&ione,ISEED,&mm,a);              // random a
// symmetrize a and increase its diagonal
  for(i=0; i<m; i++) {
    MAGMA_S_SET2REAL(a[i*m+i],(MAGMA_S_REAL(a[i*m+i])+1.*m ) );
                for(j=0; j<i; j++)
                    a[i*m+j] = (a[j*m+i]);
  }
  lapackf77_slacpy(MagmaUpperLowerStr,&m,&m,a,&m,r,&m);// a->r
// find the inverse matrix a^-1: a*X=I for mxm  symmetric,
// positive definite matrix a using the Cholesky decomposition
// obtained by magma_spotrf;  a is overwritten by the inverse
  start = get_current_time();

  magma_spotrf(MagmaLower,m,a,m,&info);
  magma_spotri(MagmaLower,m,a,m,&info);

  end = get_current_time();
  gpu_time=GetTimerValue(start,end)/1e3;          // Magma time
  printf("magma_spotrf + magma_spotri time: %7.5f sec.\
                                                \n",gpu_time);
// compute a^-1*a
  blasf77_ssymm("L","L",&m,&m,&alpha,a,&m,r,&m,&beta,c,&m);
  printf("upper left corner of a^-1*a:\n");
  magma_sprint( 4, 4, c, m );                  // part of a^-1*a
  free(a);                                    // free host memory
  free(r);                                    // free host memory
  free(c);                                    // free host memory
  magma_finalize();                            // finalize Magma
  return 0;
}
// magma_spotrf + magma_spotri time: 2.02029 sec.
//
// upper left corner of a^-1*a:
//[
//    1.0000    0.0000   -0.0000    0.0000
//    0.0000    1.0000    0.0000    0.0000
//   -0.0000    0.0000    1.0000   -0.0000
//    0.0000   -0.0000   -0.0000    1.0000
//];
```

4.4.12 `magma_dpotri` - invert a positive definite matrix in double precision, CPU interface

This function computes in double precision the inverse A^{-1} of a $m \times m$ symmetric, positive definite matrix A:

$$A\,A^{-1} = A^{-1}\,A = I.$$

It uses the Cholesky decomposition:

$$A = \begin{cases} U^T\, U & \text{in MagmaUpper,'U' case,} \\ L\, L^T & \text{in MagmaLower,'L' case,} \end{cases}$$

computed by magma_dpotrf. The matrix A is defined on the host and on exit it is replaced by its inverse. See magma-X.Y.Z/src/dpotri.cpp for more details.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                                  // initialize Magma
  magma_timestr_t  start, end;
  double   gpu_time ;
  magma_int_t  info, i, j;
  magma_int_t m = 8192;                           // a - mxm matrix
  magma_int_t mm=m*m;                            // size of a, r, c
  double *a;                          // a- mxm matrix on the host
  double *r;                          // r- mxm matrix on the host
  double *c;                          // c- mxm matrix on the host
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = {0,0,0,1};                      // seed
  magma_err_t err;
  const double alpha = 1.0;                             // alpha=1
  const double beta = 0.0;                               // beta=0
// allocate matrices on the host
  err = magma_dmalloc_cpu( &a , mm );      // host memory for a
  err = magma_dmalloc_cpu( &r , mm );      // host memory for r
  err = magma_dmalloc_cpu( &c , mm );      // host memory for c
// generate random matrix a
  lapackf77_dlarnv(&ione,ISEED,&mm,a);               // random a
// symmetrize a and increase its diagonal
  for(i=0; i<m; i++) {
    MAGMA_D_SET2REAL(a[i*m+i],(MAGMA_D_REAL(a[i*m+i])+1.*m ) );
                for(j=0; j<i; j++)
                    a[i*m+j] = (a[j*m+i]);
  }
  lapackf77_dlacpy(MagmaUpperLowerStr,&m,&m,a,&m,r,&m);// a->r
// find the inverse matrix a^-1: a*X=I for mxm  symmetric,
// positive definite matrix a using the Cholesky decomposition
// obtained by magma_spotrf;  a is overwritten by the inverse
  start = get_current_time();

  magma_dpotrf(MagmaLower,m,a,m,&info);
  magma_dpotri(MagmaLower,m,a,m,&info);

  end = get_current_time();
  gpu_time=GetTimerValue(start,end)/1e3;          // Magma time
```

```
  printf("magma_dpotrf + magma_dpotri time: %7.5f sec.\
                                                \n",gpu_time);
// compute a^-1*a
  blasf77_dsymm("L","L",&m,&m,&alpha,a,&m,r,&m,&beta,c,&m);
  printf("upper left corner of a^-1*a:\n");
  magma_dprint( 4, 4, c, m );                  // part of a^-1*a
  free(a);                                    // free host memory
  free(r);                                    // free host memory
  free(c);                                    // free host memory
  magma_finalize();                             // finalize Magma
  return 0;
}
// magma_dpotrf + magma_dpotri time: 3.09615 sec.
//
// upper left corner of a^-1*a:
//[
//    1.0000   -0.0000    0.0000    0.0000
//   -0.0000    1.0000    0.0000    0.0000
//    0.0000    0.0000    1.0000   -0.0000
//    0.0000    0.0000    0.0000    1.0000
//];
```

4.4.13 magma_spotri_gpu - invert a positive definite matrix in single precision, GPU interface

This function computes in single precision the inverse A^{-1} of a $m \times m$ symmetric, positive definite matrix A:

$$A\,A^{-1} = A^{-1}\,A = I.$$

It uses the Cholesky decomposition:

$$A = \begin{cases} U^T\,U & \text{in MagmaUpper,'U' case,} \\ L\,L^T & \text{in MagmaLower,'L' case,} \end{cases}$$

computed by magma_spotrf_gpu. The matrix A is defined on the device and on exit it is replaced by its inverse. See magma-X.Y.Z/src/spotri_gpu.cpp for more details.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                                // initialize Magma
  magma_timestr_t  start, end;
  float   gpu_time ;
  magma_int_t  info, i, j;
  magma_int_t m = 8192;                          // a - mxm matrix
```

```
  magma_int_t mm=m*m;                          // size of a, r, c
  float *a;                          // a- mxm matrix on the host
  float *d_a;                // d_a- mxm matrix a on the device
  float *d_r;                // d_r- mxm matrix r on the device
  float *d_c;                // d_c- mxm matrix c on the device
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = { 0,0,0,1 };                  // seed
  magma_err_t err;
  const float alpha = 1.0;                            // alpha=1
  const float beta = 0.0;                              // beta=0
// allocate matrices on the host
  err = magma_smalloc_cpu( &a , mm );      // host memory for a
  err = magma_smalloc( &d_a,  mm );      // device memory for a
  err = magma_smalloc( &d_r,  mm );      // device memory for r
  err = magma_smalloc( &d_c,  mm );      // device memory for c
// generate random matrix a
  lapackf77_slarnv(&ione,ISEED,&mm,a);              // random a
// symmetrize a and increase its diagonal
  for(i=0; i<m; i++) {
    MAGMA_S_SET2REAL(a[i*m+i],(MAGMA_S_REAL(a[i*m+i])+1.*m ) );
              for(j=0; j<i; j++)
                  a[i*m+j] = (a[j*m+i]);
  }
  magma_ssetmatrix( m, m, a,  m, d_a,  m );    // copy a -> d_a
  magmablas_slacpy('A',m,m,d_a,m,d_r,m);     // copy d_a -> d_r
// find the inverse matrix (d_a)^-1: d_a*X=I for mxm symmetric
// positive definite matrix d_a using the Cholesky decomposi-
// tion obtained by magma_spotrf_gpu;
// d_a is overwritten by the inverse
  start = get_current_time();

  magma_spotrf_gpu(MagmaLower,m,d_a,m,&info);
  magma_spotri_gpu(MagmaLower,m,d_a,m,&info);

  end = get_current_time();
  gpu_time=GetTimerValue(start,end)/1e3;          // Magma time
// compute a^-1*a
  magma_ssymm('L','L',m,m,alpha,d_a,m,d_r,m,beta,d_c,m);
  printf("magma_spotrf_gpu + magma_spotri_gpu time: %7.5f sec.\
                                                \n",gpu_time);
  magma_sgetmatrix( m, m, d_c, m, a, m );        // copy d_c->a
  printf("upper left corner of a^-1*a:\n");
  magma_sprint( 4, 4, a, m );                  // part of a^-1*a
  free(a);                                    // free host memory
  magma_free(d_a);                           // free device memory
  magma_free(d_r);                           // free device memory
  magma_free(d_c);                           // free device memory
  magma_finalize();                              // finalize Magma
  return 0;
}
// magma_spotrf_gpu + magma_spotri_gpu time: 1.76209 sec.
//
```

```
// upper left corner of a^-1*a:
//[
//    1.0000    0.0000   -0.0000    0.0000
//   -0.0000    1.0000    0.0000    0.0000
//    0.0000    0.0000    1.0000   -0.0000
//    0.0000   -0.0000   -0.0000    1.0000
//];
```

4.4.14 magma_dpotri_gpu - invert a positive definite matrix in double precision, GPU interface

This function computes in double precision the inverse A^{-1} of a $m \times m$ symmetric, positive definite matrix A:

$$A\,A^{-1} = A^{-1}\,A = I.$$

It uses the Cholesky decomposition:

$$A = \begin{cases} U^T\,U & \text{in MagmaUpper,'U' case,} \\ L\,L^T & \text{in MagmaLower,'L' case,} \end{cases}$$

computed by magma_dpotrf_gpu. The matrix A is defined on the device and on exit it is replaced by its inverse. See magma-X.Y.Z/src/dpotri_gpu.cpp for more details.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                              // initialize Magma
  magma_timestr_t  start, end;
  double   gpu_time ;
  magma_int_t  info, i, j;
  magma_int_t m = 8192;                       // a - mxm matrix
  magma_int_t mm=m*m;                        // size of a, r, c
  double *a;                        // a- mxm matrix on the host
  double *d_a;               // d_a- mxm matrix a on the device
  double *d_r;               // d_r- mxm matrix r on the device
  double *d_c;               // d_c- mxm matrix c on the device
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = {0,0,0,1};                    // seed
  magma_err_t err;
  const double alpha = 1.0;                           // alpha=1
  const double beta = 0.0;                             // beta=0
// allocate matrices on the host
  err = magma_dmalloc_cpu( &a , mm );     // host memory for a
  err = magma_dmalloc( &d_a,  mm );      // device memory for a
  err = magma_dmalloc( &d_r,  mm );      // device memory for r
```

```
  err = magma_dmalloc( &d_c,  mm );     // device memory for c
// generate random matrix a
  lapackf77_dlarnv(&ione,ISEED,&mm,a);              // random a
// symmetrize a and increase its diagonal
  for(i=0; i<m; i++) {
    MAGMA_D_SET2REAL(a[i*m+i],(MAGMA_D_REAL(a[i*m+i])+1.*m ) );
                for(j=0; j<i; j++)
                    a[i*m+j] = (a[j*m+i]);
  }
  magma_dsetmatrix( m, m, a,  m, d_a,  m );   // copy a -> d_a
  magmablas_dlacpy('A',m,m,d_a,m,d_r,m);    // copy d_a -> d_r
// find the inverse matrix (d_a)^-1: d_a*X=I for mxm symmetric
// positive definite matrix d_a using the Cholesky factoriza-
// tion obtained by magma_dpotrf_gpu;
// d_a is overwritten by the inverse
  start = get_current_time();

  magma_dpotrf_gpu(MagmaLower,m,d_a,m,&info);
  magma_dpotri_gpu(MagmaLower,m,d_a,m,&info);

  end = get_current_time();
  gpu_time=GetTimerValue(start,end)/1e3;         // Magma time
  magma_dsymm('L','L',m,m,alpha,d_a,m,d_r,m,beta,d_c,m);
  printf("magma_dpotrf_gpu + magma_dpotri_gpu time: %7.5f sec.\
                                                \n",gpu_time);
  magma_dgetmatrix( m, m, d_c, m, a, m );       // copy d_c->a
  printf("upper left corner of a^-1*a:\n");
  magma_dprint( 4, 4, a, m );                 // part of a^-1*a
  free(a);                                  // free host memory
  magma_free(d_a);                         // free device memory
  magma_free(d_r);                         // free device memory
  magma_free(d_c);                         // free device memory
  magma_finalize();                           // finalize Magma
  return 0;
}
// magma_dpotrf_gpu + magma_dpotri_gpu time: 2.50459 sec.
//
// upper left corner of a^-1*a:
//[
//    1.0000   -0.0000   -0.0000    0.0000
//   -0.0000    1.0000   -0.0000   -0.0000
//    0.0000   -0.0000    1.0000   -0.0000
//    0.0000    0.0000   -0.0000    1.0000
//];
```

4.5 QR decomposition and the least squares solution of general systems

4.5.1 magma_sgels_gpu - the least squares solution of a linear system using QR decomposition in single precision, GPU interface

This function solves in single precision the least squares problem

$$\min_X \|A\,X - B\|,$$

where A is an $m \times n$ matrix, $m \geq n$ and B is an $m \times nrhs$ matrix, both defined on the device. In the solution the QR factorization of A is used. The solution X overwrites B. In the current version (1.4) the first argument can take only one value `MagmaNoTrans` (or `'N'`). See magma-X.Y.Z/src/sgels_gpu.cpp for more details.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
#define min(a,b)  (((a)<(b))?(a):(b))
#define max(a,b)  (((a)<(b))?(b):(a))
int main( int argc, char** argv)
{
  magma_init();                              // initialize Magma
  magma_timestr_t start, end;
  float gpu_time, cpu_time;
  magma_int_t m = 2*8192, n = m;               // a - mxn matrix
  magma_int_t nrhs = 100;   // b - n*nrhs, c - m*nrhs matrices
  float *a;                          // a - mxn matrix on the host
  float *b, *c;// b - n*nrhs, c - m*nrhs  matrices on the host
  float *d_a, *d_c;   // d_a - mxn matrix, d_c - m*nrhs matrix
                                           // on the device
  magma_int_t mn = m*n;                            // size of a
  magma_int_t nnrhs=n*nrhs;                        // size of b
  magma_int_t mnrhs=m*nrhs;                        // size of c
  magma_int_t  ldda, lddb;       // leading dim. of d_a and d_c
  float *tau, *hwork, tmp[1]; // used in workspace preparation
  magma_int_t lworkgpu, lhwork;              // workspace sizes
  magma_int_t i, info, min_mn, nb, l1, l2;
  magma_int_t ione = 1;
  const float alpha = 1.0;                           // alpha=1
  const float beta = 0.0;                             // beta=0
  magma_int_t ISEED[4] = {0,0,0,1};                     // seed
  ldda = ((m+31)/32)*32;              // ldda=m if 32 divides m
  lddb = ldda;
  min_mn = min(m, n);
  nb = magma_get_sgeqrf_nb(m); // optimal blocksize for sgeqrf
  lworkgpu = (m-n + nb)*(nrhs+2*nb);
```

```
  magma_smalloc_cpu(&tau,min_mn);        // host memory for tau
  magma_smalloc_cpu(&a,mn);                // host memory for a
  magma_smalloc_cpu(&b,nnrhs);             // host memory for b
  magma_smalloc_cpu(&c,mnrhs);             // host memory for c
  magma_smalloc(&d_a,ldda*n);           // device memory for d_a
  magma_smalloc(&d_c,lddb*nrhs);        // device memory for d_c
// Get size for workspace
  lhwork = -1;
  lapackf77_sgeqrf(&m, &n, a, &m, tau, tmp, &lhwork, &info);
  l1 = (magma_int_t)MAGMA_S_REAL( tmp[0] );
  lhwork = -1;
  lapackf77_sormqr( MagmaLeftStr, MagmaTransStr,
                    &m, &nrhs, &min_mn, a, &m, tau,
                    c, &m, tmp, &lhwork, &info);
  l2 = (magma_int_t)MAGMA_S_REAL( tmp[0] );
  lhwork = max( max( l1, l2 ), lworkgpu );
  magma_smalloc_cpu(&hwork,lhwork); // host memory for worksp.
  lapackf77_slarnv( &ione, ISEED, &mn, a );       // random  a
  lapackf77_slaset(MagmaUpperLowerStr,&n,&nrhs,&alpha,&alpha,
                       b,&m);    // b - m*nrhs matrix of ones
  blasf77_sgemm("N","N",&m,&nrhs,&n,&alpha,a,&m,b,&m,&beta,
                       c,&m);        // right hand side c=a*b
// so the exact solution is the matrix of ones
// MAGMA
  magma_ssetmatrix( m, n, a, m, d_a, ldda );  // copy a -> d_a
  magma_ssetmatrix( m, nrhs, c, m, d_c, lddb );   //  c -> d_c
  start = get_current_time();
// solve the least squares problem  min ||d_a*x-d_c||
// using the QR decomposition, the solution overwrites d_c

  magma_sgels_gpu( MagmaNoTrans, m, n, nrhs, d_a, ldda,
                           d_c, lddb, hwork, lworkgpu, &info);

  end = get_current_time();
  gpu_time = GetTimerValue(start, end)/1e3;
  printf("MAGMA time: %7.3f sec. \n",gpu_time);  // Magma time
 // Get the solution in x
  magma_sgetmatrix( n, nrhs, d_c, lddb, b, n );    // d_c -> b
  printf("upper left corner of of the magma_sgels solution:\n");
  magma_sprint( 4, 4, b, n ); // part of the Magma QR solution
// LAPACK  version of sgels
  start = get_current_time();
  lapackf77_sgels( MagmaNoTransStr, &m, &n, &nrhs,
                   a, &m, c, &m, hwork, &lhwork, &info);
  end = get_current_time();
  cpu_time =  GetTimerValue(start, end)/1e3;
  printf("LAPACK time: %7.3f sec.\n",cpu_time); // Lapack time
  printf("upper left corner of the lapackf77_sgels solution:\n");
  magma_sprint( 4, 4, c, m );// part of the Lapack QR solution
  free(tau);                                  // free host memory
  free(a);                                    // free host memory
  free(b);                                    // free host memory
```

```
  free(c);                                    // free host memory
  free(hwork);                                // free host memory
  magma_free(d_a);                           // free device memory
  magma_free(d_c);                           // free device memory
  magma_finalize( );                           // finalize Magma
  return EXIT_SUCCESS;
}
// MAGMA time:   3.794 sec.
//
// upper left corner of of the magma_sgels solution:
//[
//   0.9942   0.9942   0.9942   0.9942
//   0.9904   0.9904   0.9904   0.9904
//   1.0057   1.0057   1.0057   1.0057
//   0.9955   0.9955   0.9955   0.9955
//];
// LAPACK time:  12.593  sec.
//
// upper left corner of the lapackf77_sgels solution:
//[
//   1.0014   1.0014   1.0014   1.0014
//   0.9976   0.9976   0.9976   0.9976
//   0.9969   0.9969   0.9969   0.9969
//   0.9986   0.9986   0.9986   0.9986
//];
```

4.5.2 magma_dgels_gpu - the least squares solution of a linear system using QR decomposition in double precision, GPU interface

This function solves in double precision the least squares problem

$$\min_X \|A\,X - B\|,$$

where A is an $m \times n$ matrix, $m \geq n$ and B is an $m \times nrhs$ matrix, both defined on the device. In the solution the QR factorization of A is used. The solution X overwrites B. In the current version (1.4) the first argument can take only one value `MagmaNoTrans` (or `'N'`). See magma-X.Y.Z/src/dgels_gpu.cpp for more details.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
#define min(a,b)  (((a)<(b))?(a):(b))
#define max(a,b)  (((a)<(b))?(b):(a))
int main( int argc, char** argv)
{
  magma_init();                               // initialize Magma
```

```
  magma_timestr_t start, end;
  double gpu_time, cpu_time;
  magma_int_t m = 2*8192, n = m;              // a - mxn matrix
  magma_int_t nrhs = 100;   // b - n*nrhs, c - m*nrhs matrices
  double *a;                        // a - mxn matrix on the host
  double *b, *c; // b - n*nrhs, c - m*nrhs  matrix on the host
  double *d_a, *d_c;  // d_a - mxn matrix, d_c - m*nrhs matrix
                                          // on the device
  magma_int_t mn = m*n;                           // size of a
  magma_int_t nnrhs=n*nrhs;                       // size of b
  magma_int_t mnrhs=m*nrhs;                       // size of c
  magma_int_t  ldda, lddb;        // leading dim of d_a and d_c
  double *tau, *hwork, tmp[1];// used in workspace preparation
  magma_int_t lworkgpu, lhwork;              // workspace sizes
  magma_int_t i, info, min_mn, nb, l1, l2;
  magma_int_t ione = 1;
  const double alpha = 1.0;                          // alpha=1
  const double beta = 0.0;                           // beta=0
  magma_int_t ISEED[4] = {0,0,0,1};                    // seed
  ldda = ((m+31)/32)*32;              // ldda=m if 32 divides m
  lddb = ldda;
  min_mn = min(m, n);
  nb = magma_get_dgeqrf_nb(m); // optimal blocksize for dgeqrf
  lworkgpu = (m-n + nb)*(nrhs+2*nb);
  magma_dmalloc_cpu(&tau,min_mn);        // host memory for tau
  magma_dmalloc_cpu(&a,mn);                // host memory for a
  magma_dmalloc_cpu(&b,nnrhs);             // host memory for b
  magma_dmalloc_cpu(&c,mnrhs);             // host memory for c
  magma_dmalloc(&d_a,ldda*n);           // device memory for d_a
  magma_dmalloc(&d_c,lddb*nrhs);        // device memory for d_c
// Get size for workspace
  lhwork = -1;
  lapackf77_dgeqrf(&m, &n, a, &m, tau, tmp, &lhwork, &info);
  l1 = (magma_int_t)MAGMA_D_REAL( tmp[0] );
  lhwork = -1;
  lapackf77_dormqr( MagmaLeftStr, MagmaTransStr,
                    &m, &nrhs, &min_mn, a, &m, tau,
                    c, &m, tmp, &lhwork, &info);
  l2 = (magma_int_t)MAGMA_D_REAL( tmp[0] );
  lhwork = max( max( l1, l2 ), lworkgpu );
  magma_dmalloc_cpu(&hwork,lhwork); // host memory for worksp.
  lapackf77_dlarnv( &ione, ISEED, &mn, a );        // random  a
  lapackf77_dlaset(MagmaUpperLowerStr,&n,&nrhs,&alpha,&alpha,
                     b,&m);     // b - m*nrhs matrix of ones
  blasf77_dgemm("N","N",&m,&nrhs,&n,&alpha,a,&m,b,&m,&beta,
                     c,&m);          // right hand side c=a*b
// so the exact solution is the matrix of ones
// MAGMA
  magma_dsetmatrix( m, n, a, m, d_a, ldda );  // copy a -> d_a
  magma_dsetmatrix( m, nrhs, c, m, d_c, lddb );   //  c -> d_c
  start = get_current_time();
// solve the least squares problem  min ||d_a*x-d_c||
```

```
// using the QR decomposition, the solution overwrites d_c

  magma_dgels_gpu( MagmaNoTrans, m, n, nrhs, d_a, ldda,
                           d_c, lddb, hwork, lworkgpu, &info);

  end = get_current_time();
  gpu_time = GetTimerValue(start, end)/1e3;
  printf("MAGMA time: %7.3f\n",gpu_time);        // Magma time
 // Get the solution in x
  magma_dgetmatrix( n, nrhs, d_c, lddb, b, n );    // d_c -> b
  printf("upper left corner of of the magma_dgels solution:\n");
  magma_dprint( 4, 4, b, n ); // part of the Magma QR solution
// LAPACK  version of sgels
  start = get_current_time();
  lapackf77_dgels( MagmaNoTransStr, &m, &n, &nrhs,
                   a, &m, c, &m, hwork, &lhwork, &info);
  end = get_current_time();
  cpu_time =  GetTimerValue(start, end)/1e3;
  printf("LAPACK time: %7.3f\n",cpu_time);      // Lapack time
  printf("upper left corner of the lapackf77_dgels solution:\n");
  magma_dprint( 4, 4, c, m );// part of the Lapack QR solution
  free(tau);                                  // free host memory
  free(a);                                    // free host memory
  free(b);                                    // free host memory
  free(c);                                    // free host memory
  free(hwork);                                // free host memory
  magma_free(d_a);                           // free device memory
  magma_free(d_c);                           // free device memory
  magma_finalize( );                            // finalize Magma
  return EXIT_SUCCESS;
}
// MAGMA time:   7.267 sec.
//
// upper left corner of of the magma_dgels solution:
//[
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//];
// LAPACK time:   25.239 sec.
//
// upper left corner of the lapackf77_dgels solution:
//[
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//   1.0000   1.0000   1.0000   1.0000
//];
```

4.5.3 `magma_sgeqrf` - QR decomposition in single precision, CPU interface

This function computes in single precision the QR factorization:

$$A = Q\ R,$$

where A is an $m \times n$ general matrix defined on the host, R is upper triangular (upper trapezoidal in general case) and Q is orthogonal. On exit the upper triangle (trapezoid) of A contains R. The orthogonal matrix Q is represented as a product of elementary reflectors $H(1) \dots H(\min(m,n))$, where $H(k) = I - \tau_k v_k v_k^T$. The real scalars τ_k are stored in an array τ and the nonzero components of vectors v_k are stored on exit in parts of columns of A corresponding the lower triangular (trapezoidal) part of A: $v_k(1:k-1) = 0, v_k(k) = 1$ and $v_k(k+1:m)$ is stored in $A(k+1:m,k)$. See `magma-X.Y.Z/src/sgeqrf.cpp` for more details.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
#define min(a,b)  (((a)<(b))?(a):(b))
#define max(a,b)  (((a)<(b))?(b):(a))
int main( int argc, char** argv)
{
  magma_init();                              // initialize Magma
  magma_timestr_t  start, end;
  float gpu_time, cpu_time;
  magma_int_t m = 8192, n = m, n2=m*n;
  float *a, *r;              // a, r - mxn matrices on the host
  float *tau;    // scalars defining the elementary reflectors
  float *hwork, tmp[1];      // hwork - workspace; tmp -used in
  magma_int_t i, info, min_mn,nb;           // workspace query
  magma_int_t ione = 1,lhwork;      // lhwork - workspace size
  magma_int_t ISEED[4] = {0,0,0,1};                    // seed
  min_mn = min(m, n);
  float mzone= MAGMA_S_NEG_ONE;
  float matnorm, work[1];   // used in difference computations
  magma_smalloc_cpu(&tau,min_mn);       // host memory for tau
  magma_smalloc_pinned(&a,n2);            // host memory for a
  magma_smalloc_pinned(&r,n2);            // host memory for r
// Get size for workspace
  nb = magma_get_sgeqrf_nb(m); // optimal blocksize for sgeqrf
  lhwork = -1;
  lapackf77_sgeqrf(&m,&n,a,&m,tau,tmp,&lhwork,&info);
  lhwork = (magma_int_t)MAGMA_S_REAL( tmp[0] );
  lhwork = max(lhwork,max(n*nb,2*nb*nb));
  magma_smalloc_cpu(&hwork,lhwork);
  min_mn= min(m, n);
  lapackf77_slarnv( &ione, ISEED, &n2, a );        // random a
```

```
  lapackf77_slacpy(MagmaUpperLowerStr,&m,&n,a,&m,r,&m);// a->r
// MAGMA
  start = get_current_time();
// compute a QR factorization of a real mxn matrix a
// a=Q*R, Q - orthogonal, R - upper triangular

  magma_sgeqrf( m, n, a, m, tau, hwork, lhwork, &info);

  end = get_current_time();
  gpu_perf = GetTimerValue(start, end)/1e3;
  printf("MAGMA time: %7.3f sec.\n",gpu_perf);  // print Magma
// LAPACK                                                time
  start = get_current_time();
  lapackf77_sgeqrf(&m,&n,r,&m,tau,hwork,&lhwork,&info);
  end = get_current_time();
  cpu_perf =GetTimerValue(start, end)/1e3;
  printf("LAPACK time: %7.3f sec.\n",cpu_perf); //print Lapack
// difference                                             time
  matnorm = lapackf77_slange("f", &m, &n, a, &m, work);
  blasf77_saxpy(&n2, &mzone, a, &ione, r, &ione);
  printf("difference:  %e\n",              // ||a-r||_F/||a||_F
  lapackf77_slange("f", &m, &n, r, &m, work) / matnorm);
// Free memory
  free(tau);                                 // free host memory
  free(hwork);                               // free host memory
  magma_free_pinned(a);                      // free host memory
  magma_free_pinned(r);                      // free host memory
  magma_finalize( );                           // finalize Magma
  return EXIT_SUCCESS;
}
// MAGMA time: 0.774 sec.
//
// LAPACK time: 1.644 sec.
//
// difference:  1.724096e-06
```

4.5.4 magma_dgeqrf - QR decomposition in double precision, CPU interface

This function computes in double precision the QR factorization:

$$A = Q\ R,$$

where A is an $m \times n$ general matrix defined on the host, R is upper triangular (upper trapezoidal in general case) and Q is orthogonal. On exit the upper triangle (trapezoid) of A contains R. The orthogonal matrix Q is represented as a product of elementary reflectors $H(1) \dots H(\min(m,n))$, where $H(k) = I - \tau_k v_k v_k^T$. The real scalars τ_k are stored in an array τ and the nonzero components of vectors v_k are stored on exit in parts of columns of A corresponding the lower triangular (trapezoidal) part of A:

$v_k(1:k-1)=0, v_k(k)=1$ and $v_k(k+1:m)$ is stored in $A(k+1:m,k)$. See magma-X.Y.Z/src/dgeqrf.cpp for more details.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
#define min(a,b)  (((a)<(b))?(a):(b))
#define max(a,b)  (((a)<(b))?(b):(a))
int main( int argc, char** argv)
{
  magma_init();                              // initialize Magma
  magma_timestr_t  start, end;
  double gpu_time, cpu_time;
  magma_int_t m = 8192, n = m, n2=m*n;   // a,r - mxn matrices
  double *a, *r;             // a, r - mxn matrices on the host
  double *tau;   // scalars defining the elementary reflectors
  double *hwork, tmp[1];    // hwork - workspace; tmp -used in
  magma_int_t i, info, min_mn,nb;           // workspace query
  magma_int_t ione = 1,lhwork;      // lhwork - workspace size
  magma_int_t ISEED[4] = {0,0,0,1};                    // seed
  min_mn = min(m, n);
  double mzone= MAGMA_S_NEG_ONE;
  double matnorm, work[1];  // used in difference computations
  magma_dmalloc_cpu(&tau,min_mn);       // host memory for tau
  magma_dmalloc_pinned(&a,n2);            // host memory for a
  magma_dmalloc_pinned(&r,n2);            // host memory for r
// Get size for workspace
  nb = magma_get_dgeqrf_nb(m); // optimal blocksize for dgeqrf
  lhwork = -1;
  lapackf77_dgeqrf(&m,&n,a,&m,tau,tmp,&lhwork,&info);
  lhwork = (magma_int_t)MAGMA_D_REAL( tmp[0] );
  lhwork = max(lhwork,max(n*nb,2*nb*nb));
  magma_dmalloc_cpu(&hwork,lhwork);
  min_mn= min(m, n);
  lapackf77_dlarnv( &ione, ISEED, &n2, a );         // random a
  lapackf77_dlacpy(MagmaUpperLowerStr,&m,&n,a,&m,r,&m);// a->r
// MAGMA
  start = get_current_time();
// compute a QR factorization of a real mxn matrix a
// a=Q*R, Q -orthogonal, R - upper triangular

  magma_dgeqrf( m, n, a, m, tau, hwork, lhwork, &info);

  end = get_current_time();
  gpu_perf = GetTimerValue(start, end)/1e3;
  printf("Magma time: %7.3f\n",gpu_perf); // print Magma time
// LAPACK
  start = get_current_time();
  lapackf77_dgeqrf(&m,&n,r,&m,tau,hwork,&lhwork,&info);
  end = get_current_time();
  cpu_perf =GetTimerValue(start, end)/1e3;
```

```
  printf("Lapack time: %7.3f\n",cpu_perf); //print Lapack time
// difference
  matnorm = lapackf77_dlange("f", &m, &n, a, &m, work);
  blasf77_daxpy(&n2, &mzone, a, &ione, r, &ione);
  printf("difference:  %e\n",              // ||a-r||_F/||a||_F
  lapackf77_dlange("f", &m, &n, r, &m, work) / matnorm);
// Free memory
  free(tau);                                  // free host memory
  free(hwork);                                // free host memory
  magma_free_pinned(a);                       // free host memory
  magma_free_pinned(r);                       // free host memory
  magma_finalize( );                            // finalize Magma
  return EXIT_SUCCESS;
}
// Magma time: 1.279 sec.
//
// Lapack time: 3.220 sec.
//
// difference:  3.050988e-15
```

4.5.5 magma_sgeqrf_gpu - QR decomposition in single precision, GPU interface

This function computes in single precision the QR factorization:

$$A = Q\ R,$$

where A is an $m \times n$ general matrix defined on the device, R is upper triangular (upper trapezoidal in general case) and Q is orthogonal. On exit the upper triangle (trapezoid) of A contains R. The orthogonal matrix Q is represented as a product of elementary reflectors $H(1) \dots H(\min(m,n))$, where $H(k) = I - \tau_k v_k v_k^T$. The real scalars τ_k are stored in an array τ and the nonzero components of vectors v_k are stored on exit in parts of columns of A corresponding to the lower triangular (trapezoidal) part of A: $v_k(1:k-1) = 0, v_k(k) = 1$ and $v_k(k+1:m)$ is stored in $A(k+1:m,k)$. See magma-X.Y.Z/src/sgeqrf_gpu.cpp for more details.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
#define min(a,b)  (((a)<(b))?(a):(b))
int main( int argc, char** argv)
{
  magma_init();                               // initialize Magma
  magma_timestr_t  start, end;
  float gpu_time, cpu_time;
  magma_int_t m = 8192, n = 8192, n2=m*n, ldda;
  float *a, *r;              // a, r - mxn matrices on the host
```

```
  float *d_a;                  // d_a mxn matrix on the device
  float *tau;    // scalars defining the elementary reflectors
  float *hwork, tmp[1];     // hwork - workspace; tmp -used in
  magma_int_t i, info, min_mn;              // workspace query
  magma_int_t ione = 1, lhwork;     // lhwork - workspace size
  magma_int_t ISEED[4] = {0,0,0,1};                   // seed
  ldda  = ((m+31)/32)*32;          // ldda = m if 32 divides m
  min_mn = min(m, n);
  float mzone= MAGMA_S_NEG_ONE;
  float matnorm, work[1];   // used in difference computations
  magma_smalloc_cpu(&tau,min_mn);       // host memory for tau
  magma_smalloc_pinned(&a,n2);            // host memory for a
  magma_smalloc_pinned(&r,n2);            // host memory for r
  magma_smalloc(&d_a,ldda*n);          // device memory for d_a
// Get size for workspace
  lhwork = -1;
  lapackf77_sgeqrf(&m,&n,a,&m,tau,tmp,&lhwork,&info);
  lhwork = (magma_int_t)MAGMA_S_REAL( tmp[0] );
  magma_smalloc_cpu(&hwork,lhwork);    // Lapack version needs
  min_mn= min(m, n);                                 // this array
  lapackf77_slarnv( &ione, ISEED, &n2, a );        // random a
// MAGMA
  magma_ssetmatrix( m, n, a, m, d_a, ldda);   // copy a -> d_a
  start = get_current_time();
// compute a QR factorization of a real mxn matrix d_a
// d_a=Q*R, Q - orthogonal, R - upper triangular

  magma_sgeqrf2_gpu( m, n, d_a, ldda, tau, &info);

  end = get_current_time();
  gpu_time = GetTimerValue(start, end)/1e3;
  printf("MAGMA time: %7.3f sec.\n",gpu_time);   // Magma time
// LAPACK
  start = get_current_time();
  lapackf77_sgeqrf(&m,&n,a,&m,tau,hwork,&lhwork,&info);
  end = get_current_time();
  cpu_time =GetTimerValue(start, end)/1e3;
  printf("LAPACK time: %7.3f sec.\n",cpu_time); // Lapack time
// difference
  magma_sgetmatrix( m, n, d_a, ldda, r, m);   // copy d_a -> r
  matnorm = lapackf77_slange("f", &m, &n, a, &m, work);
  blasf77_saxpy(&n2, &mzone, a, &ione, r, &ione);
  printf("difference:  %e\n",              // ||a-r||_F/||a||_F
  lapackf77_slange("f", &m, &n, r, &m, work) / matnorm);
// Free memory
  free(tau);                                  // free host memory
  free(hwork);                                // free host memory
  magma_free_pinned(a);                       // free host memory
  magma_free_pinned(r);                       // free host memory
  magma_free(d_a);                          // free device memory
  magma_finalize( );                           // finalize Magma
  return EXIT_SUCCESS;
```

```
}
// MAGMA time: 0.607 sec.
//
// LAPACK time: 1.649 sec.
//
// difference:  1.724096e-06
```

4.5.6 magma_dgeqrf_gpu - QR decomposition in double precision, GPU interface

This function computes in double precision the QR factorization:

$$A = Q\ R,$$

where A is an $m \times n$ general matrix defined on the device, R is upper triangular (upper trapezoidal in general case) and Q is orthogonal. On exit the upper triangle (trapezoid) of A contains R. The orthogonal matrix Q is represented as a product of elementary reflectors $H(1) \dots H(\min(m,n))$, where $H(k) = I - \tau_k v_k v_k^T$. The real scalars τ_k are stored in an array τ and the nonzero components of vectors v_k are stored on exit in parts of columns of A corresponding to the lower triangular (trapezoidal) part of A: $v_k(1:k-1) = 0, v_k(k) = 1$ and $v_k(k+1:m)$ is stored in $A(k+1:m,k)$. See magma-X.Y.Z/src/dgeqrf_gpu.cpp for more details.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
#define min(a,b)  (((a)<(b))?(a):(b))
int main( int argc, char** argv)
{
  magma_init();                              // initialize Magma
  magma_timestr_t  start, end;
  double gpu_time, cpu_time;
  magma_int_t m = 8192, n = 8192, n2=m*n, ldda;
  double *a, *r;             // a, r - mxn matrices on the host
  double *d_a;                  // d_a mxn matrix on the device
  double *tau;   // scalars defining the elementary reflectors
  double *hwork, tmp[1];     // hwork - workspace; tmp -used in
  magma_int_t i, info, min_mn;               // workspace query
  magma_int_t ione = 1, lhwork;     // lhwork - workspace size
  magma_int_t ISEED[4] = {0,0,0,1};                    // seed
  ldda  = ((m+31)/32)*32;           // ldda = m if 32 divides m
  min_mn = min(m, n);
  double mzone= MAGMA_D_NEG_ONE;
  double matnorm, work[1];  // used in difference computations
  magma_dmalloc_cpu(&tau,min_mn);       // host memory for tau
  magma_dmalloc_pinned(&a,n2);            // host memory for a
  magma_dmalloc_pinned(&r,n2);            // host memory for r
```

```
  magma_dmalloc(&d_a,ldda*n);          // device memory for d_a
// Get size for workspace
  lhwork = -1;
  lapackf77_dgeqrf(&m,&n,a,&m,tau,tmp,&lhwork,&info);
  lhwork = (magma_int_t)MAGMA_D_REAL( tmp[0] );
  magma_dmalloc_cpu(&hwork,lhwork);     // Lapack version needs
  min_mn= min(m, n);                                 // this array
  lapackf77_dlarnv( &ione, ISEED, &n2, a );          // random a
// MAGMA
  magma_dsetmatrix( m, n, a, m, d_a, ldda);    // copy a -> d_a
  start = get_current_time();
// compute a QR factorization of a real mxn matrix d_a
// d_a=Q*R, Q - orthogonal, R - upper triangular

  magma_dgeqrf2_gpu( m, n, d_a, ldda, tau, &info);

  end = get_current_time();
  gpu_time = GetTimerValue(start, end)/1e3;
  printf("MAGMA time: %7.3f sec.\n",gpu_time);    // Magma time
// LAPACK
  start = get_current_time();
  lapackf77_dgeqrf(&m,&n,a,&m,tau,hwork,&lhwork,&info);
  end = get_current_time();
  cpu_time =GetTimerValue(start, end)/1e3;
  printf("LAPACK time: %7.3f sec.\n",cpu_time); // Lapack time
// difference
  magma_dgetmatrix( m, n, d_a, ldda, r, m);    // copy d_a -> r
  matnorm = lapackf77_dlange("f", &m, &n, a, &m, work);
  blasf77_daxpy(&n2, &mzone, a, &ione, r, &ione);
  printf("difference:  %e\n",                 // ||a-r||_F/||a||_F
  lapackf77_dlange("f", &m, &n, r, &m, work) / matnorm);
// Free memory
  free(tau);                                    // free host memory
  free(hwork);                                  // free host memory
  magma_free_pinned(a);                         // free host memory
  magma_free_pinned(r);                         // free host memory
  magma_free(d_a);                             // free device memory
  magma_finalize( );                              // finalize Magma
  return EXIT_SUCCESS;
}
// MAGMA time: 1.077 sec.
//
// LAPACK time: 3.177 sec.
//
// difference:  5.050988e-15
```

4.5.7 magma_sgeqrf_mgpu - QR decomposition in single precision on multiple GPUs

This function computes in single precision the QR factorization:

$$A = Q\ R,$$

where A is an $m \times n$ general matrix, R is upper triangular (upper trapezoidal in general case) and Q is orthogonal. The matrix A and the factors are distributed on num_gpus devices. On exit the upper triangle (trapezoid) of A contains R. The orthogonal matrix Q is represented as a product of elementary reflectors $H(1) \ldots H(\min(m,n))$, where $H(k) = I - \tau_k v_k v_k^T$. The real scalars τ_k are stored in an array τ and the nonzero components of vectors v_k are stored on exit in parts of columns of A corresponding to the lower triangular (trapezoidal) part of A: $v_k(1:k-1) = 0, v_k(k) = 1$ and $v_k(k+1:m)$ is stored in $A(k+1:m,k)$. See magma-X.Y.Z/src/sgeqrf_mgpu.cpp for more details.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
#define min(a,b)  (((a)<(b))?(a):(b))
#define max(a,b)  (((a)<(b))?(b):(a))
int main( int argc, char** argv)
{
  magma_init();                                  // initialize Magma
  cudaSetDevice(0);
  magma_timestr_t  start, end;
  float cpu_time, gpu_time;
  magma_int_t m = 2*8192, n = m, n2=m*n;
  float *a, *r ;              // a, r - mxn matrices on the host
  float *d_la[4];   // pointers to memory on num_gpus devices
  float *tau;    // scalars defining the elementary reflectors
  float *h_work, tmp[1];    // hwork - workspace; tmp -used in
                                              // workspace query
  magma_int_t n_local[4];    // sizes of local parts of matrix
  magma_int_t i, k, nk, info, min_mn= min(m, n);
  int num_gpus = 2;                               // for two GPUs
  magma_int_t ione = 1, lhwork;     // lhwork - workspace size
  float c_neg_one = MAGMA_S_NEG_ONE;
  float matnorm, work[1];   // used in difference computations
  magma_int_t ISEED[4] = {0,0,0,1};                    // seed
  magma_int_t ldda = ((m+31)/32)*32;//ldda = m if 32 divides m
  magma_int_t nb = magma_get_sgeqrf_nb(m); // optim. blocksize
  printf("Number of GPUs to be used = %d\n", (int) num_gpus);
// Allocate host memory for matrices
  magma_smalloc_cpu(&tau,min_mn);       // host memory for tau
  magma_smalloc_pinned(&a,n2);            // host memory for a
  magma_smalloc_pinned(&r,n2);            // host memory for r
```

```
  for(i=0; i<num_gpus; i++){
    n_local[i] = ((n/nb)/num_gpus)*nb;
    if (i < (n/nb)%num_gpus)
      n_local[i] += nb;
    else if (i == (n/nb)%num_gpus)
      n_local[i] += n%nb;
    cudaSetDevice(i);
    magma_smalloc(&d_la[i],ldda*n_local[i]);   //device memory
                                              // on num_gpus GPUs
    printf("device %2d n_local=%4d\n",(int)i,(int)n_local[i]);
  }
  cudaSetDevice(0);
// Get size for host workspace
  lhwork = -1;
  lapackf77_sgeqrf(&m, &n, a, &m, tau, tmp, &lhwork, &info);
  lhwork = (magma_int_t)MAGMA_S_REAL( tmp[0] );
  magma_smalloc_cpu(&h_work,lhwork);//Lapack sgeqrf needs this
                                                  // array
// Random  matrix a, copy a -> r
  lapackf77_slarnv(&ione,ISEED,&n2,a);
  lapackf77_slacpy(MagmaUpperLowerStr,&m,&n,a,&m,r,&m);
// LAPACK
  start = get_current_time();
// QR decomposition on the host
  lapackf77_sgeqrf(&m,&n,a,&m,tau,h_work,&lhwork,&info);
  end = get_current_time();
  cpu_time = GetTimerValue(start, end)/1e3;
  printf("Lapack sgeqrf time: %7.5f sec.\n",cpu_time);
// MAGMA                                  // print Lapack time
  magma_ssetmatrix_1D_col_bcyclic(m, n, r, m, d_la, ldda,
           num_gpus, nb);  // distribute r -> num_gpus devices
  start = get_current_time();
// QR decomposition on num_gpus devices

  magma_sgeqrf2_mgpu( num_gpus, m, n, d_la, ldda, tau, &info);

  end = get_current_time();
  gpu_time = GetTimerValue(start, end)/1e3;
  printf("Magma sgeqrf_mgpu time: %7.5f sec.\n",gpu_time);
                                          // print Magma time
  magma_sgetmatrix_1D_col_bcyclic(m, n, d_la, ldda, r, m,
             num_gpus, nb);    // gather num_gpus devices -> r
// difference
  matnorm = lapackf77_slange("f", &m, &n, a, &m, work);
  blasf77_saxpy(&n2, &c_neg_one, a, &ione, r, &ione);
  printf("difference: %e\n",
             lapackf77_slange("f",&m,&n,r,&m,work)/matnorm);
  free(tau);                              // free host memory
  free(h_work);                           // free host memory
  magma_free_pinned(a);                   // free host memory
  magma_free_pinned(r);                   // free host memory
  for(i=0; i<num_gpus; i++){
```

```
    magma_setdevice(i);
    magma_free(d_la[i] );                    // free device memory
  }
  magma_finalize( );                          // finalize Magma
  return EXIT_SUCCESS;
}
// Number of GPUs to be used = 2
// device  0 n_local=8192
// device  1 n_local=8192
//
// Lapack sgeqrf time: 11.85600 sec.
//
// Magma sgeqrf_mgpu time: 2.18926 sec.
//
// difference: 2.724096e-06
```

4.5.8 magma_dgeqrf_mgpu - QR decomposition in double precision on multiple GPUs

This function computes in double precision the QR factorization:

$$A = Q\ R,$$

where A is an $m \times n$ general matrix, R is upper triangular (upper trapezoidal in general case) and Q is orthogonal. The matrix A and the factors are distributed on num_gpus devices. On exit the upper triangle (trapezoid) of A contains R. The orthogonal matrix Q is represented as a product of elementary reflectors $H(1) \ldots H(\min(m,n))$, where $H(k) = I - \tau_k v_k v_k^T$. The real scalars τ_k are stored in an array τ and the nonzero components of vectors v_k are stored on exit in parts of columns of A corresponding to the lower triangular (trapezoidal) part of A: $v_k(1:k-1) = 0, v_k(k) = 1$ and $v_k(k+1:m)$ is stored in $A(k+1:m,k)$. See magma-X.Y.Z/src/dgeqrf_mgpu.cpp for more details.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
#define min(a,b)  (((a)<(b))?(a):(b))
#define max(a,b)  (((a)<(b))?(b):(a))
int main( int argc, char** argv)
{
  magma_init();                              // initialize Magma
  cudaSetDevice(0);
  magma_timestr_t  start, end;
  double cpu_time, gpu_time;
  magma_int_t m = 2*8192, n = m, n2=m*n;
  double *a, *r ;              // a, r - mxn matrices on the host
  double *d_la[4];  // pointers to memory on num_gpus devices
```

```
  double *tau;   // scalars defining the elementary reflectors
  double *h_work, tmp[1];   // hwork - workspace; tmp -used in
                                          // workspace query
  magma_int_t n_local[4];    // sizes of local parts of matrix
  magma_int_t i, k, nk, info, min_mn= min(m, n);
  int num_gpus = 2;                              // for two GPUs
  magma_int_t ione = 1, lhwork;     // lhwork - workspace size
  double c_neg_one = MAGMA_D_NEG_ONE;
  double matnorm, work[1];  // used in difference computations
  magma_int_t ISEED[4] = {0,0,0,1};                    // seed
  magma_int_t ldda = ((m+31)/32)*32;//ldda = m if 32 divides m
  magma_int_t nb = magma_get_sgeqrf_nb(m); // optim. blocksize
  printf("Number of GPUs to be used = %d\n", (int) num_gpus);
// Allocate host memory for matrices
  magma_dmalloc_cpu(&tau,min_mn);       // host memory for tau
  magma_dmalloc_pinned(&a,n2);            // host memory for a
  magma_dmalloc_pinned(&r,n2);            // host memory for r
  for(i=0; i<num_gpus; i++){
    n_local[i] = ((n/nb)/num_gpus)*nb;
    if (i < (n/nb)%num_gpus)
      n_local[i] += nb;
    else if (i == (n/nb)%num_gpus)
      n_local[i] += n%nb;
    cudaSetDevice(i);
    magma_dmalloc(&d_la[i],ldda*n_local[i]);   //device memory
                                           // on num_gpus GPUs
    printf("device %2d n_local=%4d\n",(int)i,(int)n_local[i]);
  }
  cudaSetDevice(0);
// Get size for host workspace
  lhwork = -1;
  lapackf77_dgeqrf(&m, &n, a, &m, tau, tmp, &lhwork, &info);
  lhwork = (magma_int_t)MAGMA_D_REAL( tmp[0] );
  magma_dmalloc_cpu(&h_work,lhwork);//Lapack sgeqrf needs this
                                                  // array
// Random  matrix a, copy a -> r
  lapackf77_dlarnv(&ione,ISEED,&n2,a);
  lapackf77_dlacpy(MagmaUpperLowerStr,&m,&n,a,&m,r,&m);// a->r
// LAPACK
  start = get_current_time();
// QR decomposition on the host
  lapackf77_dgeqrf(&m,&n,a,&m,tau,h_work,&lhwork,&info);
  end = get_current_time();
  cpu_time = GetTimerValue(start, end)/1e3;
  printf("Lapack dgeqrf time: %7.5f sec.\n",cpu_time);
// MAGMA                                  // print Lapack time
  magma_dsetmatrix_1D_col_bcyclic(m, n, r, m, d_la, ldda,
           num_gpus, nb);  // distribute r -> num_gpus devices
  start = get_current_time();
// QR decomposition on num_gpus devices

  magma_dgeqrf2_mgpu(num_gpus,m,n,d_la,ldda,tau,&info);
```

```
  end = get_current_time();
  gpu_time = GetTimerValue(start, end)/1e3;
  printf("Magma dgeqrf_mgpu time: %7.5f sec.\n",gpu_time);
                                          // print Magma time
  magma_dgetmatrix_1D_col_bcyclic(m, n, d_la, ldda, r, m,
            num_gpus, nb);    // gather num_gpus devices -> r
// difference
  matnorm = lapackf77_dlange("f", &m, &n, a, &m, work);
  blasf77_daxpy(&n2, &c_neg_one, a, &ione, r, &ione);
  printf("difference: %e\n",
            lapackf77_dlange("f",&m,&n,r,&m,work)/matnorm);
  free(tau);                              // free host memory
  free(h_work);                           // free host memory
  magma_free_pinned(a);                   // free host memory
  magma_free_pinned(r);                   // free host memory
  for(i=0; i<num_gpus; i++){
    magma_setdevice(i);
    magma_free(d_la[i] );               // free device memory
  }
  magma_finalize( );                        // finalize Magma
  return EXIT_SUCCESS;
}
// Number of GPUs to be used = 2
// device  0 n_local=8192
// device  1 n_local=8192
//
// Lapack dgeqrf time: 23.46666  sec.
//
// Magma dgeqrf_mgpu time: 4.06405  sec.
//
// difference: 5.050988e-15
```

4.5.9 magma_sgelqf - LQ decomposition in single precision, CPU interface

This function computes in single precision the LQ factorization:

$$A = L\,Q,$$

where A is an $m \times n$ general matrix defined on the host, L is lower triangular (lower trapezoidal in general case) and Q is orthogonal. On exit the lower triangle (trapezoid) of A contains L. The orthogonal matrix Q is represented as a product of elementary reflectors $H(1) \ldots H(\min(m,n))$, where $H(k) = I - \tau_k v_k v_k^T$. The real scalars τ_k are stored in an array τ and the nonzero components of vectors v_k are stored on exit in parts of rows of A corresponding to the upper triangular (trapezoidal) part of A: $v_k(1:k-1) = 0, v_k(k) = 1$ and $v_k(k+1:n)$ is stored in $A(k, k+1:n)$. See magma-X.Y.Z/src/sgelqf.cpp for more details.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
#define min(a,b)  (((a)<(b))?(a):(b))
#define max(a,b)  (((a)<(b))?(b):(a))
int main( int argc, char** argv){
  magma_init();  magma_init();              // initialize Magma
  magma_timestr_t start, end;
  float  gpu_time, cpu_time;
  magma_int_t m = 4096, n = 4096, n2=m*n;
  float *a, *r;              // a, r - mxn matrices on the host
  float *tau;    // scalars defining the elementary reflectors
  float *h_work, tmp[1];   // h_work - workspace; tmp -used in
                                        // workspace query
  magma_int_t i, info, min_mn, nb;
  magma_int_t ione = 1, lwork;       // lwork - workspace size
  magma_int_t ISEED[4] = {0,0,0,1};                    // seed
  float matnorm, work[1];   // used in difference computations
  float mzone= MAGMA_S_NEG_ONE;
  min_mn = min(m, n);
  nb = magma_get_sgeqrf_nb(m); // optimal blocksize for sgeqrf
  magma_smalloc_cpu(&tau,min_mn);       // host memory for tau
  magma_smalloc_pinned(&a,n2);            // host memory for a
  magma_smalloc_pinned(&r,n2);            // host memory for r
// Get size for host workspace
  lwork = -1;
  lapackf77_sgelqf(&m, &n, a, &m, tau, tmp, &lwork, &info);
  lwork = (magma_int_t)MAGMA_S_REAL( tmp[0] );
  lwork = max( lwork, m*nb );
  magma_smalloc_pinned(&h_work,lwork);
// Random  matrix a,  copy a -> r
  lapackf77_slarnv( &ione, ISEED, &n2, a );
  lapackf77_slacpy(MagmaUpperLowerStr,&m,&n,a,&m,r,&m );
// MAGMA
  start = get_current_time();
// LQ factorization for a real matrix a=L*Q using Magma
// L - lower triangular, Q - orthogonal

  magma_sgelqf(m,n,r,m,tau,h_work,lwork,&info);

  end = get_current_time();
  gpu_time =  GetTimerValue(start, end)/1e3;
  printf("MAGMA time: %7.3f sec.\n",gpu_time);  // print Magma
// LAPACK                                                time
  start = get_current_time();
// LQ factorization for a real matrix a=L*Q on the host
  lapackf77_sgelqf(&m,&n,a,&m,tau,h_work,&lwork,&info);
  end = get_current_time();
  cpu_time =  GetTimerValue(start, end)/1e3;
  printf("LAPACK time: %7.3f sec.\n",cpu_time);// print Lapack
// difference                                            time
```

```
  matnorm = lapackf77_slange("f", &m, &n, a, &m, work);
  blasf77_saxpy(&n2, &mzone, a, &ione, r, &ione);
  printf("difference: %e\n",              // ||a-r||_F/||a||_F
  lapackf77_slange("f", &m, &n, r, &m, work) / matnorm);
// Free emory
  free(tau);                                  // free host memory
  magma_free_pinned(a);                       // free host memory
  magma_free_pinned(r);                       // free host memory
  magma_free_pinned(h_work);                  // free host memory
  magma_finalize( );                            // finalize Magma
  return EXIT_SUCCESS;
}
// MAGMA time:  0.322  sec.
//
// LAPACK time:   2.684 sec.
//
// difference: 1.982540e-06
```

4.5.10 magma_dgelqf - LQ decomposition in double precision, CPU interface

This function computes in double precision the LQ factorization:

$$A = L\,Q,$$

where A is an $m \times n$ general matrix defined on the host, L is lower triangular (lower trapezoidal in general case) and Q is orthogonal. On exit the lower triangle (trapezoid) of A contains L. The orthogonal matrix Q is represented as a product of elementary reflectors $H(1) \ldots H(\min(m,n))$, where $H(k) = I - \tau_k v_k v_k^T$. The real scalars τ_k are stored in an array τ and the nonzero components of vectors v_k are stored on exit in parts of rows of A corresponding to the upper triangular (trapezoidal) part of A: $v_k(1:k-1) = 0, v_k(k) = 1$ and $v_k(k+1:n)$ is stored in $A(k, k+1:n)$. See magma-X.Y.Z/src/dgelqf.cpp for more details.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
#define min(a,b)  (((a)<(b))?(a):(b))
#define max(a,b)  (((a)<(b))?(b):(a))
int main( int argc, char** argv){
  magma_init();  magma_init();                // initialize Magma
  magma_timestr_t start, end;
  double  gpu_time, cpu_time;
  magma_int_t m = 4096, n = 4096, n2=m*n;
  double *a, *r;              // a, r - mxn matrices on the host
  double *tau;   // scalars defining the elementary reflectors
  double *h_work, tmp[1];  // h_work - workspace; tmp -used in
```

```
                                                  // workspace query
  magma_int_t i, info, min_mn, nb;
  magma_int_t ione = 1, lwork;       // lwork - workspace size
  magma_int_t ISEED[4] = {0,0,0,1};                    // seed
  double matnorm, work[1];  // used in difference computations
  double mzone= MAGMA_D_NEG_ONE;
  min_mn = min(m, n);
  nb = magma_get_dgeqrf_nb(m); // optimal blocksize for dgeqrf
  magma_dmalloc_cpu(&tau,min_mn);       // host memory for tau
  magma_dmalloc_pinned(&a,n2);            // host memory for a
  magma_dmalloc_pinned(&r,n2);            // host memory for r
// Get size for host workspace
  lwork = -1;
  lapackf77_dgelqf(&m, &n, a, &m, tau, tmp, &lwork, &info);
  lwork = (magma_int_t)MAGMA_D_REAL( tmp[0] );
  lwork = max( lwork, m*nb );
  magma_dmalloc_pinned(&h_work,lwork);
// Random  matrix a,  copy a -> r
  lapackf77_dlarnv( &ione, ISEED, &n2, a );
  lapackf77_dlacpy(MagmaUpperLowerStr,&m,&n,a,&m,r,&m );
// MAGMA
  start = get_current_time();
// LQ factorization for a real matrix a=L*Q using Magma
// L - lower triangular, Q - orthogonal

  magma_dgelqf(m,n,r,m,tau,h_work,lwork,&info);

  end = get_current_time();
  gpu_time =  GetTimerValue(start, end)/1e3;
  printf("MAGMA time: %7.3f sec.\n",gpu_time);  // print Magma
// LAPACK                                                time
  start = get_current_time();
// LQ factorization for a real matrix a=L*Q on the host
  lapackf77_dgelqf(&m,&n,a,&m,tau,h_work,&lwork,&info);
  end = get_current_time();
  cpu_time =  GetTimerValue(start, end)/1e3;
  printf("LAPACK time: %7.3f sec.\n",cpu_time);// print Lapack
// difference                                            time
  matnorm = lapackf77_dlange("f", &m, &n, a, &m, work);
  blasf77_daxpy(&n2, &mzone, a, &ione, r, &ione);
  printf("difference: %e\n",              // ||a-r||_F/||a||_F
  lapackf77_dlange("f", &m, &n, r, &m, work) / matnorm);
// Free emory
  free(tau);                                 // free host memory
  magma_free_pinned(a);                      // free host memory
  magma_free_pinned(r);                      // free host memory
  magma_free_pinned(h_work);                 // free host memory
  magma_finalize( );                           // finalize Magma
  return EXIT_SUCCESS;
}
// MAGMA time:  0.542  sec.
//
```

```
// LAPACK time:  3.524  sec.
//
// difference: 3.676235e-15
```

4.5.11 magma_sgelqf_gpu - LQ decomposition in single precision, GPU interface

This function computes in single precision the LQ factorization:

$$A = L\,Q,$$

where A is an $m \times n$ general matrix defined on the device, L is lower triangular (lower trapezoidal in general case) and Q is orthogonal. On exit the lower triangle (trapezoid) of A contains L. The orthogonal matrix Q is represented as a product of elementary reflectors $H(1) \ldots H(\min(m,n))$, where $H(k) = I - \tau_k v_k v_k^T$. The real scalars τ_k are stored in an array τ and the nonzero components of vectors v_k are stored on exit in parts of rows of A corresponding to the upper triangular (trapezoidal) part of A: $v_k(1:k-1) = 0, v_k(k) = 1$ and $v_k(k+1:n)$ is stored in $A(k, k+1:n)$. See magma-X.Y.Z/src/sgelqf_gpu.cpp for more details.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
#define min(a,b)  (((a)<(b))?(a):(b))
#define max(a,b)  (((a)<(b))?(b):(a))
int main( int argc, char** argv){
  magma_init();  magma_init();              // initialize Magma
  magma_timestr_t start, end;
  float  gpu_time, cpu_time;
  magma_int_t m = 4096, n = 4096, n2=m*n;
  float *a, *r;              // a, r - mxn matrices on the host
  float *h_work, tmp[1];   // h_work - workspace; tmp -used in
                                          // workspace query
  float *tau;    // scalars defining the elementary reflectors
  float *d_a;                // d_a - mxn matrix on the device
  magma_int_t i, info, min_mn, nb;
  magma_int_t ione = 1, lwork;       // lwork - workspace size
  magma_int_t ISEED[4] = {0,0,0,1};                    // seed
  float matnorm, work[1];   // used in difference computations
  float mzone= MAGMA_S_NEG_ONE;
  min_mn = min(m, n);
  nb = magma_get_sgeqrf_nb(m); // optimal blocksize for sgeqrf
  magma_smalloc_cpu(&tau,min_mn);       // host memory for tau
  magma_smalloc_pinned(&a,n2);            // host memory for a
  magma_smalloc_pinned(&r,n2);            // host memory for r
  magma_smalloc(&d_a,n2);             // device memory for d_a
// Get size for host workspace
```

```
  lwork = -1;
  lapackf77_sgelqf(&m, &n, a, &m, tau, tmp, &lwork, &info);
  lwork = (magma_int_t)MAGMA_S_REAL( tmp[0] );
  lwork = max( lwork, m*nb );
  magma_smalloc_pinned(&h_work,lwork);     // sgeqlf needs this
// Random  matrix a,  copy a -> r                          array
  lapackf77_slarnv( &ione, ISEED, &n2, a );
  lapackf77_slacpy(MagmaUpperLowerStr,&m,&n,a,&m,r,&m );
// MAGMA
  magma_ssetmatrix( m, n, r, m, d_a, m);       // copy r -> d_a
  start = get_current_time();
// LQ factorization for a real matrix d_a=L*Q on the  device
// L - lower triangular, Q - orthogonal

  magma_sgelqf_gpu(m,n,d_a,m,tau,h_work,lwork,&info);

  end = get_current_time();
  gpu_time =  GetTimerValue(start, end)/1e3;
  printf("MAGMA time: %7.3f sec.\n",gpu_time);  // print Magma
// LAPACK                                                  time
  start = get_current_time();
// LQ factorization for a real matrix a=L*Q on the host
  lapackf77_sgelqf(&m,&n,a,&m,tau,h_work,&lwork,&info);
  end = get_current_time();
  cpu_time =  GetTimerValue(start, end)/1e3;
  printf("LAPACK time: %7.3f sec.\n",cpu_time);// print Lapack
// difference                                              time
  magma_sgetmatrix( m, n, d_a, m, r, m);
  matnorm = lapackf77_slange("f", &m, &n, a, &m, work);
  blasf77_saxpy(&n2, &mzone, a, &ione, r, &ione);
  printf("difference: %e\n",              // ||a-r||_F/||a||_F
  lapackf77_slange("f", &m, &n, r, &m, work) / matnorm);
// Free emory
  free(tau);                                 // free host memory
  magma_free_pinned(a);                      // free host memory
  magma_free_pinned(r);                      // free host memory
  magma_free_pinned(h_work);                 // free host memory
  magma_free(d_a);                         // free device memory
  magma_finalize( );                           // finalize Magma
  return EXIT_SUCCESS;
}
// MAGMA time:   0.135 sec.
//
// LAPACK time:  2.008 sec.
//
// difference: 1.982540e-06
```

4.5.12 `magma_dgelqf_gpu` - LQ decomposition in double precision, GPU interface

This function computes in double precision the LQ factorization:

$$A = L\,Q,$$

where A is an $m \times n$ general matrix defined on the device, L is lower triangular (lower trapezoidal in general case) and Q is orthogonal. On exit the lower triangle (trapezoid) of A contains L. The orthogonal matrix Q is represented as a product of elementary reflectors $H(1)\ldots H(\min(m,n))$, where $H(k) = I - \tau_k v_k v_k^T$. The real scalars τ_k are stored in an array τ and the nonzero components of vectors v_k are stored on exit in parts of rows of A corresponding to the upper triangular (trapezoidal) part of A: $v_k(1:k-1) = 0, v_k(k) = 1$ and $v_k(k+1:n)$ is stored in $A(k, k+1:n)$. See `magma-X.Y.Z/src/dgelqf_gpu.cpp` for more details.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
#define min(a,b)  (((a)<(b))?(a):(b))
#define max(a,b)  (((a)<(b))?(b):(a))
int main( int argc, char** argv){
  magma_init();  magma_init();              // initialize Magma
  magma_timestr_t start, end;
  double  gpu_time, cpu_time;
  magma_int_t m = 4096, n = 4096, n2=m*n;
  double *a, *r;            // a, r - mxn matrices on the host
  double *h_work, tmp[1];  // h_work - workspace; tmp -used in
                                        // workspace query
  double *tau;   // scalars defining the elementary reflectors
  double *d_a;               // d_a - mxn matrix on the device
  magma_int_t i, info, min_mn, nb;
  magma_int_t ione = 1, lwork;       // lwork - workspace size
  magma_int_t ISEED[4] = {0,0,0,1};                    // seed
  double matnorm, work[1];  // used in difference computations
  double mzone= MAGMA_D_NEG_ONE;
  min_mn = min(m, n);
  nb = magma_get_dgeqrf_nb(m); // optimal blocksize for dgeqrf
  magma_dmalloc_cpu(&tau,min_mn);       // host memory for tau
  magma_dmalloc_pinned(&a,n2);            // host memory for a
  magma_dmalloc_pinned(&r,n2);            // host memory for r
  magma_dmalloc(&d_a,n2);             // device memory for d_a
// Get size for host workspace
  lwork = -1;
  lapackf77_dgelqf(&m, &n, a, &m, tau, tmp, &lwork, &info);
  lwork = (magma_int_t)MAGMA_D_REAL( tmp[0] );
  lwork = max( lwork, m*nb );
  magma_dmalloc_pinned(&h_work,lwork);    // dgeqlf needs this
```

```
// Random  matrix a,  copy a -> r                          array
  lapackf77_dlarnv( &ione, ISEED, &n2, a );
  lapackf77_dlacpy(MagmaUpperLowerStr,&m,&n,a,&m,r,&m );
// MAGMA
  magma_dsetmatrix( m, n, r, m, d_a, m);      // copy r -> d_a
  start = get_current_time();
// LQ factorization for a real matrix d_a=L*Q on the  device
// L - lower triangular, Q - orthogonal

  magma_dgelqf_gpu(m,n,d_a,m,tau,h_work,lwork,&info);

  end = get_current_time();
  gpu_time =  GetTimerValue(start, end)/1e3;
  printf("MAGMA time: %7.3f sec.\n",gpu_time);  // print Magma
// LAPACK                                                  time
  start = get_current_time();
// LQ factorization for a real matrix a=L*Q on the host
  lapackf77_dgelqf(&m,&n,a,&m,tau,h_work,&lwork,&info);
  end = get_current_time();
  cpu_time =  GetTimerValue(start, end)/1e3;
  printf("LAPACK time: %7.3f sec.\n",cpu_time);// print Lapack
// difference                                              time
  magma_dgetmatrix( m, n, d_a, m, r, m);
  matnorm = lapackf77_dlange("f", &m, &n, a, &m, work);
  blasf77_daxpy(&n2, &mzone, a, &ione, r, &ione);
  printf("difference: %e\n",              // ||a-r||_F/||a||_F
  lapackf77_dlange("f", &m, &n, r, &m, work) / matnorm);
// Free emory
  free(tau);                                  // free host memory
  magma_free_pinned(a);                       // free host memory
  magma_free_pinned(r);                       // free host memory
  magma_free_pinned(h_work);                  // free host memory
  magma_free(d_a);                          // free device memory
  magma_finalize( );                            // finalize Magma
  return EXIT_SUCCESS;
}
// MAGMA time:  0.447  sec.
//
// LAPACK time:  3.843  sec.
//
// difference: 3.676235e-15
```

4.5.13 magma_sgeqp3 - QR decomposition with column pivoting in single precision, CPU interface

This function computes in single precision a QR factorization with column pivoting:

$$A\,P = Q\,R,$$

where A is an $m \times n$ matrix defined on the host, R is upper triangular (trapezoidal), Q is orthogonal and P is a permutation matrix. On exit

the upper triangle (trapezoid) of A contains R. The orthogonal matrix Q is represented as a product of elementary reflectors $H(1) \dots H(\min(m,n))$, where $H(k) = I - \tau_k v_k v_k^T$. The real scalars τ_k are stored in an array τ and the nonzero components of the vectors v_k are stored on exit in parts of columns of A corresponding to its upper triangular (trapezoidal) part: $v_k(1:k-1) = 0, v_k(k) = 1$ and $v_k(k+1:m)$ is stored in $A(k+1:m,k)$. The information on columns pivoting is contained in $jptv$. On exit if $jptv(j) = k$, then j-th column of AP was the k-th column of A. See magma-X.Y.Z/src/sgeqp3.cpp for more details.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
#define min(a,b)  (((a)<(b))?(a):(b))
#define max(a,b)  (((a)<(b))?(b):(a))
int main( int argc, char** argv)
{
  magma_init();                              // initialize Magma
  magma_timestr_t  start, end;
  float  gpu_time, cpu_time;
  magma_int_t m = 8192, n = m, n2=m*n;
  float *a, *r;              // a, r - mxn matrices on the host
  float *h_work;                                    // workspace
  float *tau;    // scalars defining the elementary reflectors
  magma_int_t *jpvt;                     // pivoting information
  magma_int_t i, j, info, min_mn=min(m, n), nb;
  magma_int_t ione = 1, lwork;       // lwork - workspace size
  magma_int_t ISEED[4] = {0,0,0,1};                    // seed
  float  c_neg_one = MAGMA_S_NEG_ONE;
  nb = magma_get_sgeqp3_nb(min_mn);       // optimal blocksize
  jpvt=(magma_int_t*)malloc(n*sizeof(magma_int_t)); //host mem.
                                                    // for jpvt
  magma_smalloc_cpu(&tau,min_mn);       // host memory for tau
  magma_smalloc_pinned(&a,n2);            // host memory for a
  magma_smalloc_pinned(&r,n2);            // host memory for r
  lwork = 2*n + ( n+1 )*nb;
  lwork = max(lwork, m * n + n);
  magma_smalloc_cpu(&h_work,lwork);  // host memory for h_work
// Random  matrix a,  copy a -> r
  lapackf77_slarnv(&ione,ISEED,&n2,a);
  lapackf77_slacpy(MagmaUpperLowerStr,&m,&n,a,&m,r,&m);// a->r
// LAPACK
  for (j = 0; j < n; j++)
    jpvt[j] = 0;
  start = get_current_time();
// QR decomposition with column pivoting, Lapack version
  lapackf77_sgeqp3(&m,&n,r,&m,jpvt,tau,h_work,&lwork,&info);
  end = get_current_time();
  cpu_time = GetTimerValue(start, end) * 1e-3;
  printf("LAPACK time: %7.3f sec.\n",cpu_time); // Lapack time
```

```
// MAGMA
  lapackf77_slacpy(MagmaUpperLowerStr,&m,&n,a,&m,r,&m);
  for (j = 0; j < n; j++)
    jpvt[j] = 0 ;
  start = get_current_time();
// QR decomposition with column pivoting, Magma version

  magma_sgeqp3(m,n,r,m,jpvt,tau,h_work,lwork,&info);

  end = get_current_time();
  gpu_time = GetTimerValue(start, end) * 1e-3;
  printf("MAGMA time: %7.3f sec.\n",gpu_time);   // Magma time
  float result[3], ulp;
//slamch determines single precision machine parameters
  ulp = lapackf77_slamch( "P" );
// Compute norm( A*P - Q*R )
  result[0] = lapackf77_sqpt01(&m, &n, &min_mn, a, r, &m,
                               tau, jpvt, h_work, &lwork);
  result[0] *= ulp;
  printf("error %e\n",result[0]);
// Free memory
  free(jpvt);                                  // free host memory
  free(tau);                                   // free host memory
  magma_free_pinned(a);                        // free host memory
  magma_free_pinned(r);                        // free host memory
  free( h_work );                              // free host memory
  magma_finalize( );                             // finalize Magma
  return EXIT_SUCCESS;
}
// LAPACK time:  31.383  sec.
//
// MAGMA time:   14.461 sec.
//
// error 4.985993e-10
```

4.5.14 magma_dgeqp3 - QR decomposition with column pivoting in double precision, CPU interface

This function computes in double precision a QR factorization with column pivoting:

$$A\,P = Q\,R,$$

where A is an $m \times n$ matrix defined on the host, R is upper triangular (trapezoidal), Q is orthogonal and P is a permutation matrix. On exit the upper triangle (trapezoid) of A contains R. The orthogonal matrix Q is represented as a product of elementary reflectors $H(1) \ldots H(\min(m,n))$, where $H(k) = I - \tau_k v_k v_k^T$. The real scalars τ_k are stored in an array τ and the nonzero components of the vectors v_k are stored on exit in parts of columns of A corresponding to its upper triangular (trapezoidal) part: $v_k(1:k-1) = 0, v_k(k) = 1$ and $v_k(k+1:m)$ is stored in $A(k+1:m,k)$.

The information on columns pivoting is contained in $jptv$. On exit if $jptv(j) = k$, then j-th column of AP was the k-th column of A. See magma-X.Y.Z/src/dgeqp3.cpp for more details.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
#define min(a,b)  (((a)<(b))?(a):(b))
#define max(a,b)  (((a)<(b))?(b):(a))
int main( int argc, char** argv)
{
  magma_init();                              // initialize Magma
  magma_timestr_t  start, end;
  double  gpu_time, cpu_time;
  magma_int_t m = 8192, n = m, n2=m*n;
  double *a, *r;            // a, r - mxn matrices on the host
  double *h_work;                                  // workspace
  double *tau;   // scalars defining the elementary reflectors
  magma_int_t *jpvt;                    // pivoting information
  magma_int_t i, j, info, min_mn=min(m, n), nb;
  magma_int_t ione = 1, lwork;       // lwork - workspace size
  magma_int_t ISEED[4] = {0,0,0,1};                    // seed
  double  c_neg_one = MAGMA_D_NEG_ONE;
  nb = magma_get_dgeqp3_nb(min_mn);       // optimal blocksize
  jpvt=(magma_int_t*)malloc(n*sizeof(magma_int_t)); //host mem.
                                                   // for jpvt
  magma_dmalloc_cpu(&tau,min_mn);       // host memory for tau
  magma_dmalloc_pinned(&a,n2);            // host memory for a
  magma_dmalloc_pinned(&r,n2);            // host memory for r
  lwork = 2*n + ( n+1 )*nb;
  lwork = max(lwork, m * n + n);
  magma_dmalloc_cpu(&h_work,lwork);  // host memory for h_work
// Random  matrix a,  copy a -> r
  lapackf77_dlarnv(&ione,ISEED,&n2,a);
  lapackf77_dlacpy(MagmaUpperLowerStr,&m,&n,a,&m,r,&m);// a->r
// LAPACK
  for (j = 0; j < n; j++)
    jpvt[j] = 0;
  start = get_current_time();
// QR decomposition with column pivoting, Lapack version
  lapackf77_dgeqp3(&m,&n,r,&m,jpvt,tau,h_work,&lwork,&info);
  end = get_current_time();
  cpu_time = GetTimerValue(start, end) * 1e-3;
  printf("LAPACK time: %7.3f sec.\n",cpu_time); // Lapack time
// MAGMA
  lapackf77_dlacpy(MagmaUpperLowerStr,&m,&n,a,&m,r,&m);
  for (j = 0; j < n; j++)
    jpvt[j] = 0 ;
  start = get_current_time();
// QR decomposition with column pivoting, Magma version
```

```
  magma_dgeqp3(m,n,r,m,jpvt,tau,h_work,lwork,&info);

  end = get_current_time();
  gpu_time = GetTimerValue(start, end) * 1e-3;
  printf("MAGMA time: %7.3f sec.\n",gpu_time);   // Magma time
  double result[3], ulp;
//dlamch determines double precision machine parameters
  ulp = lapackf77_dlamch( "P" );
// Compute norm( A*P - Q*R )
  result[0] = lapackf77_dqpt01(&m, &n, &min_mn, a, r, &m,
                               tau, jpvt, h_work, &lwork);
  result[0] *= ulp;
  printf("error %e\n",result[0]);
// Free memory
  free(jpvt);                                  // free host memory
  free(tau);                                   // free host memory
  magma_free_pinned(a);                        // free host memory
  magma_free_pinned(r);                        // free host memory
  free( h_work );                              // free host memory
  magma_finalize( );                             // finalize Magma
  return EXIT_SUCCESS;
}
// LAPACK time:  57.512  sec.
//
// MAGMA time:  16.946  sec.
//
// error 1.791955e-18
```

4.6 Eigenvalues and eigenvectors for general matrices

4.6.1 magma_sgeev - compute the eigenvalues and optionally eigenvectors of a general real matrix in single precision, CPU interface, small matrix

This function computes in single precision the eigenvalues and, optionally, the left and/or right eigenvectors for an $n \times n$ matrix A defined on the host. The first parameter can take the values `MagmaNoVec,'N'` or `MagmaVec,'V'` and answers the question whether the left eigenvectors are to be computed. Similarly the second parameter answers the question whether the right eigenvectors are to be computed. The computed eigenvectors are normalized to have Euclidean norm equal to one. If computed, the left eigenvectors are stored in columns of an array `VL` and the right eigenvectors in columns of `VR`. The real and imaginary parts of eigenvalues are stored in arrays `wr`, `wi` respectively. See magma-X.Y.Z/src/sgeev.cpp for more details.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
#define max(a,b)  (((a)<(b))?(b):(a))
int main( int argc, char** argv) {
  magma_init();                                  // initialize Magma
  magma_int_t n=1024, n2=n*n;
  float  *a, *r;              // a, r - nxn matrices on the host
  float *VL, *VR;            // VL,VR - nxn matrices of left and
                                                 // right eigenvectors
  float *wr1, *wr2;      // wr1,wr2 - real parts of eigenvalues
  float *wi1, *wi2;             // wi1,wi2 - imaginary parts of
  magma_int_t ione = 1, i, j, info, nb;          // eigenvalues
  float mione = -1.0f, error, *h_work;    // h_work - workspace
  magma_int_t incr = 1, inci = 1, lwork;// lwork -worksp. size
  nb = magma_get_sgehrd_nb(n); // optimal blocksize for sgehrd
  float work[1];              // used in difference computations
  lwork = n*(2+nb);
  lwork = max(lwork,n*(5+2*n));
  magma_smalloc_cpu(&wr1,n);             // host memory for real
  magma_smalloc_cpu(&wr2,n);              // and imaginary parts
  magma_smalloc_cpu(&wi1,n);                   // of eigenvalues
  magma_smalloc_cpu(&wi2,n);
  magma_smalloc_cpu(&a,n2);                 // host memory for a
  magma_smalloc_cpu(&r,n2);                 // host memory for r
  magma_smalloc_cpu(&VL,n2);              // host memory for left
  magma_smalloc_cpu(&VR,n2);            // and right eigenvectors
  magma_smalloc_cpu(&h_work,lwork);  // host memory for h_work
// define a, r                              //      [1 0 0 0 0 ...]
  for(i=0;i<n;i++){                        //      [0 2 0 0 0 ...]
    a[i*n+i]=(float)(i+1);                 // a = [0 0 3 0 0 ...]
    r[i*n+i]=(float)(i+1);                 //      [0 0 0 4 0 ...]
  }                                        //      [0 0 0 0 5 ...]
  printf("upper left corner of a:\n");  //       ............
  magma_sprint(5,5,a,n);  // print a
// compute the eigenvalues  and the right eigenvectors
// for a general, real  nxn matrix a,
// Magma version, left eigenvectors not computed,
// right eigenvectors are computed

  magma_sgeev('N','V',n,r,n,wr1,wi1,VL,n,
                                   VR,n,h_work,lwork,&info);

  printf("first 5 eigenvalues of a:\n");
  for(j=0;j<5;j++)
    printf("%f+%f*I\n",wr1[j],wi1[j]);    // print eigenvalues
  printf("left upper corner of right eigenvectors matrix:\n");
  magma_sprint(5,5,VR,n);           // print right eigenvectors
// Lapack version                                  // in columns
  lapackf77_sgeev("N","V",&n,a,&n,wr2,wi2,VL,&n,VR,&n,
                                   h_work,&lwork,&info);
```

```
// difference in real parts of eigenvalues
  blasf77_saxpy( &n, &mione, wr1, &incr, wr2, &incr);
  error = lapackf77_slange( "M", &n, &ione, wr2, &n, work );
  printf("difference in real parts: %e\n",error);
// difference in imaginary parts of eigenvalues
  blasf77_saxpy( &n, &mione, wi1, &inci, wi2, &inci);
  error = lapackf77_slange( "M", &n, &ione, wi2, &n, work );
  printf("difference in imaginary parts: %e\n",error);
  free(wr1);                                  // free host memory
  free(wr2);                                  // free host memory
  free(wi1);                                  // free host memory
  free(wi2);                                  // free host memory
  free(a);                                    // free host memory
  free(r);                                    // free host memory
  free(VL);                                   // free host memory
  free(VR);                                   // free host memory
  free(h_work);                               // free host memory
  magma_finalize();                             // finalize Magma
  return EXIT_SUCCESS;
}
// upper left corner of a:
//[
//   1.0000   0.       0.       0.       0.
//   0.       2.0000   0.       0.       0.
//   0.       0.       3.0000   0.       0.
//   0.       0.       0.       4.0000   0.
//   0.       0.       0.       0.       5.0000
//];
// first 5 eigenvalues of a:
// 1.000000+0.000000*I
// 2.000000+0.000000*I
// 3.000000+0.000000*I
// 4.000000+0.000000*I
// 5.000000+0.000000*I
// left upper corner of right eigenvectors matrix:
//[
//   1.0000   0.       0.       0.       0.
//   0.       1.0000   0.       0.       0.
//   0.       0.       1.0000   0.       0.
//   0.       0.       0.       1.0000   0.
//   0.       0.       0.       0.       1.0000
//];
// difference in real parts: 0.000000e+00
// difference in imaginary parts: 0.000000e+00
```

4.6.2 `magma_dgeev` - compute the eigenvalues and optionally eigenvectors of a general real matrix in double precision, CPU interface, small matrix

This function computes in double precision the eigenvalues and, optionally, the left and/or right eigenvectors for an $n \times n$ matrix A defined on the host. The first parameter can take the values `MagmaNoVec,'N'` or `MagmaVec,'V'` and answers the question whether the left eigenvectors are to be computed. Similarly the second parameter answers the question whether the right eigenvectors are to be computed. The computed eigenvectors are normalized to have Euclidean norm equal to one. If computed, the left eigenvectors are stored in columns of an array `VL` and the right eigenvectors in columns of `VR`. The real and imaginary parts of eigenvalues are stored in arrays `wr, wi` respectively. See magma-X.Y.Z/src/dgeev.cpp for more details.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
#define max(a,b)  (((a)<(b))?(b):(a))
int main( int argc, char** argv) {
  magma_init();                                  // initialize Magma
  magma_int_t n=1024, n2=n*n;
  double  *a, *r;            // a, r - nxn matrices on the host
  double *VL, *VR;          // VL,VR - nxn matrices of left and
                                              // right eigenvectors
  double *wr1, *wr2;     // wr1,wr2 - real parts of eigenvalues
  double *wi1, *wi2;            // wi1,wi2 - imaginary parts of
  magma_int_t ione = 1, i, j, info, nb;          // eigenvalues
  double mione = -1.0 , error, *h_work;  // h_work - workspace
  magma_int_t incr = 1, inci = 1, lwork;// lwork -worksp. size
  nb = magma_get_dgehrd_nb(n); // optimal blocksize for dgehrd
  double work[1];            // used in difference computations
  lwork = n*(2+nb);
  lwork = max(lwork,n*(5+2*n));
  magma_dmalloc_cpu(&wr1,n);             // host memory for real
  magma_dmalloc_cpu(&wr2,n);              // and imaginary parts
  magma_dmalloc_cpu(&wi1,n);                   // of eigenvalues
  magma_dmalloc_cpu(&wi2,n);
  magma_dmalloc_cpu(&a,n2);                 // host memory for a
  magma_dmalloc_cpu(&r,n2);                 // host memory for r
  magma_dmalloc_cpu(&VL,n2);              // host memory for left
  magma_dmalloc_cpu(&VR,n2);            // and right eigenvectors
  magma_dmalloc_cpu(&h_work,lwork);  // host memory for h_work
// define a, r                                //      [1 0 0 0 0 ...]
  for(i=0;i<n;i++){                          //      [0 2 0 0 0 ...]
    a[i*n+i]=(double)(i+1);                  // a =  [0 0 3 0 0 ...]
    r[i*n+i]=(double)(i+1);                  //      [0 0 0 4 0 ...]
  }                                          //      [0 0 0 0 5 ...]
```

```
  printf("upper left corner of a:\n");  //      ............
  magma_dprint(5,5,a,n);  // print a
// compute the eigenvalues  and the right eigenvectors
// for a general, real  nxn matrix a,
// Magma version, left eigenvectors not computed,
// right eigenvectors are computed

  magma_dgeev('N','V',n,r,n,wr1,wi1,VL,n,
                                  VR,n,h_work,lwork,&info);

  printf("first 5 eigenvalues of a:\n");
  for(j=0;j<5;j++)
    printf("%f+%f*I\n",wr1[j],wi1[j]);    // print eigenvalues
  printf("left upper corner of right eigenvectors matrix:\n");
  magma_dprint(5,5,VR,n);          // print right eigenvectors
// Lapack version                                  // in columns
  lapackf77_dgeev("N","V",&n,a,&n,wr2,wi2,VL,&n,VR,&n,
                                        h_work,&lwork,&info);
// difference in real parts of eigenvalues
  blasf77_daxpy( &n, &mione, wr1, &incr, wr2, &incr);
  error = lapackf77_dlange( "M", &n, &ione, wr2, &n, work );
  printf("difference in real parts: %e\n",error);
// difference in imaginary parts of eigenvalues
  blasf77_daxpy( &n, &mione, wi1, &inci, wi2, &inci);
  error = lapackf77_dlange( "M", &n, &ione, wi2, &n, work );
  printf("difference in imaginary parts: %e\n",error);
  free(wr1);                                  // free host memory
  free(wr2);                                  // free host memory
  free(wi1);                                  // free host memory
  free(wi2);                                  // free host memory
  free(a);                                    // free host memory
  free(r);                                    // free host memory
  free(VL);                                   // free host memory
  free(VR);                                   // free host memory
  free(h_work);                               // free host memory
  magma_finalize();                             // finalize Magma
  return EXIT_SUCCESS;
}
// upper left corner of a:
//[
//   1.0000   0.       0.       0.       0.
//   0.       2.0000   0.       0.       0.
//   0.       0.       3.0000   0.       0.
//   0.       0.       0.       4.0000   0.
//   0.       0.       0.       0.       5.0000
//];
// first 5 eigenvalues of a:
// 1.000000+0.000000*I
// 2.000000+0.000000*I
// 3.000000+0.000000*I
// 4.000000+0.000000*I
// 5.000000+0.000000*I
```

```
// left upper corner of right eigenvectors matrix:
//[
//   1.0000   0.       0.       0.       0.
//   0.       1.0000   0.       0.       0.
//   0.       0.       1.0000   0.       0.
//   0.       0.       0.       1.0000   0.
//   0.       0.       0.       0.       1.0000
//];
//difference in real parts: 0.000000e+00
//difference in imaginary parts: 0.000000e+00
```

4.6.3 magma_sgeev - compute the eigenvalues and optionally eigenvectors of a general real matrix in single precision, CPU interface, big matrix

This function computes in single precision the eigenvalues and, optionally, the left and/or right eigenvectors for an $n \times n$ matrix A defined on the host. The first parameter can take the values `MagmaNoVec,'N'` or `MagmaVec,'V'` and answers the question whether the left eigenvectors are to be computed. Similarly the second parameter answers the question whether the right eigenvectors are to be computed. The computed eigenvectors are normalized to have Euclidean norm equal to one. If computed, the left eigenvectors are stored in columns of an array `VL` and the right eigenvectors in columns of `VR`. The real and imaginary parts of eigenvalues are stored in arrays `wr, wi` respectively. See magma-X.Y.Z/src/sgeev.cpp for more details.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
#define max(a,b)  (((a)<(b))?(b):(a))
int main( int argc, char** argv) {
  magma_init();                              // initialize Magma
  magma_timestr_t  start, end;
  magma_int_t n=8192, n2=n*n;
  float *a, *r;              // a, r - nxn matrices on the host
  float *VL, *VR;           // VL,VR - nxn matrices of left and
                                          // right eigenvectors
  float *wr1, *wr2;     // wr1,wr2 - real parts of eigenvalues
  float *wi1, *wi2;// wi1,wi2 - imaginary parts of eigenvalues
  float  gpu_time, cpu_time, *h_work;    // h_work - workspace
  magma_int_t ione=1,i,j,info,nb,lwork;// lwork - worksp. size
  magma_int_t ISEED[4] = {0,0,0,1};                    // seed
  nb = magma_get_sgehrd_nb(n); // optimal blocksize for sgehrd
  lwork = n*(2+nb);
  lwork = max(lwork, n*(5+2*n));
  magma_smalloc_cpu(&wr1,n);           // host memory for real
  magma_smalloc_cpu(&wr2,n);            // and imaginary parts
  magma_smalloc_cpu(&wi1,n);                 // of eigenvalues
```

```
  magma_smalloc_cpu(&wi2,n);
  magma_smalloc_cpu(&a,n2);                  // host memory for a
  magma_smalloc_pinned(&r,n2);               // host memory for r
  magma_smalloc_pinned(&VL,n2);            // host memory for left
  magma_smalloc_pinned(&VR,n2);          // and right eigenvectors
  magma_smalloc_pinned(&h_work,lwork);//host memory for h_work
// Random  matrix a,  copy a -> r
  lapackf77_slarnv(&ione,ISEED,&n2,a);
  lapackf77_slacpy(MagmaUpperLowerStr,&n,&n,a,&n,r,&n);
// MAGMA
  start = get_current_time();
// compute the eigenvalues of a general, real  nxn matrix a,
// Magma version, left and right eigenvectors not computed

  magma_sgeev('N','N',n,r,n, wr1,wi1,
                             VL,n,VR,n,h_work,lwork,&info);

  end = get_current_time();
  gpu_time = GetTimerValue(start,end) / 1e3;
  printf("sgeev gpu time: %7.5f sec.\n",gpu_time);     // Magma
// LAPACK                                                 // time
  start = get_current_time();
// compute the eigenvalues of a general, real  nxn matrix a,
// Lapack version
  lapackf77_sgeev("N", "N", &n, a, &n,
        wr2, wi2,  VL, &n, VR, &n, h_work, &lwork, &info);
  end = get_current_time();
  cpu_time = GetTimerValue(start,end) / 1e3;
  printf("sgeev cpu time: %7.5f sec.\n",cpu_time);   // Lapack
  free(wr1);                                           // time
  free(wr2);                                  // free host memory
  free(wi1);                                  // free host memory
  free(wi2);                                  // free host memory
  free(a);                                    // free host memory
  magma_free_pinned(r);                       // free host memory
  magma_free_pinned(VL);                      // free host memory
  magma_free_pinned(VR);                      // free host memory
  magma_free_pinned(h_work);                  // free host memory
  magma_finalize( );                            // finalize Magma
  return EXIT_SUCCESS;
}
// sgeev GPU time: 43.44775 sec.
// sgeev CPU time: 100.97041 sec.
```

4.6.4 magma_dgeev - compute the eigenvalues and optionally eigenvectors of a general real matrix in double precision, CPU interface, big matrix

This function computes in double precision the eigenvalues and, optionally, the left and/or right eigenvectors for an $n \times n$ matrix A defined on

the host. The first parameter can take the values `MagmaNoVec,'N'` or `MagmaVec,'V'` and answers the question whether the left eigenvectors are to be computed. Similarly the second parameter answers the question whether the right eigenvectors are to be computed. The computed eigenvectors are normalized to have Euclidean norm equal to one. If computed, the left eigenvectors are stored in columns of an array `VL` and the right eigenvectors in columns of `VR`. The real and imaginary parts of eigenvalues are stored in arrays `wr, wi` respectively. See magma-X.Y.Z/src/dgeev.cpp for more details.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
#define max(a,b)  (((a)<(b))?(b):(a))
int main( int argc, char** argv) {
  magma_init();                              // initialize Magma
  magma_timestr_t  start, end;
  magma_int_t n=8192, n2=n*n;
  double *a, *r;             // a, r - nxn matrices on the host
  double *VL, *VR;          // VL,VR - nxn matrices of left and
                                          // right eigenvectors
  double *wr1, *wr2;     // wr1,wr2 - real parts of eigenvalues
  double *wi1, *wi2;// wi1,wi2 -imaginary parts of eigenvalues
  double  gpu_time, cpu_time, *h_work;   // h_work - workspace
  magma_int_t ione=1,i,j,info,nb,lwork;// lwork - worksp. size
  magma_int_t ISEED[4] = {0,0,0,1};                    // seed
  nb = magma_get_dgehrd_nb(n); // optimal blocksize for dgehrd
  lwork = n*(2+nb);
  lwork = max(lwork, n*(5+2*n));
  magma_dmalloc_cpu(&wr1,n);           // host memory for real
  magma_dmalloc_cpu(&wr2,n);            // and imaginary parts
  magma_dmalloc_cpu(&wi1,n);                 // of eigenvalues
  magma_dmalloc_cpu(&wi2,n);
  magma_dmalloc_cpu(&a,n2);                // host memory for a
  magma_dmalloc_pinned(&r,n2);             // host memory for r
  magma_dmalloc_pinned(&VL,n2);         // host memory for left
  magma_dmalloc_pinned(&VR,n2);       // and right eigenvectors
  magma_dmalloc_pinned(&h_work,lwork);//host memory for h_work
// Random  matrix a,  copy a -> r
  lapackf77_dlarnv(&ione,ISEED,&n2,a);
  lapackf77_dlacpy(MagmaUpperLowerStr,&n,&n,a,&n,r,&n);
// MAGMA
  start = get_current_time();
// compute the eigenvalues of a general, real  nxn matrix a,
// Magma version, left and right eigenvectors not computed

  magma_dgeev('N','N',n,r,lda, wr1,wi1,
                            VL,n,VR,n,h_work,lwork,&info);

  end = get_current_time();
```

```
  gpu_time = GetTimerValue(start,end) / 1e3;
  printf("dgeev gpu time: %7.5f sec.\n",gpu_time);     // Magma
// LAPACK                                               // time
  start = get_current_time();
// compute the eigenvalues of a general, real  nxn matrix a,
// Lapack version
  lapackf77_dgeev("N", "N", &n, a, &n,
        wr2, wi2,  VL, &n, VR, &n, h_work, &lwork, &info);
  end = get_current_time();
  cpu_time = GetTimerValue(start,end) / 1e3;
  printf("dgeev cpu time: %7.5f sec.\n",cpu_time);   // Lapack
  free(wr1);                                           // time
  free(wr2);                                  // free host memory
  free(wi1);                                  // free host memory
  free(wi2);                                  // free host memory
  free(a);                                    // free host memory
  magma_free_pinned(r);                       // free host memory
  magma_free_pinned(VL);                      // free host memory
  magma_free_pinned(VR);                      // free host memory
  magma_free_pinned(h_work);                  // free host memory
  magma_finalize( );                            // finalize Magma
  return EXIT_SUCCESS;
}
// dgeev gpu time:  91.21487 sec.
// dgeev cpu time: 212.40578 sec.
```

4.6.5 magma_sgehrd - reduce a general matrix to the upper Hessenberg form in single precision, CPU interface

This function using the single precision reduces a general real $n \times n$ matrix A defined on the host to upper Hessenberg form:

$$Q^T \, A \, Q = H,$$

where Q is an orthogonal matrix and H has zero elements below the first subdiagonal. The orthogonal matrix Q is represented as a product of elementary reflectors $H(ilo) \ldots H(ihi)$, where $H(k) = I - \tau_k v_k v_k^T$. The real scalars τ_k are stored in an array τ and the information on vectors v_k is stored on exit in the lower triangular part of A below the first subdiagonal: $v_k(1:k) = 0, v_k(k+1) = 1$ and $v_k(ihi+1:n) = 0$; $v_k(k+2:ihi)$ is stored in $A(k+2:ihi, k)$. The function uses also an array dT defined on the device, storing blocks of triangular matrices used in the reduction process. See magma-X.Y.Z/src/sgehrd.cpp for more details.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
int main( int argc, char** argv)
```

```
{
  magma_init();                              // initialize Magma
  magma_timestr_t start, end;
  float  gpu_time, cpu_time;
  magma_int_t n=4096, n2=n*n;
  float *a, *r, *r1;      // a,r,r1 - nxn matrices on the host
  float *tau;    // scalars defining the elementary reflectors
  float *h_work;                                    // workspace
  magma_int_t i, info;
  magma_int_t ione = 1, nb, lwork;   // lwork - workspace size
  float *dT; // store nb*nb blocks of triangular matrices used
  magma_int_t ilo=ione, ihi=n;                  // in reduction
  float mone= MAGMA_S_NEG_ONE;
  magma_int_t ISEED[4] = {0,0,0,1};                     // seed
  float  work[1];           // used in difference computations
  nb = magma_get_sgehrd_nb(n);// optimal block size for sgehrd
  lwork = n*nb;
  magma_smalloc_cpu(&a,n2);               // host memory for a
  magma_smalloc_cpu(&tau,n);             // host memory for tau
  magma_smalloc_pinned(&r,n2);            // host memory for r
  magma_smalloc_pinned(&r1,n2);          // host memory for r1
  magma_smalloc_pinned(&h_work,lwork);//host memory for h_work
  magma_smalloc(&dT,nb*n);             // device memory for dT
// Random  matrix a,  copy a -> r, a -> r1
  lapackf77_slarnv( &ione, ISEED, &n2, a );
  lapackf77_slacpy(MagmaUpperLowerStr,&n,&n,a,&n,r,&n);
  lapackf77_slacpy(MagmaUpperLowerStr,&n,&n,a,&n,r1,&n);
// MAGMA
  start = get_current_time();
// reduce the matrix r to upper Hessenberg form by an
// orthogonal transformation, Magma version

  magma_sgehrd(n,ilo,ihi,r,n,tau,h_work,lwork,dT,&info);

  end = get_current_time();
  gpu_time =  GetTimerValue(start,end)/1e3;
  printf("Magma time: %7.3f sec.\n",gpu_time);   // Magma time
  {
    int i, j;
    for(j=0; j<n-1; j++)
      for(i=j+2; i<n; i++)
        r[i+j*n] = MAGMA_S_ZERO;
  }
  printf("upper left corner of the Hessenberg form:\n");
  magma_sprint(5,5,r,n);          // print the Hessenberg form
// LAPACK
  start = get_current_time();
// reduce the matrix r1 to upper Hessenberg form by an
// orthogonal transformation, Lapack version
  lapackf77_sgehrd(&n,&ione,&n,r1,&n,tau,h_work,&lwork,&info);
  end = get_current_time();
  cpu_time = GetTimerValue(start,end)/1e3;      // Lapack time
```

```
  printf("Lapack time: %7.3f sec.\n",cpu_time);
  {
    int i, j;
      for(j=0; j<n-1; j++)
        for(i=j+2; i<n; i++)
          r1[i+j*n] = MAGMA_S_ZERO;
  }
// difference
  blasf77_saxpy(&n2,&mone,r,&ione,r1,&ione);
  printf("max difference: %e\n",
              lapackf77_slange("M", &n, &n, r1, &n, work));
  free(a);                                    // free host memory
  free(tau);                                  // free host memory
  magma_free_pinned(h_work);                  // free host memory
  magma_free_pinned(r);                       // free host memory
  magma_free_pinned(r1);                      // free host memory
  magma_free(dT);                             // free host memory
  magma_finalize( );                            // finalize Magma
  return EXIT_SUCCESS;
}
// Magma time:  1.702  sec.
// upper left corner of the Hessenberg form:
//[
//   0.1206 -27.7263 -16.3929  -0.3493  -0.3279
// -36.9378 1527.1729 890.8776   9.0395   0.4183
//   0.     891.8640 520.4537   5.4098   0.0378
//   0.       0.      21.1049   0.3039   0.5484
//   0.       0.       0.      18.3435  -0.3502
//];
// Lapack time:  9.272  sec.
// max difference: 1.500323e-03
```

4.6.6 `magma_dgehrd` - reduce a general matrix to the upper Hessenberg form in double precision, CPU interface

This function using the double precision reduces a general real $n \times n$ matrix A defined on the host to upper Hessenberg form:

$$Q^T \, A \, Q = H,$$

where Q is an orthogonal matrix and H has zero elements below the first subdiagonal. The orthogonal matrix Q is represented as a product of elementary reflectors $H(ilo) \ldots H(ihi)$, where $H(k) = I - \tau_k v_k v_k^T$. The real scalars τ_k are stored in an array τ and the information on vectors v_k is stored on exit in the lower triangular part of A below the first subdiagonal: $v_k(1:k) = 0, v_k(k+1) = 1$ and $v_k(ihi+1:n) = 0$; $v_k(k+2:ihi)$ is stored in $A(k+2:ihi, k)$. The function uses also an array dT defined on the device, storing blocks of triangular matrices used in the reduction process. See `magma-X.Y.Z/src/dgehrd.cpp` for more details.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
int main( int argc, char** argv)
{
  magma_init();                              // initialize Magma
  magma_timestr_t start, end;
  double  gpu_time, cpu_time;
  magma_int_t n=4096, n2=n*n;
  double *a, *r, *r1;     // a,r,r1 - nxn matrices on the host
  double *tau;   // scalars defining the elementary reflectors
  double *h_work;                                  // workspace
  magma_int_t i, info;
  magma_int_t ione = 1, nb, lwork;   // lwork - workspace size
  double *dT;// store nb*nb blocks of triangular matrices used
  magma_int_t ilo=ione, ihi=n;                  // in reduction
  double mone= MAGMA_D_NEG_ONE;
  magma_int_t ISEED[4] = {0,0,0,1};                    // seed
  double  work[1];          // used in difference computations
  nb = magma_get_dgehrd_nb(n);// optimal block size for dgehrd
  lwork = n*nb;
  magma_dmalloc_cpu(&a,n2);               // host memory for a
  magma_dmalloc_cpu(&tau,n);             // host memory for tau
  magma_dmalloc_pinned(&r,n2);            // host memory for r
  magma_dmalloc_pinned(&r1,n2);          // host memory for r1
  magma_dmalloc_pinned(&h_work,lwork);//host memory for h_work
  magma_dmalloc(&dT,nb*n);              // device memory for dT
// Random  matrix a,  copy a -> r, a -> r1
  lapackf77_dlarnv( &ione, ISEED, &n2, a );
  lapackf77_dlacpy(MagmaUpperLowerStr,&n,&n,a,&n,r,&n);
  lapackf77_dlacpy(MagmaUpperLowerStr,&n,&n,a,&n,r1,&n);
// MAGMA
  start = get_current_time();
// reduce the matrix r to upper Hessenberg form by an
// orthogonal transformation, Magma version

  magma_dgehrd(n,ilo,ihi,r,n,tau,h_work,lwork,dT,&info);

  end = get_current_time();
  gpu_time =  GetTimerValue(start,end)/1e3;
  printf("Magma time: %7.3f sec.\n",gpu_time);   // Magma time
  {
    int i, j;
    for(j=0; j<n-1; j++)
      for(i=j+2; i<n; i++)
        r[i+j*n] = MAGMA_D_ZERO;
  }
  printf("upper left corner of the Hessenberg form:\n");
  magma_dprint(5,5,r,n);          // print the Hessenberg form
// LAPACK
  start = get_current_time();
```

```
// reduce the matrix r1 to upper Hessenberg form by an
// orthogonal transformation, Lapack version
  lapackf77_dgehrd(&n,&ione,&n,r1,&n,tau,h_work,&lwork,&info);
  end = get_current_time();
  cpu_time = GetTimerValue(start,end)/1e3;      // Lapack time
  printf("Lapack time: %7.3f sec.\n",cpu_time);
  {
    int i, j;
      for(j=0; j<n-1; j++)
        for(i=j+2; i<n; i++)
          r1[i+j*n] = MAGMA_D_ZERO;
  }
// difference
  blasf77_daxpy(&n2,&mone,r,&ione,r1,&ione);
  printf("max difference: %e\n",
              lapackf77_dlange("M", &n, &n, r1, &n, work));
  free(a);                                    // free host memory
  free(tau);                                  // free host memory
  magma_free_pinned(h_work);                  // free host memory
  magma_free_pinned(r);                       // free host memory
  magma_free_pinned(r1);                      // free host memory
  magma_free(dT);                             // free host memory
  magma_finalize( );                            // finalize Magma
  return EXIT_SUCCESS;
}
// Magma time: 2.493   sec.
// upper left corner of the Hessenberg form:
//[
//   0.1206 -27.7263 -16.3929  -0.3493  -0.3279
// -36.9379 1527.1693 890.8763  9.0395   0.4182
//   0.     891.8629 520.4536   5.4098   0.0378
//   0.       0.      21.1049   0.3039   0.5484
//   0.       0.       0.      18.3435  -0.3502
//];
// Lapack time:  18.462  sec.
// max difference: 1.858180e-12
```

4.7 Eigenvalues and eigenvectors for symmetric matrices

4.7.1 `magma_ssyevd` - compute the eigenvalues and optionally eigenvectors of a symmetric real matrix in single precision, CPU interface, small matrix

This function computes in single precision all eigenvalues and, optionally, eigenvectors of a real symmetric matrix A defined on the host. The first parameter can take the values `MagmaVec,'V'` or `MagmaNoVec,'N'` and answers the question whether the eigenvectors are desired. If the eigenvectors are desired, it uses a divide and conquer algorithm. The symmetric matrix

A can be stored in lower (`MagmaLower,'L'`) or upper (`MagmaUpper,'U'`) mode. If the eigenvectors are desired, then on exit A contains orthonormal eigenvectors. The eigenvalues are stored in an array `w`. See magma-X.Y.Z/src/ssyevd.cpp for more details.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
int main( int argc, char** argv) {
  magma_init();                                  // initialize Magma
  magma_int_t n=1024, n2=n*n;
  float  *a, *r;              // a, r - nxn matrices on the host
  float  *h_work;                                  // workspace
  magma_int_t lwork;                              // h_work size
  magma_int_t  *iwork;                             // workspace
  magma_int_t  liwork;                            // iwork size
  float *w1, *w2;             // w1,w2 - vectors of  eigenvalues
  float error, work[1];       // used in difference computations
  magma_int_t ione = 1, i, j, info;
  float mione = -1.0f;
  magma_int_t incr = 1, inci = 1;
  magma_smalloc_cpu(&w1,n);              // host memory for real
  magma_smalloc_cpu(&w2,n);                        // eigenvalues
  magma_smalloc_cpu(&a,n2);                  // host memory for a
  magma_smalloc_cpu(&r,n2);                  // host memory for r
// Query for workspace sizes
  float aux_work[1];
  magma_int_t aux_iwork[1];
  magma_ssyevd('V','L',n,r,n,w1,aux_work,-1,
                               aux_iwork,-1,&info );
  lwork  = (magma_int_t) aux_work[0];
  liwork = aux_iwork[0];
  iwork=(magma_int_t*)malloc(liwork*sizeof(magma_int_t));
  magma_smalloc_cpu(&h_work,lwork); // host mem. for workspace
// define a, r                                //      [1 0 0 0 0 ...]
  for(i=0;i<n;i++){                          //      [0 2 0 0 0 ...]
    a[i*n+i]=(float)(i+1);                   // a = [0 0 3 0 0 ...]
    r[i*n+i]=(float)(i+1);                   //      [0 0 0 4 0 ...]
  }                                          //      [0 0 0 0 5 ...]
  printf("upper left corner of a:\n");  //       .............
  magma_sprint(5,5,a,n);                        // print part of a
// compute the eigenvalues  and eigenvectors for a symmetric,
// real nxn matrix; Magma version

  magma_ssyevd(MagmaVec,MagmaLower,n,r,n,w1,h_work,lwork,
                                      iwork,liwork,&info);

  printf("first 5 eigenvalues of a:\n");
  for(j=0;j<5;j++)
    printf("%f\n",w1[j]);             // print first eigenvalues
```

```
  printf("left upper corner of the matrix of eigenvectors:\n");
  magma_sprint(5,5,r,n); // part of the matrix of eigenvectors
// Lapack version
  lapackf77_ssyevd("V","L",&n,a,&n,w2,h_work,&lwork,iwork,
                                                  &liwork,&info);
// difference in  eigenvalues
  blasf77_saxpy( &n, &mione, w1, &incr, w2, &incr);
  error = lapackf77_slange( "M", &n, &ione, w2, &n, work );
  printf("difference in eigenvalues: %e\n",error);
  free(w1);                                    // free host memory
  free(w2);                                    // free host memory
  free(a);                                     // free host memory
  free(r);                                     // free host memory
  free(h_work);                                // free host memory
  magma_finalize();                              // finalize Magma
  return EXIT_SUCCESS;
}
// upper left corner of a:
//[
//   1.0000   0.       0.       0.       0.
//   0.       2.0000   0.       0.       0.
//   0.       0.       3.0000   0.       0.
//   0.       0.       0.       4.0000   0.
//   0.       0.       0.       0.       5.0000
//];
// first 5 eigenvalues of a:
// 1.000000
// 2.000000
// 3.000000
// 4.000000
// 5.000000
// left upper corner of the matrix of eigenvectors:
//[
//   1.0000   0.       0.       0.       0.
//   0.       1.0000   0.       0.       0.
//   0.       0.       1.0000   0.       0.
//   0.       0.       0.       1.0000   0.
//   0.       0.       0.       0.       1.0000
//];
// difference in eigenvalues: 0.000000e+00
```

4.7.2 `magma_dsyevd` - compute the eigenvalues and optionally eigenvectors of a symmetric real matrix in double precision, CPU interface, small matrix

This function computes in double precision all eigenvalues and, optionally, eigenvectors of a real symmetric matrix A defined on the host. The first parameter can take the values `MagmaVec,'V'` or `MagmaNoVec,'N'` and answers the question whether the eigenvectors are desired. If the eigenvectors are desired, it uses a divide and conquer algorithm. The symmetric matrix

A can be stored in lower (`MagmaLower,'L'`) or upper (`MagmaUpper,'U'`) mode. If the eigenvectors are desired, then on exit A contains orthonormal eigenvectors. The eigenvalues are stored in an array `w`. See magma-X.Y.Z/src/dsyevd.cpp for more details.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
int main( int argc, char** argv) {
  magma_init();                                  // initialize Magma
  magma_int_t n=1024, n2=n*n;
  double  *a, *r;             // a, r - nxn matrices on the host
  double  *h_work;                                  // workspace
  magma_int_t lwork;                               // h_work size
  magma_int_t  *iwork;                              // workspace
  magma_int_t  liwork;                             // iwork size
  double *w1, *w2;            // w1,w2 - vectors of  eigenvalues
  double error, work[1];      // used in difference computations
  magma_int_t ione = 1, i, j, info;
  double mione = -1.0;
  magma_int_t incr = 1, inci = 1;
  magma_dmalloc_cpu(&w1,n);              // host memory for real
  magma_dmalloc_cpu(&w2,n);                        // eigenvalues
  magma_dmalloc_cpu(&a,n2);                  // host memory for a
  magma_dmalloc_cpu(&r,n2);                  // host memory for r
// Query for workspace sizes
  double aux_work[1];
  magma_int_t aux_iwork[1];
  magma_dsyevd('V','L',n,r,n,w1,aux_work,-1,
                               aux_iwork,-1,&info );
  lwork  = (magma_int_t) aux_work[0];
  liwork = aux_iwork[0];
  iwork=(magma_int_t*)malloc(liwork*sizeof(magma_int_t));
  magma_dmalloc_cpu(&h_work,lwork); // host mem. for workspace
// define a, r                                //      [1 0 0 0 0 ...]
  for(i=0;i<n;i++){                          //      [0 2 0 0 0 ...]
    a[i*n+i]=(double)(i+1);                  // a = [0 0 3 0 0 ...]
    r[i*n+i]=(double)(i+1);                  //      [0 0 0 4 0 ...]
  }                                          //      [0 0 0 0 5 ...]
  printf("upper left corner of a:\n");  //       ............
  magma_dprint(5,5,a,n);                        // print part of a
// compute the eigenvalues  and eigenvectors for a symmetric,
// real nxn matrix; Magma version

  magma_dsyevd(MagmaVec,MagmaLower,n,r,lda,w1,h_work,lwork,
                                       iwork,liwork,&info);

  printf("first 5 eigenvalues of a:\n");
  for(j=0;j<5;j++)
    printf("%f\n",w1[j]);             // print first eigenvalues
```

```
  printf("left upper corner of the matrix of eigenvectors:\n");
  magma_dprint(5,5,r,n); // part of the matrix of eigenvectors
// Lapack version
  lapackf77_dsyevd("V","L",&n,a,&n,w2,h_work,&lwork,iwork,
                                                  &liwork,&info);
// difference in  eigenvalues
  blasf77_daxpy( &n, &mione, w1, &incr, w2, &incr);
  error = lapackf77_dlange( "M", &n, &ione, w2, &n, work );
  printf("difference in eigenvalues: %e\n",error);
  free(w1);                                   // free host memory
  free(w2);                                   // free host memory
  free(a);                                    // free host memory
  free(r);                                    // free host memory
  free(h_work);                               // free host memory
  magma_finalize();                             // finalize Magma
  return EXIT_SUCCESS;
}
// upper left corner of a:
//[
//   1.0000   0.       0.       0.       0.
//   0.       2.0000   0.       0.       0.
//   0.       0.       3.0000   0.       0.
//   0.       0.       0.       4.0000   0.
//   0.       0.       0.       0.       5.0000
//];
// first 5 eigenvalues of a:
// 1.000000
// 2.000000
// 3.000000
// 4.000000
// 5.000000
// left upper corner of the matrix of eigenvectors:
//[
//   1.0000   0.       0.       0.       0.
//   0.       1.0000   0.       0.       0.
//   0.       0.       1.0000   0.       0.
//   0.       0.       0.       1.0000   0.
//   0.       0.       0.       0.       1.0000
//];
```

4.7.3 `magma_ssyevd` - compute the eigenvalues and optionally eigenvectors of a symmetric real matrix in single precision, CPU interface, big matrix

This function computes in single precision all eigenvalues and, optionally, eigenvectors of a real symmetric matrix A defined on the host. The first parameter can take the values `MagmaVec,'V'` or `MagmaNoVec,'N'` and answers the question whether the eigenvectors are desired. If the eigenvectors are desired, it uses a divide and conquer algorithm. The symmetric matrix A can be stored in lower (`MagmaLower,'L'`) or upper (`MagmaUpper,'U'`)

mode. If the eigenvectors are desired, then on exit A contains orthonormal eigenvectors. The eigenvalues are stored in an array `w`. See magma-X.Y.Z/src/ssyevd.cpp for more details.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
int main( int argc, char** argv) {
  magma_init();                              // initialize Magma
  magma_timestr_t  start, end;
  float  gpu_time, cpu_time;
  magma_int_t n=8192, n2=n*n;
  float  *a, *r;            // a, r - nxn matrices on the host
  float  *h_work;                                 // workspace
  magma_int_t lwork;                             // h_work size
  magma_int_t  *iwork;                            // workspace
  magma_int_t  liwork;                           // iwork size
  float *w1, *w2;           // w1,w2 - vectors of  eigenvalues
  float error, work[1];     // used in difference computations
  magma_int_t ione = 1, i, j, info;
  float mione = -1.0f;
  magma_int_t incr = 1, inci = 1;
  magma_int_t ISEED[4] = {0,0,0,1};                    // seed
  magma_smalloc_cpu(&w1,n);             // host memory for real
  magma_smalloc_cpu(&w2,n);                      // eigenvalues
  magma_smalloc_cpu(&a,n2);                 // host memory for a
  magma_smalloc_cpu(&r,n2);                 // host memory for r
// Query for workspace sizes
  float aux_work[1];
  magma_int_t aux_iwork[1];
  magma_ssyevd('V','L',n,r,n,w1,aux_work,-1,
                              aux_iwork,-1,&info );
  lwork  = (magma_int_t) aux_work[0];
  liwork = aux_iwork[0];
  iwork=(magma_int_t*)malloc(liwork*sizeof(magma_int_t));
  magma_smalloc_cpu(&h_work,lwork);    // memory for workspace
// Random  matrix a,  copy a -> r
  lapackf77_slarnv(&ione,ISEED,&n2,a);
  lapackf77_slacpy(MagmaUpperLowerStr,&n,&n,a,&n,r,&n);
  start = get_current_time();
// compute the eigenvalues  and eigenvectors for a symmetric,
// real nxn matrix; Magma version

  magma_ssyevd(MagmaVec,MagmaLower,n,r,n,w1,h_work,lwork,
                                       iwork,liwork,&info);

  end = get_current_time();
  gpu_time = GetTimerValue(start,end) / 1e3;
  printf("ssyevd gpu time: %7.5f sec.\n",gpu_time);   // Magma
// Lapack version                                       // time
```

```
  start = get_current_time();
  lapackf77_ssyevd("V","L",&n,a,&n,w2,h_work,&lwork,iwork,
                                                &liwork,&info);
  end = get_current_time();
  cpu_time = GetTimerValue(start,end) / 1e3;
  printf("ssyevd cpu time: %7.5f sec.\n",cpu_time);  // Lapack
// difference in  eigenvalues                          // time
  blasf77_saxpy( &n, &mione, w1, &incr, w2, &incr);
  error = lapackf77_slange( "M", &n, &ione, w2, &n, work );
  printf("difference in eigenvalues: %e\n",error);
  free(w1);                                  // free host memory
  free(w2);                                  // free host memory
  free(a);                                   // free host memory
  free(r);                                   // free host memory
  free(h_work);                              // free host memory
  magma_finalize();                            // finalize Magma
  return EXIT_SUCCESS;
}
// ssyevd gpu time: 19.18347 sec.                      // 1 GPU
// ssyevd cpu time: 19.19710 sec.                     // 2 CPUs
// difference in eigenvalues: 9.765625e-04
```

4.7.4 magma_dsyevd - compute the eigenvalues and optionally eigenvectors of a symmetric real matrix in double precision, CPU interface, big matrix

This function computes in double precision all eigenvalues and, optionally, eigenvectors of a real symmetric matrix A defined on the host. The first parameter can take the values `MagmaVec,'V'` or `MagmaNoVec,'N'` and answers the question whether the eigenvectors are desired. If the eigenvectors are desired, it uses a divide and conquer algorithm. The symmetric matrix A can be stored in lower (`MagmaLower,'L'`) or upper (`MagmaUpper,'U'`) mode. If the eigenvectors are desired, then on exit A contains orthonormal eigenvectors. The eigenvalues are stored in an array `w`. See magma-X.Y.Z/src/dsyevd.cpp for more details.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
int main( int argc, char** argv) {
  magma_init();                              // initialize Magma
  magma_timestr_t  start, end;
  double  gpu_time, cpu_time;
  magma_int_t n=8192, n2=n*n;
  double  *a, *r;           // a, r - nxn matrices on the host
  double  *h_work;                                 // workspace
  magma_int_t lwork;                              // h_work size
  magma_int_t  *iwork;                             // workspace
```

```
  magma_int_t  liwork;                              // iwork size
  double *w1, *w2;           // w1,w2 - vectors of  eigenvalues
  double error, work[1];     // used in difference computations
  magma_int_t ione = 1, i, j, info;
  double mione = -1.0;
  magma_int_t incr = 1, inci = 1;
  magma_int_t ISEED[4] = {0,0,0,1};                     // seed
  magma_dmalloc_cpu(&w1,n);             // host memory for real
  magma_dmalloc_cpu(&w2,n);                     // eigenvalues
  magma_dmalloc_cpu(&a,n2);                // host memory for a
  magma_dmalloc_cpu(&r,n2);                // host memory for r
// Query for workspace sizes
  double aux_work[1];
  magma_int_t aux_iwork[1];
  magma_dsyevd('V','L',n,r,n,w1,aux_work,-1,
                               aux_iwork,-1,&info );
  lwork  = (magma_int_t) aux_work[0];
  liwork = aux_iwork[0];
  iwork=(magma_int_t*)malloc(liwork*sizeof(magma_int_t));
  magma_dmalloc_cpu(&h_work,lwork);     // memory for workspace
// Random  matrix a,  copy a -> r
  lapackf77_dlarnv(&ione,ISEED,&n2,a);
  lapackf77_dlacpy(MagmaUpperLowerStr,&n,&n,a,&n,r,&n);
  start = get_current_time();
// compute the eigenvalues  and eigenvectors for a symmetric,
// real nxn matrix; Magma version

  magma_dsyevd(MagmaVec,MagmaLower,n,r,n,w1,h_work,lwork,
                                        iwork,liwork,&info);

  end = get_current_time();
  gpu_time = GetTimerValue(start,end) / 1e3;
  printf("dsyevd gpu time: %7.5f sec.\n",gpu_time);    // Magma
// Lapack version                                        // time
  start = get_current_time();
  lapackf77_dsyevd("V","L",&n,a,&n,w2,h_work,&lwork,iwork,
                                          &liwork,&info);
  end = get_current_time();
  cpu_time = GetTimerValue(start,end) / 1e3;
  printf("dsyevd cpu time: %7.5f sec.\n",cpu_time);  // Lapack
// difference in  eigenvalues                           // time
  blasf77_daxpy( &n, &mione, w1, &incr, w2, &incr);
  error = lapackf77_dlange( "M", &n, &ione, w2, &n, work );
  printf("difference in eigenvalues: %e\n",error);
  free(w1);                                  // free host memory
  free(w2);                                  // free host memory
  free(a);                                   // free host memory
  free(r);                                   // free host memory
  free(h_work);                              // free host memory
  magma_finalize();                            // finalize Magma
  return EXIT_SUCCESS;
}
```

```
// dsyevd gpu time: 34.93392 sec.                    // 1 GPU
// dsyevd cpu time: 44.03702 sec.                   // 2 CPUs
// difference in eigenvalues: 1.273293e-11
```

4.7.5 magma_ssyevd_gpu - compute the eigenvalues and optionally eigenvectors of a symmetric real matrix in single precision, GPU interface, small matrix

This function computes in single precision all eigenvalues and, optionally, eigenvectors of a real symmetric matrix A defined on the device. The first parameter can take the values `MagmaVec,'V'` or `MagmaNoVec,'N'` and answers the question whether the eigenvectors are desired. If the eigenvectors are desired, it uses a divide and conquer algorithm. The symmetric matrix A can be stored in lower (`MagmaLower,'L'`) or upper (`MagmaUpper,'U'`) mode. If the eigenvectors are desired, then on exit A contains orthonormal eigenvectors. The eigenvalues are stored in an array `w`. See magma-X.Y.Z/src/ssyevd_gpu.cpp for more details.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
int main( int argc, char** argv) {
  magma_init();                              // initialize Magma
  magma_int_t n=1024, n2=n*n;
  float  *a, *r;             // a, r - nxn matrices on the host
  float *d_r;                        // nxn matrix on the device
  float  *h_work;                                 // workspace
  magma_int_t lwork;                             // h_work size
  magma_int_t  *iwork;                            // workspace
  magma_int_t  liwork;                           // iwork size
  float *w1, *w2;            // w1,w2 - vectors of  eigenvalues
  float error, work[1];      // used in difference computations
  magma_int_t ione = 1, i, j, info;
  float mione = -1.0f;
  magma_int_t incr = 1, inci = 1;
  magma_smalloc_cpu(&w1,n);             // host memory for real
  magma_smalloc_cpu(&w2,n);                       // eigenvalues
  magma_smalloc_cpu(&a,n2);                // host memory for a
  magma_smalloc_cpu(&r,n2);                // host memory for r
  magma_smalloc(&d_r,n2);               // device memory for d_r
// Query for workspace sizes
  float aux_work[1];
  magma_int_t aux_iwork[1];
  magma_ssyevd_gpu('V','L',n,d_r,n,w1,r,n,aux_work,-1,
                             aux_iwork,-1,&info );
  lwork  = (magma_int_t) aux_work[0];
  liwork = aux_iwork[0];
  iwork=(magma_int_t*)malloc(liwork*sizeof(magma_int_t));
```

```
  magma_smalloc_cpu(&h_work,lwork);     // memory for workspace
// define a, r                           //      [1 0 0 0 0 ...]
  for(i=0;i<n;i++){                     //      [0 2 0 0 0 ...]
    a[i*n+i]=(float)(i+1);              // a = [0 0 3 0 0 ...]
    r[i*n+i]=(float)(i+1);              //      [0 0 0 4 0 ...]
  }                                     //      [0 0 0 0 5 ...]
  printf("upper left corner of a:\n");  //       ..............
  magma_sprint(5,5,a,n);                        // print part of a
  magma_ssetmatrix( n, n, a, n, d_r, n);       // copy a -> d_r
// compute the eigenvalues  and eigenvectors for a symmetric,
// real nxn matrix; Magma version

  magma_ssyevd_gpu(MagmaVec,MagmaLower,n,d_r,n,w1,r,n,
                           h_work,lwork,iwork,liwork,&info);

  printf("first 5 eigenvalues of a:\n");
  for(j=0;j<5;j++)
    printf("%f\n",w1[j]);             // print first eigenvalues
  printf("left upper corner of the matrix of eigenvectors:\n");
  magma_sgetmatrix( n, n, d_r, n, r, n );      // copy d_r -> r
  magma_sprint(5,5,r,n); // part of the matrix of eigenvectors
// Lapack version
  lapackf77_ssyevd("V","L",&n,a,&n,w2,h_work,&lwork,iwork,
                                              &liwork,&info);
// difference in  eigenvalues
  blasf77_saxpy( &n, &mione, w1, &incr, w2, &incr);
  error = lapackf77_slange( "M", &n, &ione, w2, &n, work );
  printf("difference in eigenvalues: %e\n",error);
  free(w1);                                    // free host memory
  free(w2);                                    // free host memory
  free(a);                                     // free host memory
  free(r);                                     // free host memory
  free(h_work);                                // free host memory
  magma_free(d_r);                            // free device memory
  magma_finalize();                               // finalize Magma
  return EXIT_SUCCESS;
}
// upper left corner of a:
//[
//   1.0000   0.       0.       0.       0.
//   0.       2.0000   0.       0.       0.
//   0.       0.       3.0000   0.       0.
//   0.       0.       0.       4.0000   0.
//   0.       0.       0.       0.       5.0000
//];
// first 5 eigenvalues of a:
// 1.000000
// 2.000000
// 3.000000
// 4.000000
// 5.000000
// left upper corner of the matrix of eigenvectors:
```

```
// [
//   1.0000   0.       0.       0.       0.
//   0.       1.0000   0.       0.       0.
//   0.       0.       1.0000   0.       0.
//   0.       0.       0.       1.0000   0.
//   0.       0.       0.       0.       1.0000
//];
// difference in eigenvalues: 0.000000e+00
```

4.7.6 magma_dsyevd_gpu - compute the eigenvalues and optionally eigenvectors of a symmetric real matrix in double precision, GPU interface, small matrix

This function computes in double precision all eigenvalues and, optionally, eigenvectors of a real symmetric matrix A defined on the device. The first parameter can take the values `MagmaVec,'V'` or `MagmaNoVec,'N'` and answers the question whether the eigenvectors are desired. If the eigenvectors are desired, it uses a divide and conquer algorithm. The symmetric matrix A can be stored in lower (`MagmaLower,'L'`) or upper (`MagmaUpper,'U'`) mode. If the eigenvectors are desired, then on exit A contains orthonormal eigenvectors. The eigenvalues are stored in an array `w`. See magma-X.Y.Z/src/dsyevd_gpu.cpp for more details.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
int main( int argc, char** argv) {
  magma_init();                              // initialize Magma
  magma_int_t n=1024, n2=n*n;
  double  *a, *r;           // a, r - nxn matrices on the host
  double *d_r;                     // nxn matrix on the device
  double  *h_work;                                 // workspace
  magma_int_t lwork;                             // h_work size
  magma_int_t  *iwork;                             // workspace
  magma_int_t  liwork;                            // iwork size
  double *w1, *w2;          // w1,w2 - vectors of  eigenvalues
  double error, work[1];    // used in difference computations
  magma_int_t ione = 1, i, j, info;
  double mione = -1.0;
  magma_int_t incr = 1, inci = 1;
  magma_dmalloc_cpu(&w1,n);            // host memory for real
  magma_dmalloc_cpu(&w2,n);                     // eigenvalues
  magma_dmalloc_cpu(&a,n2);                // host memory for a
  magma_dmalloc_cpu(&r,n2);                // host memory for r
  magma_dmalloc(&d_r,n2);               // device memory for d_r
// Query for workspace sizes
  double aux_work[1];
  magma_int_t aux_iwork[1];
```

```
  magma_dsyevd_gpu('V','L',n,d_r,n,w1,r,n,aux_work,-1,
                                aux_iwork,-1,&info );
  lwork  = (magma_int_t) aux_work[0];
  liwork = aux_iwork[0];
  iwork=(magma_int_t*)malloc(liwork*sizeof(magma_int_t));
  magma_dmalloc_cpu(&h_work,lwork);     // memory for workspace
// define a, r                           //      [1 0 0 0 0 ...]
  for(i=0;i<n;i++){                     //      [0 2 0 0 0 ...]
    a[i*n+i]=(double)(i+1);             // a = [0 0 3 0 0 ...]
    r[i*n+i]=(double)(i+1);             //      [0 0 0 4 0 ...]
  }                                     //      [0 0 0 0 5 ...]
  printf("upper left corner of a:\n"); //       ..............
  magma_dprint(5,5,a,n);                        // print part of a
  magma_dsetmatrix( n, n, a, n, d_r, n);       // copy a -> d_r
// compute the eigenvalues  and eigenvectors for a symmetric,
// real nxn matrix; Magma version

  magma_dsyevd_gpu(MagmaVec,MagmaLower,n,d_r,n,w1,r,n,
                           h_work,lwork,iwork,liwork,&info);

  printf("first 5 eigenvalues of a:\n");
  for(j=0;j<5;j++)
    printf("%f\n",w1[j]);           // print first eigenvalues
  printf("left upper corner of the matrix of eigenvectors:\n");
  magma_dgetmatrix( n, n, d_r, n, r, n );      // copy d_r -> r
  magma_dprint(5,5,r,n); // part of the matrix of eigenvectors
// Lapack version
  lapackf77_dsyevd("V","L",&n,a,&n,w2,h_work,&lwork,iwork,
                                          &liwork,&info);
// difference in  eigenvalues
  blasf77_daxpy( &n, &mione, w1, &incr, w2, &incr);
  error = lapackf77_dlange( "M", &n, &ione, w2, &n, work );
  printf("difference in eigenvalues: %e\n",error);
  free(w1);                                    // free host memory
  free(w2);                                    // free host memory
  free(a);                                     // free host memory
  free(r);                                     // free host memory
  free(h_work);                                // free host memory
  magma_free(d_r);                            // free device memory
  magma_finalize();                              // finalize Magma
  return EXIT_SUCCESS;
}
// upper left corner of a:
//[
//   1.0000   0.       0.       0.       0.
//   0.       2.0000   0.       0.       0.
//   0.       0.       3.0000   0.       0.
//   0.       0.       0.       4.0000   0.
//   0.       0.       0.       0.       5.0000
//];
// first 5 eigenvalues of a:
// 1.000000
```

```
// 2.000000
// 3.000000
// 4.000000
// 5.000000
// left upper corner of the matrix of eigenvectors:
//[
//    1.0000    0.        0.        0.        0.
//    0.        1.0000    0.        0.        0.
//    0.        0.        1.0000    0.        0.
//    0.        0.        0.        1.0000    0.
//    0.        0.        0.        0.        1.0000
//];
// difference in eigenvalues: 0.000000e+00
```

4.7.7 magma_ssyevd_gpu - compute the eigenvalues and optionally eigenvectors of a symmetric real matrix in single precision, GPU interface, big matrix

This function computes in single precision all eigenvalues and, optionally, eigenvectors of a real symmetric matrix A defined on the device. The first parameter can take the values `MagmaVec,'V'` or `MagmaNoVec,'N'` and answers the question whether the eigenvectors are desired. If the eigenvectors are desired, it uses a divide and conquer algorithm. The symmetric matrix A can be stored in lower (`MagmaLower,'L'`) or upper (`MagmaUpper,'U'`) mode. If the eigenvectors are desired, then on exit A contains orthonormal eigenvectors. The eigenvalues are stored in an array `w`. See magma-X.Y.Z/src/ssyevd_gpu.cpp for more details.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
int main( int argc, char** argv) {
  magma_init();                              // initialize Magma
  magma_timestr_t  start, end;
  float  gpu_time, cpu_time;
  magma_int_t n=8192, n2=n*n;
  float  *a, *r;            // a, r - nxn matrices on the host
  float *d_r;                      // nxn matrix on the device
  float  *h_work;                                 // workspace
  magma_int_t lwork;                             // h_work size
  magma_int_t  *iwork;                             // workspace
  magma_int_t  liwork;                            // iwork size
  float *w1, *w2;           // w1,w2 - vectors of  eigenvalues
  float error, work[1];     // used in difference computations
  magma_int_t ione = 1, i, j, info;
  float mione = -1.0f;
  magma_int_t incr = 1, inci = 1;
  magma_int_t ISEED[4] = {0,0,0,1};                    // seed
```

```
  magma_smalloc_cpu(&w1,n);              // host memory for real
  magma_smalloc_cpu(&w2,n);                       // eigenvalues
  magma_smalloc_cpu(&a,n2);                 // host memory for a
  magma_smalloc_cpu(&r,n2);                 // host memory for r
  magma_smalloc(&d_r,n2);                // device memory for d_r
// Query for workspace sizes
  float aux_work[1];
  magma_int_t aux_iwork[1];
  magma_ssyevd_gpu('V','L',n,d_r,n,w1,r,n,aux_work,-1,
                              aux_iwork,-1,&info );
  lwork  = (magma_int_t) aux_work[0];
  liwork = aux_iwork[0];
  iwork=(magma_int_t*)malloc(liwork*sizeof(magma_int_t));
  magma_smalloc_cpu(&h_work,lwork);     // memory for workspace
// Random  matrix a,  copy a -> r
  lapackf77_slarnv(&ione,ISEED,&n2,a);
  lapackf77_slacpy(MagmaUpperLowerStr,&n,&n,a,&n,r,&n);
  magma_ssetmatrix( n, n, a, n, d_r, n);       // copy a -> d_r
// compute the eigenvalues  and eigenvectors for a symmetric,
// real nxn matrix; Magma version
  start = get_current_time();

  magma_ssyevd_gpu(MagmaVec,MagmaLower,n,d_r,n,w1,r,n,
                              h_work,lwork,iwork,liwork,&info);

  end = get_current_time();
  gpu_time = GetTimerValue(start,end) / 1e3;
  printf("ssyevd gpu time: %7.5f sec.\n",gpu_time);//Mag.time
// Lapack version
  start = get_current_time();
  lapackf77_ssyevd("V","L",&n,a,&n,w2,h_work,&lwork,iwork,
                                          &liwork,&info);
  end = get_current_time();
  cpu_time = GetTimerValue(start,end) / 1e3;
  printf("ssyevd cpu time: %7.5f sec.\n",cpu_time);  // Lapack
// difference in  eigenvalues                          // time
  blasf77_saxpy( &n, &mione, w1, &incr, w2, &incr);
  error = lapackf77_slange( "M", &n, &ione, w2, &n, work );
  printf("difference in eigenvalues: %e\n",error);
  free(w1);                                  // free host memory
  free(w2);                                  // free host memory
  free(a);                                   // free host memory
  free(r);                                   // free host memory
  free(h_work);                              // free host memory
  magma_free(d_r);                         // free device memory
  magma_finalize();                            // finalize Magma
  return EXIT_SUCCESS;
}
// ssyevd gpu time: 19.50048 sec.                      // 1 GPU
// ssyevd cpu time: 19.86725 sec.                     // 2 CPUs
// difference in eigenvalues: 1.358032e-04
```

4.7.8 `magma_dsyevd_gpu` - compute the eigenvalues and optionally eigenvectors of a symmetric real matrix in double precision, GPU interface, big matrix

This function computes in double precision all eigenvalues and, optionally, eigenvectors of a real symmetric matrix A defined on the device. The first parameter can take the values `MagmaVec,'V'` or `MagmaNoVec,'N'` and answers the question whether the eigenvectors are desired. If the eigenvectors are desired, it uses a divide and conquer algorithm. The symmetric matrix A can be stored in lower (`MagmaLower,'L'`) or upper (`MagmaUpper,'U'`) mode. If the eigenvectors are desired, then on exit A contains orthonormal eigenvectors. The eigenvalues are stored in an array `w`. See `magma-X.Y.Z/src/dsyevd_gpu.cpp` for more details.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
int main( int argc, char** argv) {
  magma_init();                              // initialize Magma
  magma_timestr_t  start, end;
  double  gpu_time, cpu_time;
  magma_int_t n=8192, n2=n*n;
  double  *a, *r;           // a, r - nxn matrices on the host
  double *d_r;                      // nxn matrix on the device
  double  *h_work;                                // workspace
  magma_int_t lwork;                             // h_work size
  magma_int_t  *iwork;                            // workspace
  magma_int_t  liwork;                           // iwork size
  double *w1, *w2;          // w1,w2 - vectors of  eigenvalues
  double error, work[1];    // used in difference computations
  magma_int_t ione = 1, i, j, info;
  double mione = -1.0;
  magma_int_t incr = 1, inci = 1;
  magma_int_t ISEED[4] = {0,0,0,1};                    // seed
  magma_dmalloc_cpu(&w1,n);            // host memory for real
  magma_dmalloc_cpu(&w2,n);                     // eigenvalues
  magma_dmalloc_cpu(&a,n2);                // host memory for a
  magma_dmalloc_cpu(&r,n2);                // host memory for r
  magma_dmalloc(&d_r,n2);             // device memory for d_r
// Query for workspace sizes
  double aux_work[1];
  magma_int_t aux_iwork[1];
  magma_dsyevd_gpu('V','L',n,d_r,n,w1,r,n,aux_work,-1,
                           aux_iwork,-1,&info );
  lwork  = (magma_int_t) aux_work[0];
  liwork = aux_iwork[0];
  iwork=(magma_int_t*)malloc(liwork*sizeof(magma_int_t));
  magma_dmalloc_cpu(&h_work,lwork);    // memory for workspace
// Random  matrix a,  copy a -> r
```

```
  lapackf77_dlarnv(&ione,ISEED,&n2,a);
  lapackf77_dlacpy(MagmaUpperLowerStr,&n,&n,a,&n,r,&n);
  magma_dsetmatrix( n, n, a, n, d_r, n);      // copy a -> d_r
// compute the eigenvalues  and eigenvectors for a symmetric,
// real nxn matrix; Magma version
  start = get_current_time();

  magma_dsyevd_gpu(MagmaVec,MagmaLower,n,d_r,n,
                       w1,r,n,h_work,lwork,iwork,liwork,&info);

  end = get_current_time();
  gpu_time = GetTimerValue(start,end) / 1e3;
  printf("dsyevd gpu time: %7.5f sec.\n",gpu_time);//Mag. time
// Lapack version
  start = get_current_time();
  lapackf77_dsyevd("V","L",&n,a,&n,w2,h_work,&lwork,iwork,
                                          &liwork,&info);
  end = get_current_time();
  cpu_time = GetTimerValue(start,end) / 1e3;
  printf("dsyevd cpu time: %7.5f sec.\n",cpu_time);  // Lapack
// difference in  eigenvalues                          // time
  blasf77_daxpy( &n, &mione, w1, &incr, w2, &incr);
  error = lapackf77_dlange( "M", &n, &ione, w2, &n, work );
  printf("difference in eigenvalues: %e\n",error);
  free(w1);                                  // free host memory
  free(w2);                                  // free host memory
  free(a);                                   // free host memory
  free(r);                                   // free host memory
  free(h_work);                              // free host memory
  magma_free(d_r);                         // free device memory
  magma_finalize();                           // finalize Magma
  return EXIT_SUCCESS;
}
// dsyevd gpu time: 35.31227 sec.                      // 1 GPU
// dsyevd cpu time: 43.09366 sec.                     // 2 CPUs
// difference in eigenvalues: 1.364242e-12
```

4.8 Singular value decomposition

4.8.1 magma_sgesvd - compute the singular value decomposition of a general real matrix in single precision, CPU interface

This function computes in single precision the singular value decomposition of an $m \times n$ matrix defined on the host:

$$A = u\ \sigma\ v^T,$$

where σ is an $m \times n$ matrix which is zero except for its $\min(m, n)$ diagonal elements (singular values), u is an $m \times m$ orthogonal matrix and v is an

$n \times n$ orthogonal matrix. The first $\min(m,n)$ columns of u and v are the left and right singular vectors of A. The first argument can take the following values:
'A' - all m columns of u are returned in an array u;
'S' - the first $\min(m,n)$ columns of u (the left singular vectors) are returned in the array u;
'O' - the first $\min(m,n)$ columns of u are overwritten on the array A;
'N' - no left singular vectors are computed.
Similarly the second argument can take the following values:
'A' - all n rows of v^T are returned in an array vt;
'S' - the first $\min(m,n)$ rows of v^T (the right singular vectors) are returned in the array vt;
'O' - the first $\min(m,n)$ rows of v^T are overwritten on the array A;
'N' - no right singular vectors are computed.
The singular values are stored in an array s.
See magma-X.Y.Z/src/sgesvd.cpp for more details.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
#define min(a,b)  (((a)<(b))?(a):(b))
int main( int argc, char** argv)
{
  magma_init();                              // initialize Magma
  real_Double_t   gpu_time, cpu_time;
// Matrix size
  magma_int_t m=8192, n=8192, n2=m*n, min_mn=min(m,n);
  float *a, *r;                             // a,r - mxn matrices
  float *u, *vt;// u - mxm matrix, vt - nxn matrix on the host
  float *s1, *s2;                 // vectors of singular values
  magma_int_t info;
  magma_int_t ione  = 1;
  float work[1], error = 1.;// used in difference computations
  float mone = -1.0, *h_work;             // h_work - workspace
  magma_int_t lwork;                          // workspace size
  magma_int_t ISEED[4] = {0,0,0,1};                    // seed
// Allocate host memory
  magma_smalloc_cpu(&a,n2);                // host memory for a
  magma_smalloc_cpu(&vt,n*n);             // host memory for vt
  magma_smalloc_cpu(&u,m*m);               // host memory for u
  magma_smalloc_cpu(&s1,min_mn);          // host memory for s1
  magma_smalloc_cpu(&s2,min_mn);          // host memory for s2
  magma_smalloc_pinned(&r,n2);             // host memory for r
  magma_int_t nb = magma_get_sgesvd_nb(n);  // opt. block size
  lwork = (m+n)*nb + 3*min_mn;
  magma_smalloc_pinned(&h_work,lwork); // host mem. for h_work
// Random  matrices
  lapackf77_slarnv(&ione,ISEED,&n2,a);
  lapackf77_slacpy(MagmaUpperLowerStr,&m,&n,a,&m,r,&m);// a->r
```

```
// MAGMA
  gpu_time = magma_wtime();
// compute the singular value decomposition of a
// and optionally the left and right singular vectors:
// a = u*sigma*vt; the diagonal elements of sigma (s1 array)
// are the singular values of a in descending order
// the first min(m,n) columns of u contain the left sing. vec.
// the first min(m,n) columns of vt contain the right sing. v.

  magma_sgesvd('S','S',m,n,r,m,s1,u,m,vt,n,h_work,
                                              lwork,&info );

  gpu_time = magma_wtime() - gpu_time;
  printf("sgesvd gpu time:  %7.5f\n", gpu_time); // Magma time
// LAPACK
  cpu_time = magma_wtime();
  lapackf77_sgesvd("S","S",&m,&n,a,&m,s2,u,&m,vt,&n,h_work,
                                                &lwork,&info);
  cpu_time = magma_wtime() - cpu_time;
  printf("sgesvd cpu time:  %7.5f\n", cpu_time);// Lapack time
// difference
  error=lapackf77_slange("f",&min_mn,&ione,s1,&min_mn,work);
  blasf77_saxpy(&min_mn,&mone,s1,&ione,s2,&ione);
  error=lapackf77_slange("f",&min_mn,&ione,s2,&min_mn,work)/
                                                      error;
  printf("difference:  %e\n", error );// difference in singul.
                                                    // values
// Free memory
  free(a);                                  // free host memory
  free(vt);                                 // free host memory
  free(s1);                                 // free host memory
  free(s2);                                 // free host memory
  free(u);                                  // free host memory
  magma_free_pinned(h_work);                // free host memory
  magma_free_pinned(r);                     // free host memory
  magma_finalize( );                          // finalize Magma
  return EXIT_SUCCESS;
}
// sgesvd gpu time: 110.83179  sec.                   // 1 GPU
// sgesvd cpu time: 136.71580  sec.                  // 2 CPUs
// difference:  1.470985e-06
```

4.8.2 magma_dgesvd - compute the singular value decomposition of a general real matrix in double precision, CPU interface

This function computes in double precision the singular value decomposition of an $m \times n$ matrix defined on the host:

$$A = u\ \sigma\ v^T,$$

where σ is an $m \times n$ matrix which is zero except for its $\min(m,n)$ diagonal elements (singular values), u is an $m \times m$ orthogonal matrix and v is an $n \times n$ orthogonal matrix. The first $\min(m,n)$ columns of u and v are the left and right singular vectors of A. The first argument can take the following values:
'A' - all m columns of u are returned in an array u;
'S' - the first $\min(m,n)$ columns of u (the left singular vectors) are returned in the array u;
'O' - the first $\min(m,n)$ columns of u are overwritten on the array A;
'N' - no left singular vectors are computed.
Similarly the second argument can take the following values:
'A' - all n rows of v^T are returned in an array vt;
'S' - the first $\min(m,n)$ rows of v^T (the right singular vectors) are returned in the array vt;
'O' - the first $\min(m,n)$ rows of v^T are overwritten on the array A;
'N' - no right singular vectors are computed.
The singular values are stored in an array s.
See magma-X.Y.Z/src/dgesvd.cpp for more details.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
#define min(a,b)  (((a)<(b))?(a):(b))
int main( int argc, char** argv)
{
  magma_init();                              // initialize Magma
  real_Double_t   gpu_time, cpu_time;
// Matrix size
  magma_int_t m=8192, n=8192, n2=m*n, min_mn=min(m,n);
  double *a, *r;                            // a,r - mxn matrices
  double *u, *vt;//u - mxm matrix, vt - nxn matrix on the host
  double *s1, *s2;                // vectors of singular values
  magma_int_t info;
  magma_int_t ione  = 1;
  double work[1], error = 1.;//used in difference computations
  double mone = -1.0, *h_work;            // h_work - workspace
  magma_int_t lwork;                          // workspace size
  magma_int_t ISEED[4] = {0,0,0,1};                    // seed
// Allocate host memory
  magma_dmalloc_cpu(&a,n2);                 // host memory for a
  magma_dmalloc_cpu(&vt,n*n);              // host memory for vt
  magma_dmalloc_cpu(&u,m*m);                // host memory for u
  magma_dmalloc_cpu(&s1,min_mn);           // host memory for s1
  magma_dmalloc_cpu(&s2,min_mn);           // host memory for s2
  magma_dmalloc_pinned(&r,n2);              // host memory for r
  magma_int_t nb = magma_get_dgesvd_nb(n);  // opt. block size
  lwork = (m+n)*nb + 3*min_mn;
  magma_dmalloc_pinned(&h_work,lwork); // host mem. for h_work
```

```
// Random  matrices
  lapackf77_dlarnv(&ione,ISEED,&n2,a);
  lapackf77_dlacpy(MagmaUpperLowerStr,&m,&n,a,&m,r,&m); //a->r
// MAGMA
  gpu_time = magma_wtime();
// compute the singular value decomposition of a
// and optionally the left and right singular vectors:
// a = u*sigma*vt; the diagonal elements of sigma (s1 array)
// are the singular values of a in descending order
// the first min(m,n) columns of u contain the left sing. vec.
// the first min(m,n) columns of vt contain the right sing. v.

  magma_dgesvd('S','S',m,n,r,m,s1,u,m,vt,n,h_work,
                                              lwork,&info );

  gpu_time = magma_wtime() - gpu_time;
  printf("dgesvd gpu time:  %7.5f\n", gpu_time); // Magma time
// LAPACK
  cpu_time = magma_wtime();
  lapackf77_dgesvd("S","S",&m,&n,a,&m,s2,u,&m,vt,&n,h_work,
                                                 &lwork,&info);
  cpu_time = magma_wtime() - cpu_time;
  printf("dgesvd cpu time:  %7.5f\n", cpu_time);// Lapack time
// difference
  error=lapackf77_dlange("f",&min_mn,&ione,s1,&min_mn,work);
  blasf77_daxpy(&min_mn,&mone,s1,&ione,s2,&ione);
  error=lapackf77_dlange("f",&min_mn,&ione,s2,&min_mn,work)/
                                                       error;
  printf("difference:  %e\n", error );// difference in singul.
                                                    // values
// Free memory
  free(a);                                  // free host memory
  free(vt);                                 // free host memory
  free(s1);                                 // free host memory
  free(s2);                                 // free host memory
  free(u);                                  // free host memory
  magma_free_pinned(h_work);                // free host memory
  magma_free_pinned(r);                     // free host memory
  magma_finalize( );                          // finalize Magma
  return EXIT_SUCCESS;
}
// dgesvd gpu time: 101.91289  sec.                  // 1 GPU
// dgesvd cpu time: 177.75227  sec.                 // 2 CPUs
// difference:  3.643387e-15
```

4.8.3 `magma_sgebrd` - reduce a real matrix to bidiagonal form by orthogonal transformations in single precision, CPU interface

This function reduces in single precision an $m \times n$ matrix A defined on the host to upper or lower bidiagonal form by orthogonal transformations:

$$Q^T \; A \; P = B,$$

where P, Q are orthogonal and B is bidiagonal. If $m \geq n$, B is upper bidiagonal; if $m < n$, B is lower bidiagonal. The obtained diagonal and the super/subdiagonal are written to diag and offdiag arrays respectively. If $m \geq n$, the elements below the diagonal, with the array *tauq* represent the orthogonal matrix Q as a product of elementary reflectors $H_k = I - tauq_k \cdot v_k \cdot v_k^T$, and the elements above the first superdiagonal with the array *taup* represent the orthogonal matrix P as a product of elementary reflectors $G_k = I - taup_k \cdot u_k \cdot u_k^T$. See `magma-X.Y.Z/src/sgebrd.cpp` for more details.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
#define min(a,b)  (((a)<(b))?(a):(b))
int main( int argc, char** argv){
  magma_init();                                  // initialize Magma
  magma_timestr_t start, end;
  float gpu_time, cpu_time;
  magma_int_t m = 8192, n = m, n2=m*n;
  float *a, *r;              // a,r - mxn matrices on the host
  float *h_work;                                       // workspace
  magma_int_t lhwork;                             // size of h_work
  float *taup, *tauq;   // arrays descr. elementary reflectors
  float *diag, *offdiag;      // bidiagonal form in two arrays
  magma_int_t i, info, minmn=min(m,n), nb;
  magma_int_t ione  = 1;
  magma_int_t ISEED[4] = {0,0,0,1};                       // seed
  nb  = magma_get_sgebrd_nb(n);           // optimal block size
  magma_smalloc_cpu(&a,m*n);               // host memory for a
  magma_smalloc_cpu(&tauq,minmn);       // host memory for tauq
  magma_smalloc_cpu(&taup,minmn);       // host memory for taup
  magma_smalloc_cpu(&diag,minmn);       // host memory for diag
  magma_smalloc_cpu(&offdiag,minmn-1);// host mem. for offdiag
  magma_smalloc_pinned(&r,m*n);            // host memory for r
  lhwork = (m + n)*nb;
  magma_smalloc_pinned(&h_work,lhwork);// host mem. for h_work
// Random  matrices
  lapackf77_slarnv( &ione, ISEED, &n2, a );
  lapackf77_slacpy(MagmaUpperLowerStr,&m,&n,a,&m,r,&m);// a->r
// MAGMA
```

```
  start = get_current_time();
// reduce the matrix a to upper bidiagonal form by orthogonal
// transformations: q^T*a*p, the obtained diagonal and the
// superdiagonal are written to diag and offdiag arrays resp.;
// the elements below the diagonal, represent the orthogonal
// matrix q as a product of elementary reflectors described
// by tauq; elements above the first superdiagonal  represent
// the orthogonal matrix p as a product of elementary reflect-
// ors described by taup;

  magma_sgebrd(m,n,r,m,diag,offdiag,tauq,taup,h_work,lhwork,
                                                    &info);

  end = get_current_time();
  gpu_time = GetTimerValue(start,end)/1e3;       // Magma time
  printf("sgebrd gpu time: %7.5f sec.\n",gpu_time);
// LAPACK
  start = get_current_time();
  lapackf77_sgebrd(&m,&n,a,&m,diag,offdiag,tauq,taup,h_work,
                                                &lhwork,&info);
  end = get_current_time();
  cpu_time = GetTimerValue(start,end)/1e3;      // Lapack time
  printf("sgebrd cpu time: %7.5f sec.\n",cpu_time);
// free memory
  free(a);                                    // free host memory
  free(tauq);                                 // free host memory
  free(taup);                                 // free host memory
  free(diag);                                 // free host memory
  magma_free_pinned(r);                       // free host memory
  magma_free_pinned(h_work);                  // free host memory
  magma_finalize( );                            // finalize Magma
  return EXIT_SUCCESS;
}
// sgebrd gpu time: 23.68982 sec.
// sgebrd cpu time: 52.67531 sec.
```

4.8.4 magma_dgebrd - reduce a real matrix to bidiagonal form by orthogonal transformations in double precision, CPU interface

This function reduces in double precision an $m \times n$ matrix A defined on the host to upper or lower bidiagonal form by orthogonal transformations:

$$Q^T A P = B,$$

where P, Q are orthogonal and B is bidiagonal. If $m \geq n$, B is upper bidiagonal; if $m < n$, B is lower bidiagonal. The obtained diagonal and the super/subdiagonal are written to diag and offdiag arrays respectively. If $m \geq n$, the elements below the diagonal, with the array *tauq* represent the orthogonal matrix Q as a product of elementary reflectors $H_k = I -$

$tauq_k \cdot v_k \cdot v_k^T$, and the elements above the first superdiagonal with the array *taup* represent the orthogonal matrix P as a product of elementary reflectors $G_k = I - taup_k \cdot u_k \cdot u_k^T$. See `magma-X.Y.Z/src/dgebrd.cpp` for more details.

```
#include <stdio.h>
#include <cuda.h>
#include "magma.h"
#include "magma_lapack.h"
#define min(a,b)  (((a)<(b))?(a):(b))
int main( int argc, char** argv){
  magma_init();                              // initialize Magma
  magma_timestr_t start, end;
  double gpu_time, cpu_time;
  magma_int_t m = 8192, n = m, n2=m*n;
  double *a, *r;             // a,r - mxn matrices on the host
  double *h_work;                                  // workspace
  magma_int_t lhwork;                          // size of h_work
  double *taup, *tauq;  // arrays descr. elementary reflectors
  double *diag, *offdiag;     // bidiagonal form in two arrays
  magma_int_t i, info, minmn=min(m,n), nb;
  magma_int_t ione  = 1;
  magma_int_t ISEED[4] = {0,0,0,1};                    // seed
  nb  = magma_get_dgebrd_nb(n);           // optimal block size
  magma_dmalloc_cpu(&a,m*n);               // host memory for a
  magma_dmalloc_cpu(&tauq,minmn);       // host memory for tauq
  magma_dmalloc_cpu(&taup,minmn);       // host memory for taup
  magma_dmalloc_cpu(&diag,minmn);       // host memory for diag
  magma_dmalloc_cpu(&offdiag,minmn-1);// host mem. for offdiag
  magma_dmalloc_pinned(&r,m*n);            // host memory for r
  lhwork = (m + n)*nb;
  magma_dmalloc_pinned(&h_work,lhwork);// host mem. for h_work
// Random  matrices
  lapackf77_dlarnv( &ione, ISEED, &n2, a );
  lapackf77_dlacpy(MagmaUpperLowerStr,&m,&n,a,&m,r,&m);// a->r
// MAGMA
  start = get_current_time();
// reduce the matrix a to upper bidiagonal form by orthogonal
// transformations: q^T*a*p, the obtained diagonal and the
// superdiagonal are written to diag and offdiag arrays resp.;
// the elements below the diagonal, represent the orthogonal
// matrix q as a product of elementary reflectors described
// by tauq; elements above the first superdiagonal  represent
// the orthogonal matrix p as a product of elementary reflect-
// ors described by taup;

  magma_dgebrd(m,n,r,m,diag,offdiag,tauq,taup,h_work,lhwork,
                                                    &info);

  end = get_current_time();
  gpu_time = GetTimerValue(start,end)/1e3;       // Magma time
```

```
  printf("dgebrd gpu time: %7.5f sec.\n",gpu_time);
// LAPACK
  start = get_current_time();
  lapackf77_dgebrd(&m,&n,a,&m,diag,offdiag,tauq,taup,h_work,
                                                  &lhwork,&info);
  end = get_current_time();
  cpu_time = GetTimerValue(start,end)/1e3;      // Lapack time
  printf("dgebrd cpu time: %7.5f sec.\n",cpu_time);
// free memory
  free(a);                                    // free host memory
  free(tauq);                                 // free host memory
  free(taup);                                 // free host memory
  free(diag);                                 // free host memory
  magma_free_pinned(r);                       // free host memory
  magma_free_pinned(h_work);                  // free host memory
  magma_finalize( );                            // finalize Magma
  return EXIT_SUCCESS;
}
// dgebrd gpu time: 34.17384 sec.
// dgebrd cpu time: 110.03189 sec.
```

Bibliography

[CUBLAS] *CUBLAS LIBRARY User Guide*, Nvidia, July 2013
http://docs.nvidia.com/cuda/cublas/index.html

[MAGMA] *MAGMA, Matrix Algebra on GPU and Multicore Architectures*,ICL, Univ. of Tenessee, August 2013
http://icl.cs.utk.edu/magma/docs

[ARRAYFIRE] Chrzęszczyk A., *Matrix Computations on the GPU with ArrayFire for C/C++*, Accelereyes, May 2012
http://www.accelereyes.com/support/whitepapers

[FUND] Watkins D. S., *Fundamentals of Matrix Computations, 2nd ed.*, John Willey & Sons, New York 2002

[MATR] Golub G. H, van Loan C. F., *Matrix Computations, 3rd ed.* Johns Hopkins Univ. Press, Baltimore 1996

[LAPACK] Anderson E., Bai Z., Bischof C., Blackford S., Demmel J., Dongarra J., et al *LAPACK Users' Guide, 3rd ed.*, August 1999
http://www.netlib.org/lapack/lug/