Matrix computations on the GPU

CUBLAS, CUSOLVER and MAGMA by example

Andrzej Chrzęszczyk
Jan Kochanowski University, Kielce, Poland
Jacob Anders
CSIRO, Canberra, Australia

Version 2017

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 three of them, CUBLAS, MAGMA and CUSOLVER 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 200 code samples. We believe that the presented document can be an useful addition to the existing documentation for CUBLAS, CUSOLVER and MAGMA.

Remarks on using unified memory.
Unified memory is a single memory address space which allows applications to allocate data, that can be read or written from code running on either CPU or GPU. Unification of memory spaces means that there is no need for explicit memory transfers between host and device. This makes the CUDA programming easier. Allocating unified memory is as simple as replacing calls to `malloc` or `cudaMalloc` with calls to `cudaMallocManaged`. When

code running on CPU or GPU accesses data allocated this way, the CUDA system takes care of migrating memory pages to the memory of the accessing processor. Let us note however, that a carefully tuned CUDA program that uses streams and `cudaMemcpyAsync` to efficiently overlap execution with data transfer may perform better than a CUDA program that only uses unified memory. Users of unified memory are still free to use `cudaMemcpy` or `cudaMemcpyAsync` for performance optimization. Additionally, aplications can guide the driver using `cudaMemAdvise` and explicitly migrate memory using `cudaMemPrefetchAsync`. Note also that unified memory examples, which do not call `cudaMemcpy`, require an explicit `cudaDeviceSynchronize` before the host program can safely use the output from the GPU. The memory allocated with `cudaMallocManaged` should be released with `cudaFree`.

Our main purpose is to show a set of examples containing matrix computations on GPUs which are *easy to understand*. On the other hand, the performance is the main reason for using GPUs in matrix computations. Therefore we have decided to present (almost) *all examples in two versions.* First we demonstrate a traditional version with explicit data copying between host and device. Next we give the same examples using the unified memory. The second approach is simpler and probably more appropriate for beginners. The users which need more control on the data will probably prefer the first one or will combine both.

Contents

Chapter 1

CUDA Toolkit

1.1 Installing CUDA Toolkit

All libraries described in our text: CUBLAS, CUSOLVER and MAGMA need CUDA (Compute Unified Device Architecture) environment. In fact CUBLAS and CUSOLVER are parts of CUDA. The environment can be downloaded from https://developer.nvidia.com/cuda-downloads. On the same page one can find complete documentation, including installation instructions for Windows, Linux and Mac OSX. At the time of writing, the current release was CUDA v8.0. In our examples we shall use Ubuntu 16.04. In this system one can install CUDA using the command

```
# apt-get install cuda
```

Let us remark however that `apt-get` changes the paths and makes the installation of other tools more difficult.
If the CUDA software is installed and configured correctly, and the CUDA code samples are copied to the `$HOME` directory, the executable:

```
$ ~/NVIDIA_CUDA-8.0_Samples/1_Utilities/deviceQuery/deviceQuery
```

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

```
$ ~/NVIDIA_CUDA-8.0_Samples/5_Simulations/nbody/
nbody -benchmark -numbodies=256000 -numdevices=1
# (in the case of one device)
```

gives the opportunity to check GPU performance. On GeForce GTX 1080 card, one can obtain for example

```
Compute 6.1 CUDA device: [GeForce GTX 1080]
number of bodies = 256000
256000 bodies, total time for 10 iterations: 2413.114 ms
= 271.583 billion interactions per second
= 5431.653 single-precision GFLOP/s at 20 flops per interaction
```

The state of devices can be checked using

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

Wed Jul 19 11:29:44 2017
+-----------------------------------------------------------------------------+
|NVIDIA-SMI 378.13                Driver Version: 378.13                      |
|-------------------------------+----------------------+----------------------+
|GPU  Name        Persistence-M|Bus-Id        Disp.A |Volatile Uncorr. ECC |
|Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage |GPU-Util  Compute M. |
|===============================+======================+======================|
|  0  GeForce GTX 1080    Off  |0000:01:00.0      On |                  N/A |
|24%   36C    P8    10W / 180W |    222MiB /  8110MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
|Processes:                                                       GPU Memory |
| GPU       PID  Type  Process name                               Usage      |
|=============================================================================|
|   0      1006    G   /usr/lib/xorg/Xorg                              131MiB |
|   0      1546    G   compiz                                           88MiB |
+-----------------------------------------------------------------------------+
```

1.2 Measuring GPUs performance

It seems that one of the simplest ways of benchmarking systems with GPU devices is to use the Magma library. The library will be introduced in one of the next chapters but now let us remark, that as an by-product of Magma installation, one obtains the directory `testing` with ready to use testing binaries. Bellow we present the results of running four of them. We tested a system with Linux Ubuntu 16.04 and

- Intel i7 6700K CPU, 16GB RAM
- Nvidia GeForce GTX 1080 card,
- `magma-2.2.0` compiled with `OpenBLAS`.

As we remarked such a system seems to be sufficient for training and single precision calculations on GPU, but GeForce GTX cards have strongly restricted double precision capabilities. As a consequence in the benchmarks we present, the double precision performance is not so impressive. Using professional cards, one can expect that the double precision functions are only two times slower than the corresponding single precision ones.

Solving the general NxN linear system in single precision.

```
./testing_sgesv --lapack
%   N  NRHS   CPU Gflop/s (sec)   GPU Gflop/s (sec)
%===================================================
 1088     1     75.89 (   0.01)     74.72 (   0.01)
 2112     1    181.11 (   0.03)    230.11 (   0.03)
 3136     1    208.44 (   0.10)    438.46 (   0.05)
 4160     1    227.21 (   0.21)    494.41 (   0.10)
 5184     1    240.69 (   0.39)    632.86 (   0.15)
 6208     1    250.69 (   0.64)    778.43 (   0.20)
 7232     1    266.17 (   0.95)    920.08 (   0.27)
 8256     1    273.73 (   1.37)   1072.90 (   0.35)
 9280     1    285.11 (   1.87)   1220.60 (   0.44)
```

Let us repeat that the executables used in this section are contained in `testing` subdirectory of Magma installation directory.

Solving the general NxN linear system in double precision.

```
./testing_dgesv --lapack
%   N  NRHS   CPU Gflop/s (sec)   GPU Gflop/s (sec)
%====================================================
 1088     1     86.97 (   0.01)     58.13 (   0.01)
 2112     1     92.03 (   0.07)    117.56 (   0.05)
 3136     1    102.69 (   0.20)    145.29 (   0.14)
 4160     1    111.11 (   0.43)    179.07 (   0.27)
 5184     1    120.00 (   0.77)    200.15 (   0.46)
 6208     1    126.85 (   1.26)    211.75 (   0.75)
 7232     1    132.66 (   1.90)    219.35 (   1.15)
 8256     1    137.11 (   2.74)    225.68 (   1.66)
 9280     1    142.41 (   3.74)    232.11 (   2.30)
10304     1    145.79 (   5.00)    236.96 (   3.08)
```

Matrix-matrix product in single precision.

```
./testing_sgemm --lapack
% transA = No transpose, transB = No transpose
%   M     N     K   MAGMA Gflop/s (ms)  cuBLAS Gflop/s (ms)   CPU Gflop/s (ms)
%==============================================================================
 1088  1088  1088   2985.30 (   0.86)    5377.70 (   0.48)    344.92 (   7.47)
 2112  2112  2112   4317.43 (   4.36)    6461.15 (   2.92)    317.30 (  59.38)
 3136  3136  3136   4507.27 (  13.68)    6807.34 (   9.06)    347.10 ( 177.71)
 4160  4160  4160   4531.90 (  31.77)    6897.06 (  20.88)    380.82 ( 378.08)
 5184  5184  5184   5212.77 (  53.45)    7501.88 (  37.14)    390.93 ( 712.74)
 6208  6208  6208   4896.58 (  97.72)    7407.86 (  64.59)    398.93 (1199.45)
 7232  7232  7232   5021.23 ( 150.66)    7600.73 (  99.53)    401.84 (1882.57)
 8256  8256  8256   4974.09 ( 226.27)    7444.21 ( 151.19)    414.80 (2713.30)
 9280  9280  9280   4931.62 ( 324.10)    7460.25 ( 214.25)    412.64 (3873.52)
10304 10304 10304   4820.21 ( 453.92)    7505.78 ( 291.51)    418.97 (5222.33)
```

Matrix-matrix product in double precision.

```
./testing_dgemm --lapack
% transA = No transpose, transB = No transpose
%   M     N     K   MAGMA Gflop/s (ms)  cuBLAS Gflop/s (ms)   CPU Gflop/s (ms)
%==============================================================================
 1088  1088  1088    213.21 (  12.08)     218.40 (  11.79)     72.51 (  35.52)
 2112  2112  2112    220.69 (  85.37)     231.89 (  81.25)    165.29 ( 113.99)
 3136  3136  3136    257.86 ( 239.21)     277.21 ( 222.51)    180.38 ( 341.96)
 4160  4160  4160    261.32 ( 550.98)     278.40 ( 517.18)    183.05 ( 786.56)
 5184  5184  5184    259.34 (1074.36)     278.91 ( 998.97)    190.42 (1463.25)
 6208  6208  6208    258.53 (1850.89)     279.77 (1710.36)    190.80 (2507.84)
 7232  7232  7232    260.83 (2900.31)     281.12 (2690.99)    194.81 (3883.22)
 8256  8256  8256    260.32 (4323.45)     279.53 (4026.29)    196.29 (5733.71)
 9280  9280  9280    258.77 (6176.75)     277.68 (5756.20)    197.17 (8106.52)
10304 10304 10304    257.44 (8499.15)     277.09 (7896.35)    196.48 (11135.99)
```

Chapter 2

CUBLAS by example

2.1 General remarks on the examples

CUBLAS is an abbreviation for CUDA Basic Linear Algebra Subprograms. In the file `/usr/local/cuda/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 rather 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.

Remarks on compilation. All examples in this chapter contain simple compilation instructions. Notice that the examples, in which we use the unified memory, have the names of the form `example.cu`, while the other ones have the form `example.c`. The simplest compilation method we know is respectively

```
nvcc example.c -lcublas
```

and

```
nvcc example.cu -lcublas}
```

If the extension `c` is preferred in the second case, then all occurrences of the function `cudaMallocManaged` should have the third argument (integer) 1.
If `g++` command is preferred, then the syntax of the form
`cudaMallocManaged((void**)&x,n*sizeof(float),1);` should be used instead of `cudaMallocManaged(&x,n*sizeof(float),1);`
the constant `EXIT_SUCCESS` should be replaced by 0, and the header `cuda_runtime_api.h` should be included.
An example of compilation with `g++`:

```
g++ 001isamaxu.c -I/usr/local/cuda-8.0/include
-L/usr/local/cuda/lib64 -lcuda -lcublas -lcudart
```

2.2 CUBLAS Level-1. Scalar and vector based operations

2.2.1 `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
```

2.2.2 `cublasIsamax, cublasIsamin` - unified memory version

```
// nvcc 001isamax.cu -lcublas
#include <stdio.h>
#include "cublas_v2.h"
#define n 6                                         // length of x
int main(void){
  cublasHandle_t handle;                           // CUBLAS context
  int j;                                        // index of elements
  float* x;                                              // n-vector
  cudaMallocManaged(&x,n*sizeof(float)); // unified mem.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("%4.0f,",x[j]);                                // print x
  printf("\n");
  cublasCreate(&handle);         // initialize CUBLAS context
  int result;          // index of the maximal/minimal element
// find the smallest index of the element of x   with maximal
// absolute value

  cublasIsamax(handle,n,x,1,&result);

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

  cublasIsamin(handle,n,x,1,&result);

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

2.2.3 `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
```

2.2.4 cublasSasum - unified memory version

```
//nvcc 003sasum.cu -lcublas
#include <stdio.h>
#include "cublas_v2.h"
#define n 6                                        // length of x
int main(void){
  cublasHandle_t handle;                           // CUBLAS context
  int j;                                         // index of elements
  float* x;                                    // n-vector on the host
  cudaMallocManaged(&x,n*sizeof(float)); // unified mem.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");
  cublasCreate(&handle);          // initialize CUBLAS context
  float result;
// add absolute values of elements of the array x:
                                             // |x[0]|+...+|x[n-1]|

  cublasSasum(handle,n,x,1,&result);

  cudaDeviceSynchronize();
//print the result
  printf("sum of the absolute values of elements of x:%4.0f\n",
                                                          result);
  cudaFree(x);                                   // free  memory
  cublasDestroy(handle);             // destroy CUBLAS context
  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
```

2.2.5 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}
```

2.2.6 cublasSaxpy - unified memory version

```
//nvcc 004saxpy.cu -lcublas
#include <stdio.h>
#include "cublas_v2.h"
#define n 6                                     // length of x,y
int main(void){
  cublasHandle_t handle;                      // CUBLAS context
  int j;                                  // index of elements
  float* x;                                         // n-vector
  float* y;                                         // n-vector
  cudaMallocManaged(&x,n*sizeof(float)); // unified mem.for x
  for(j=0;j<n;j++)
    x[j]=(float)j;                         // x={0,1,2,3,4,5}
  cudaMallocManaged(&y,n*sizeof(float)); // unified mem.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");
  cublasCreate(&handle);         // initialize CUBLAS context
  float al=2.0;                                        // al=2
// multiply the vector x by the scalar al and add to  y
// y = al*x + y,   x,y - n-vectors; al - scalar

  cublasSaxpy(handle,n,&al,x,1,y,1);

  cudaDeviceSynchronize();
  printf("y after Saxpy:\n");          // print y after Saxpy
  for(j=0;j<n;j++)
    printf("%2.0f,",y[j]);
  printf("\n");
  cudaFree(x);                                // free  memory
  cudaFree(y);                                // free  memory
  cublasDestroy(handle);           // destroy CUBLAS context
  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}
```

2.2.7 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,
```

2.2.8 cublasScopy - unified memory version

```
//nvcc 005scopy.cu -lcublas
#include <stdio.h>
#include "cublas_v2.h"
#define n 6                                       // length of x,y
int main(void){
  cublasHandle_t handle;                       // CUBLAS context
  int j;                                     // index of elements
  float* x;                                             // n-vector
  float* y;                                             // n-vector
  cudaMallocManaged((void**)&x,n*sizeof(float));//u.mem 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");
  cudaMallocManaged((void**)&y,n*sizeof(float));//u.mem for y
  cublasCreate(&handle);          // initialize CUBLAS context
// copy the vector x into y:  x -> y

  cublasScopy(handle,n,x,1,y,1);

  cudaDeviceSynchronize();
  printf("y after copy:\n");
  for(j=0;j<n;j++)
    printf("%2.0f,",y[j]);                              // print y
  printf("\n");
  cudaFree(x);                                    // free  memory
  cudaFree(y);                                    // free  memory
  cublasDestroy(handle);            // destroy CUBLAS context
  return 0;
}
// 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,
```

2.2.9 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
```

2.2.10 cublasSdot - unified memory version

```
//nvcc 006sdot.cu -lcublas
#include <stdio.h>
#include "cublas_v2.h"
#define n 6                                     // length of x,y
int main(void){
  cublasHandle_t handle;
  int j;                                   // index of elements
  float* x;                                          // n-vector
  float* y;                                          // n-vector
  cudaMallocManaged(&x,n*sizeof(float)); // unified mem.for x
  for(j=0;j<n;j++)
    x[j]=(float)j;                           // x={0,1,2,3,4,5}
  cudaMallocManaged(&y,n*sizeof(float)); // unified mem.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");
  cublasCreate(&handle);          // initialize CUBLAS context
  float result;
// dot prod. of two vectors x,y:  x[0]*y[0]+...+x[n-1]*y[n-1]

  cublasSdot(handle,n,x,1,y,1,&result);

  cudaDeviceSynchronize();
  printf("dot product x.y:\n");
  printf("%7.0f\n",result);                   // print the result
  cudaFree(x);                                    // free  memory
  cudaFree(y);                                    // free  memory
  cublasDestroy(handle);            // destroy CUBLAS context
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
```

2.2.11 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}
```

2.2.12 cublasSnrm2 - unified memory version

```
//nvcc 007snrm2.cu -lcublas
#include <stdio.h>
#include "cublas_v2.h"
#define n 6                                        // length of x
int main(void){
  cublasHandle_t handle;                        // CUBLAS context
  int j;                                     // index of elements
  float* x;                                            // n-vector
  cudaMallocManaged((void**)&x,n*sizeof(float));   // unified
  for(j=0;j<n;j++)                                 // memory for x
    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");
  cublasCreate(&handle);         // initialize CUBLAS context
  float result;
// Euclidean norm of the vector x: \sqrt{x[0]^2+...+x[n-1]^2}

  cublasSnrm2(handle,n,x,1,&result);

  cudaDeviceSynchronize();
  printf("Euclidean norm of  x: ");
  printf("%7.3f\n",result);                    // print the result
  cudaFree(x);                                     // free  memory
  cublasDestroy(handle);                 // destroy CUBLAS context
  return 0;
}
// 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}
```

2.2.13 `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]
```

2.2.14 cublasSrot - unified memory version

```
// nvcc 008srot.cu -lcublas
#include <stdio.h>
#include "cublas_v2.h"
#define n 6                                          // length of x,y
int main(void){
  cublasHandle_t handle;                              // CUBLAS context
  int j;                                           // index of elements
  float* x;                                                 // n-vector
  float* y;                                                 // n-vector
  cudaMallocManaged(&x,n*sizeof(float)); // unified mem.for x
  for(j=0;j<n;j++)
    x[j]=(float)j;                                 // x={0,1,2,3,4,5}
  cudaMallocManaged(&y,n*sizeof(float)); // unified mem.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");
  cublasCreate(&handle);         // initialize CUBLAS context
  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]

  cublasSrot(handle,n,x,1,y,1,&c,&s);

  cudaDeviceSynchronize();
  printf("x after Srot:\n");              // print x after Srot
  for(j=0;j<n;j++)
    printf("%7.3f,",x[j]);
  printf("\n");

  printf("y after Srot:\n");              // print y after Srot
  for(j=0;j<n;j++)
    printf("%7.3f,",y[j]);
  printf("\n");
  cudaFree(x);                                         // free  memory
  cudaFree(y);                                         // free  memory
  cublasDestroy(handle);                  // destroy CUBLAS context
}
```

```
// 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]
```

2.2.15 `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 second entry of 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   ]
```

2.2.16 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 d_x and d_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]
```

2.2.17 `cublasSrotm` - unified memory version

```
// nvcc 010srotmVec.cu  -lcublas
#include <stdio.h>
#include "cublas_v2.h"
#define n 6                                     // length of x,y
int main(void){
  cublasHandle_t handle;                        // CUBLAS context
  int j;                                    // index of elements
  float* x;                                          // n-vector
  float* y;                                          // n-vector
  float* param;
  cudaMallocManaged((void**)&x,n*sizeof(float));   // unified
  for(j=0;j<n;j++)                               // memory for x
    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");
  cudaMallocManaged((void**)&y,n*sizeof(float));   // unified
  for(j=0;j<n;j++)                               // memory for y
    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]

  cublasCreate(&handle);           // initialize CUBLAS context
//                                                  [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 ]

  cublasSrotm(handle,n,x,1,y,1,param);

  cudaDeviceSynchronize();
  printf("x after Srotm x:\n");           // print x after Srotm
  for(j=0;j<n;j++)
    printf("%7.3f,",x[j]);
  printf("\n");
  printf("y after Srotm y:\n");        // print y after Srotm
  for(j=0;j<n;j++)
    printf("%7.3f,",y[j]);
  printf("\n");
  cudaFree(x);                                  // free  memory
  cudaFree(y);                                  // free  memory
  cublasDestroy(handle);              // destroy CUBLAS context
  free(param);                                  // free  memory
  return 0;
}
// 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]
```

2.2.18 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
```

2.2.19 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}
```

2.2.20 cublasSscal - unified memory version

```
// nvcc 012sscal.cu -lcublas
#include <stdio.h>
#include "cublas_v2.h"
#define n 6                                       // length of x
int main(void){
  cublasHandle_t handle;                      // CUBLAS context
  int j;                                    // index of elements
  float* x;                                          // n-vector
  cudaMallocManaged(&x,n*sizeof(float)); // unified mem.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");
  cublasCreate(&handle);          // initialize CUBLAS context
  float al=2.0;                                          // al=2
// scale the vector x by the scalar al: x = al*x

  cublasSscal(handle,n,&al,x,1);

  cudaDeviceSynchronize();
  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(x);                                  // free  memory
  cublasDestroy(handle);             // destroy CUBLAS context
  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}
```

2.2.21 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
```

2.2.22 cublasSswap - unified memory version

```
// nvcc 013sswap.cu -lcublas
#include <stdio.h>
#include "cublas_v2.h"
#define n 6                                        // length of x,y
int main(void){
  cublasHandle_t handle;                            // CUBLAS context
  int j;                                         // index of elements
  float* x;                                               // n-vector
  float* y;                                               // n-vector
  cudaMallocManaged(&x,n*sizeof(float)); // unified mem.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");
  cudaMallocManaged(&y,n*sizeof(float)); // unified mem.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");
  cublasCreate(&handle);          // initialize CUBLAS context
// swap x and y

  cublasSswap(handle,n,x,1,y,1);

  cudaDeviceSynchronize();
  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(x);                                        // free  memory
  cudaFree(y);                                        // free  memory
  cublasDestroy(handle);             // destroy CUBLAS context
  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
```

2.3 CUBLAS Level-2. Matrix-vector operations

2.3.1 `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]
```

2.3.2 cublasSgbmv – unified memory version

```
// nvcc 013sgbmv.cu -lcublas
#include <stdio.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){
  cublasHandle_t handle;                         // CUBLAS context
  int i,j;                                 // row and column index
// declaration and allocation of a,x,y in unified memory
  float* a; //mxn matrix                    // a:
  float* x; //n-vector                      //   20 15 11
  float* y; //m-vector                      //   25 21 16 12
                                            //      26 22 17 13
                                            //         27 23 18 14
                                            //            28 24 19
  cudaMallocManaged(&a,m*n*sizeof(float));//unified mem.for a
  cudaMallocManaged(&x,n*sizeof(float));  //unified mem.for x
  cudaMallocManaged(&y,m*sizeof(float));  //unified mem.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
  cublasCreate(&handle);
  float al=1.0f;                                          // al=1
  float bet=1.0f;                                        // bet=1
// banded matrix-vector multiplication:
//  y = al*a*x + bet*y;      a - mxn banded matrix;
//  x - n-vector, y - m-vector; al,bet - scalars

 cublasSgbmv(handle,CUBLAS_OP_N,m,n,kl,ku,&al,a,m,x,1,&bet,y,1);

  cudaDeviceSynchronize();
  printf("y after Sgbmv:\n");          // print y after Sgbmv
  for(j=0;j<m;j++){
      printf("%7.0f",y[j]);
      printf("\n");
  }
  cudaFree(a);                                     // free  memory
  cudaFree(x);                                      // free  memory
  cudaFree(y);                                      // free  memory
  cublasDestroy(handle);                  // destroy CUBLAS context
  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]
```

2.3.3 `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=0.0f;                                      // 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]
```

2.3.4 cublasSgemv – unified memory version

```
// nvcc 014sgemv.cu -lcublas
#include <stdio.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){
  cublasHandle_t handle;                            // CUBLAS context
  int i,j;                         // i-row index, j-column index
  float* a;                                          // a -mxn matrix
  float* x;                                           // x - n-vector
  float* y;                                           // y - m-vector
  cudaMallocManaged(&a,m*n*sizeof(float));//unified mem.for a
  cudaMallocManaged(&x,n*sizeof(float));  //unified mem.for x
  cudaMallocManaged(&y,m*sizeof(float));  //unified mem.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
  cublasCreate(&handle);
  float al=1.0f;                                             // al=1
  float bet=0.0f;                                           // bet=1
// matrix-vector multiplication:   y = al*a*x + bet*y
// a - mxn matrix; x - n-vector, y - m-vector;
// al,bet - scalars
```

```
  cublasSgemv(handle,CUBLAS_OP_N,m,n,&al,a,m,x,1,&bet,y,1);

  cudaDeviceSynchronize();
  printf("y after Sgemv::\n");
  for(j=0;j<m;j++){
      printf("%5.0f",y[j]);            // print y after Sgemv
      printf("\n");
  }
  cudaFree(a);                                  // free  memory
  cudaFree(x);                                  // free  memory
  cudaFree(y);                                  // free  memory
  cublasDestroy(handle);             // destroy CUBLAS context
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]
```

2.3.5 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]
```

2.3.6 cublasSger - unified memory version

```
// nvcc 015sger.cu -lcublas
#include <stdio.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){
  cublasHandle_t handle;                     // CUBLAS context
  int i,j;                      // i-row index, j-column index
  float* a;                                  // a -mxn matrix
  float* x;                                   // x - n-vector
  float* y;                                   // y - m-vector
  cudaMallocManaged(&a,m*n*sizeof(float));//unified mem.for a
  cudaMallocManaged(&x,n*sizeof(float));  //unified mem.for x
  cudaMallocManaged(&y,m*sizeof(float));  //unified mem.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
  cublasCreate(&handle);          // initialize CUBLAS context
  float al=2.0f;                                       // al=2
// rank-1 update of a:  a = al*x*y^T + a
// a - mxn matrix;  x -m-vector, y - n-vector; al -scalar

  cublasSger(handle,m,n,&al,x,1,y,1,a,m);

  cudaDeviceSynchronize();

// 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(a);                                  // free  memory
  cudaFree(x);                                  // free  memory
  cudaFree(y);                                  // free  memory
  cublasDestroy(handle);            // destroy CUBLAS context
  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]
```

2.3.7 `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]
```

2.3.8 cublasSsbmv - unified memory version

```
// nvcc 016ssbmv.cu -lcublas
#include <stdio.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){
  cublasHandle_t handle;                        // CUBLAS context
  int i,j;                              // row index, column index
  float *a; // nxn matrix a                  //lower triangle of a:
  float *x; // n-vector x                      //11
  float *y; // n-vector y                      //17,12
  cudaMallocManaged(&a,n*n*sizeof(float));//   18,13
// unified memory for a                        //      19,14
  cudaMallocManaged(&x,n*sizeof(float));  //         20,15
// unified memory for x                        //            21,16
  cudaMallocManaged(&y,n*sizeof(float));  //unified mem.for y
// 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
  cublasCreate(&handle);          // initialize CUBLAS context
  float al=1.0f;                                      // al=1
  float bet=1.0f;                                    // bet=1
// symmetric banded matrix-vector multiplication:
// y = al*a*x + bet*y,
// a - nxn symmetric banded matrix;
// x,y - n-vectors; al,bet - scalars

  cublasSsbmv(handle,CUBLAS_FILL_MODE_LOWER,n,k,&al,a,n,x,1,
                                                  &bet,y,1);

  cudaDeviceSynchronize();
  printf("y after Ssbmv:\n");
  for(j=0;j<n;j++){
      printf("%7.0f",y[j]);            // print y after Ssbmv
      printf("\n");
  }
  cudaFree(a);                                      // free  memory
  cudaFree(x);                                      // free  memory
  cudaFree(y);                                      // free  memory
  cublasDestroy(handle);                   // destroy CUBLAS context
}
// 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]
```

2.3.9 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]
```

2.3.10 cublasSspmv - unified memory version

```
// nvcc 017sspmv.cu -lcublas
#include <stdio.h>
#include "cublas_v2.h"
#define n 6                    // number of rows and columns of a
int main(void){
  cublasHandle_t handle;                          // CUBLAS context
  int i,j,l,m; // indices                      // a:
  float *a; //lower triangle of nxn            // 11
            // matrix a                        // 12,17
  float *x; // n-vector x                      // 13,18,22
  float *y; // n-vector y                      // 14,19,23,26
                                               // 15,20,24,27,29
// unified memory for a                        // 16,21,25,28,30,31
  cudaMallocManaged(&a,n*(n+1)/2*sizeof(float));
  cudaMallocManaged(&x,n*sizeof(float)); // unified mem.for x
  cudaMallocManaged(&y,n*sizeof(float)); // unified mem.for y
//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
  cublasCreate(&handle);          // initialize CUBLAS context
  float al=1.0f;                                        // al=1
  float bet=1.0f;                                      // bet=1
// symmetric packed matrix-vector multiplication:
// y = al*a*x + bet*y; a -symmetric nxn matrix
// in packed format; x,y - n-vectors;  al,bet - scalars

  cublasSspmv(handle,CUBLAS_FILL_MODE_LOWER,n,&al,a,x,1,&bet,y,1);

  cudaDeviceSynchronize();
  printf("y after Sspmv:\n");           // print y after Sspmv
  for(j=0;j<n;j++){
      printf("%7.0f",y[j]);
      printf("\n");
  }
  cudaFree(a);                                    // free  memory
  cudaFree(x);                                    // free  memory
  cudaFree(y);                                    // free  memory
  cublasDestroy(handle);               // destroy CUBLAS context
  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]
```

2.3.11 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]
```

2.3.12 cublasSspr - unified memory version

```
// nvcc 018sspr.cu -lcublas
#include <stdio.h>
#include "cublas_v2.h"
#define n 6                  // number of rows and columns of a
int main(void){
  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
                                              //13,18,22
                                              //14,19,23,26
                                              //15,20,24,27,29
// unified memory for a                       //16,21,25,28,30,31
  cudaMallocManaged(&a,n*(n+1)/2*sizeof(float));
  cudaMallocManaged(&x,n*sizeof(float)); //unified mem. for x
//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
  cublasCreate(&handle);         // initialize CUBLAS context
  float al=1.0f;                                     // al=1
// rank-1 update of a symmetric matrix a :
// a = al*x*x^T + a
// a - symmetric nxn matrix in packed format; x n-vector;
// al - scalar

  cublasSspr(handle,CUBLAS_FILL_MODE_LOWER,n,&al,x,1,a);

  cudaDeviceSynchronize();
  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(a);                                 // free  memory
  cudaFree(x);                                 // free  memory
  cublasDestroy(handle);           // destroy CUBLAS context
  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]
```

2.3.13 `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 019sspr2.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 an 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; d_x,d_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]
```

2.3.14 cublasSspr2 - unified memory version

```
// nvcc 019ssspr2.cu -lcublas
#include <stdio.h>
#include "cublas_v2.h"
#define n 6                    // number of rows and columns of a
int main(void){
  cublasHandle_t handle;                      // CUBLAS context
  int i,j,l,m;                                       // indices
  float *a;                    // lower triangle of an nxn matrix
  float *x; // n-vector                    // a:
  float *y; // n-vector                    // 11
                                           // 12,17
                                           // 13,18,22
                                           // 14,19,23,26
                                           // 15,20,24,27,29
                                           // 16,21,25,28,30,31
// unified memory for a
  cudaMallocManaged(&a,n*(n+1)/2*sizeof(float));
  cudaMallocManaged(&x,n*sizeof(float)); //unified mem. for x
  cudaMallocManaged(&y,n*sizeof(float)); //unified mem. for y
//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
  cublasCreate(&handle);          // initialize CUBLAS context
  float al=1.0f;                                     // al=1.0
// rank-2 update of symmetric matrix a :
// a = al*(x*y^T + y*x^T) + a
// a - symmetric  nxn matrix in packed form; x,y - n-vect.;
// al - scalar

  cublasSspr2(handle,CUBLAS_FILL_MODE_LOWER,n,&al,x,1,y,1,a);

  cudaDeviceSynchronize();
// 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(a);                                   // free  memory
  cudaFree(x);                                   // free  memory
  cudaFree(y);                                   // free  memory
  cublasDestroy(handle);              // destroy CUBLAS context
  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]
```

2.3.15 `cublasSsymv` - symmetric matrix-vector multiplication

This function performs the symmetric matrix-vector multiplication.

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

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]
```

2.3.16 cublasSsymv - unified memory version

```
// nvcc 020ssymv.cu -lcublas
#include <stdio.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){
  cublasHandle_t handle;                     // CUBLAS context
  int i,j;                   // i - row index, j - column index
  float* a;                                       // nxn matrix
  float* x;                                         // n-vector
  float* y;                                          // n-vector
  cudaMallocManaged(&a,n*n*sizeof(float));//unif.memory for a
  cudaMallocManaged(&x,n*sizeof(float));  //unif.memory for x
  cudaMallocManaged(&y,n*sizeof(float));  //unif.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
  cublasCreate(&handle);         // initialize CUBLAS context
  float al=1.0f;                                      // al=1.0
  float bet=1.0f;                                    // bet=1.0
// symmetric matrix-vector multiplication:
// y = al*a*x + bet*y
// a - nxn symmetric matrix; x,y - n-vectors;
// al,bet - scalars

  cublasSsymv(handle,CUBLAS_FILL_MODE_LOWER,n,&al,a,n,x,1,&bet,
                                                        y,1);
  cudaDeviceSynchronize();
  printf("y after Ssymv:\n");         // print y after Ssymv
  for(j=0;j<n;j++)
  printf("%7.0f\n",y[j]);
  cudaFree(a);                                 // free  memory
  cudaFree(x);                                 // free  memory
  cudaFree(y);                                 // free  memory
  cublasDestroy(handle);           // destroy CUBLAS context
  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]
```

2.3.17 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]
```

2.3.18 cublasSsyr - unified memory version

```
// nvcc 021ssyr.cu -lcublas
#include <stdio.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){
  cublasHandle_t handle;                          // CUBLAS context
  int i,j;                   // i - row index, j - column index
  float* a;                                           // nxn matrix
  float* x;                                             // n-vector
  cudaMallocManaged(&a,n*n*sizeof(float));//unif.memory for a
  cudaMallocManaged(&x,n*sizeof(float));  //unif.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
  cublasCreate(&handle);           // initialize CUBLAS context
  float al=1.0f;                                      // al=1.0
// symmetric rank-1 update of a:  a = al*x*x^T  + a
// a - nxn symmetric matrix; x - n-vector; al - scalar

  cublasSsyr(handle,CUBLAS_FILL_MODE_LOWER,n,&al,x,1,a,n);
  cudaDeviceSynchronize();
// 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(a);                                  // free  memory
  cudaFree(x);                                  // free  memory
  cublasDestroy(handle);              // destroy CUBLAS context
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]
```

2.3.19 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]
```

2.3.20 cublasSsyr2 - unified memory version

```
// nvcc 022ssyr2.cu -lcublas
#include <stdio.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){
  cublasHandle_t handle;                       // CUBLAS context
  int i,j;                  // i - row index, j - column index
  float* a;                                        // nxn matrix
  float* x;                                          // n-vector
  float* y;                                          // n-vector
  cudaMallocManaged(&a,n*n*sizeof(float));//unif.memory for a
  cudaMallocManaged(&x,n*sizeof(float));  //unif.memory for x
  cudaMallocManaged(&y,n*sizeof(float));  //unif.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

  cublasCreate(&handle);          // initialize CUBLAS context
  float al=1.0f;                                       // al=1
// symmetric rank-2 update of a:
// a = al*(x*y^T + y*x^T) + a
// a - nxn symmetric matrix; x,y - n-vectors; al -scalar

  cublasSsyr2(handle,CUBLAS_FILL_MODE_LOWER,n,&al,x,1,y,1,a,n);

  cudaDeviceSynchronize();
// 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(a);                                  // free  memory
  cudaFree(x);                                  // free  memory
  cudaFree(y);                              // free  memory
  cublasDestroy(handle);           // destroy CUBLAS context
  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]
```

2.3.21 `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]
```

2.3.22 cublasStbmv - unified memory version

```
// nvcc 023stbmv.cu -lcublas
#include <stdio.h>
#include "cublas_v2.h"
#define n 6                  // number of rows and columns of a
#define k 1                           // number of subdiagonals
int main(void){
  cublasHandle_t handle;                        // CUBLAS context
  int i,j;                                // lower triangle of a:
  float *a;             //nxn matrix a    //  11
  float *x;             //n-vector x      //  17,12
                                          //     18,13
                                          //        19,14
                                          //           20,15
                                          //              21,16
  cudaMallocManaged(&a,n*n*sizeof(float));//unif.memory for a
  cudaMallocManaged(&x,n*sizeof(float));  //unif.memory for x
// 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
  cublasCreate(&handle);          // initialize CUBLAS context
// triangular banded matrix-vector multiplication:  x = a*x;
// a - nxn lower triangular banded matrix; x - n-vector

  cublasStbmv(handle,CUBLAS_FILL_MODE_LOWER,CUBLAS_OP_N,
                            CUBLAS_DIAG_NON_UNIT,n,k,a,n,x,1);

  cudaDeviceSynchronize();
  printf("x after Stbmv :\n");          // print x after Stbmv
  for(j=0;j<n;j++){
      printf("%7.0f",x[j]);
      printf("\n");
  }
  cudaFree(a);                                   // free   memory
  cudaFree(x);                                   // free   memory
  cublasDestroy(handle);              // destroy CUBLAS context
  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]
```

2.3.23 `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*X=d_x;
// the solution 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]
```

2.3.24 cublasStbsv - unified memory version

```
// nvcc 024stbsv.cu -lcublas
#include <stdio.h>
#include "cublas_v2.h"
#define n 6                   // number of rows and columns of a
#define k 1                         // number of subdiagonals
int main(void){
  cublasHandle_t handle;                      // CUBLAS context
  int i,j;                               // lower triangle of a:
  float *a; //nxn matrix a                  // 11
  float *x; //n-vector x                    // 17,12
                                            //    18,13
                                            //       19,14
                                            //          20,15
                                            //             21,16
  cudaMallocManaged(&a,n*n*sizeof(float));//unif.memory for a
  cudaMallocManaged(&x,n*sizeof(float));  //unif.memory for x
//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
  cublasCreate(&handle);         // initialize CUBLAS context
// solve a triangular banded linear system:   a*X=x;
// the solution X  overwrites the right hand side x;
// a - nxn banded lower triangular matrix; x - n-vector

  cublasStbsv(handle,CUBLAS_FILL_MODE_LOWER,CUBLAS_OP_N,
                           CUBLAS_DIAG_NON_UNIT,n,k,a,n,x,1);

  cudaDeviceSynchronize();
// print the solution
  printf("solution :\n");                  // print x after Stbsv
  for(j=0;j<n;j++){
      printf("%9.6f",x[j]);
      printf("\n");
  }
  cudaFree(a);                                      // free  memory
  cudaFree(x);                                      // free  memory
  cublasDestroy(handle);               // destroy CUBLAS context
  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]
```

2.3.25 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 an 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]
```

2.3.26 cublasStpmv - unified memory version

```
// nvcc 025stpmv.cu -lcublas
#include <stdio.h>
#include "cublas_v2.h"
#define n 6                   // number of rows and columns of a
int main(void){
  cublasHandle_t handle;                      // CUBLAS context
  int i,j;
  float* a;                  // lower triangle of an nxn matrix
  float* x;                                         //  n-vector
//unified memory for a
  cudaMallocManaged(&a,n*(n+1)/2*sizeof(float));
  cudaMallocManaged(&x,n*sizeof(float)); //unif. memory 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
                                             //15,20,24,27,29
                                             //16,21,25,28,30,31
  cublasCreate(&handle);         // initialize CUBLAS context
// triangular packed matrix-vector multiplication:
// x = a*x;  a - nxn lower triangular matrix
// in packed format;  x - n-vector

  cublasStpmv(handle,CUBLAS_FILL_MODE_LOWER,CUBLAS_OP_N,
                                CUBLAS_DIAG_NON_UNIT,n,a,x,1);
  cudaDeviceSynchronize();
  printf("x after Stpmv :\n");          // print x after Stpmv
  for(j=0;j<n;j++){
      printf("%7.0f",x[j]);
      printf("\n");
  }
  cudaFree(a);                                   // free memory
  cudaFree(x);                                   // free memory
  cublasDestroy(handle);              // destroy CUBLAS context
  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]
```

2.3.27 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*X=d_x,
// the solution 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]
```

2.3.28 cublasStpsv - unified memory version

```
// nvcc 026stpsv.cu -lcublas
#include <stdio.h>
#include "cublas_v2.h"
```

```
#define n 6                  // number of rows and columns of a
int main(void){
  cublasHandle_t handle;                       // CUBLAS context
  int i,j;                        // i-row index, j-column index
  float* a;                                        // nxn matrix a
  float* x;                                          // n-vector x
// unified memory for a
  cudaMallocManaged(&a,n*(n+1)/2*sizeof(float));
  cudaMallocManaged(&x,n*sizeof(float)); //unif. memory 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
                                          //15,20,24,27,29
                                          //16,21,25,28,30,31
  cublasCreate(&handle);          // initialize CUBLAS context
// solve the packed triangular linear system:  a*X=x,
// the solution  X overwrites the right hand side x
// a - nxn lower triang. matrix in packed form; x - n-vector

  cublasStpsv(handle,CUBLAS_FILL_MODE_LOWER,CUBLAS_OP_N,
                             CUBLAS_DIAG_NON_UNIT,n,a,x,1);

  cudaDeviceSynchronize();
  printf("solution :\n");              // print x after Stpsv
  for(j=0;j<n;j++){
      printf("%9.6f",x[j]);
      printf("\n");
  }
  cudaFree(a);                                    // free  memory
  cudaFree(x);                                    // free  memory
  cublasDestroy(handle);             // destroy CUBLAS context
  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]
```

2.3.29 `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]
```

2.3.30 `cublasStrmv` - unified memory version

```
// nvcc 027strmv.cu -lcublas
#include <stdio.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){
  cublasHandle_t handle;                        // CUBLAS context
  int i,j;                        // i-row index, j-column index
  float* a;                                       // nxn matrix a
  float* x;                                         // n-vector x
  cudaMallocManaged(&a,n*n*sizeof(float));//unif.memory for a
  cudaMallocManaged(&x,n*sizeof(float));  //unif.memory 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
  cublasCreate(&handle);          // initialize CUBLAS context
// triangular matrix-vector multiplication:  x = a*x
// a - triangular nxn matrix in lower mode; x - n-vector

  cublasStrmv(handle,CUBLAS_FILL_MODE_LOWER,CUBLAS_OP_N,
                           CUBLAS_DIAG_NON_UNIT,n,a,n,x,1);
  cudaDeviceSynchronize();
  printf("multiplication result :\n"); // print x after Strmv
  for(j=0;j<n;j++){
      printf("%7.0f",x[j]);
      printf("\n");
  }
  cudaFree(a);                                        // free  memory
  cudaFree(x);                                        // free  memory
  cublasDestroy(handle);                   // destroy CUBLAS context
  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]
```

2.3.31 `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]
```

2.3.32 cublasStrsv - unified memory version

```
// nvcc 028strsv.cu -lcublas
#include <stdio.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){
  cublasHandle_t handle;                          // CUBLAS context
  int i,j;                            // i-row index, j-column index
  float* a;                                            // nxn matrix
  float* x;                                              // n-vector
  cudaMallocManaged(&a,n*n*sizeof(float));//unif.memory for a
  cudaMallocManaged(&x,n*sizeof(float)); //unif.memory  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
  cublasCreate(&handle);          // initialize CUBLAS context
// solve the triangular linear system:  a*X=x,
// the solution X overwrites the right hand side x,
// a - nxn triangular matrix in lower mode; x - n-vector

  cublasStrsv(handle,CUBLAS_FILL_MODE_LOWER,CUBLAS_OP_N,
                          CUBLAS_DIAG_NON_UNIT,n,a,n,x,1);

  cudaDeviceSynchronize();
  printf("solution :\n");                    // print x after Strsv
  for(j=0;j<n;j++){
      printf("%9.6f",x[j]);
      printf("\n");
  }
  cudaFree(a);                                     // free  memory
  cudaFree(x);                                     // free  memory
  cublasDestroy(handle);              // destroy CUBLAS context
  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]
```

2.3.33 `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]
```

2.3.34 cublasChemv - unified memory version

```
// nvcc 029Chemv.cu -lcublas
#include <stdio.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){
  cublasHandle_t handle;                       // CUBLAS context
  int i,j;                         // i-row index, j-column index
  cuComplex *a;                             // complex nxn matrix
  cuComplex *x;                               // complex n-vector
  cuComplex *y;                               // complex n-vector
  cudaMallocManaged(&a,n*n*sizeof(cuComplex));//unif.memory a
  cudaMallocManaged(&x,n*sizeof(cuComplex));  //unif.memory x
  cudaMallocManaged(&y,n*sizeof(cuComplex));  //unif.memory 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
  cublasCreate(&handle);        // initialize CUBLAS context
  cuComplex al={1.0f,0.0f};                          // al=1
  cuComplex bet={1.0f,0.0f};                        // bet=1
// Hermitian matrix-vector multiplication:
// y=al*a*x + bet*y
// a - nxn Hermitian matrix; x,y - n-vectors;
// al,bet -scalars

  cublasChemv(handle,CUBLAS_FILL_MODE_LOWER,n,&al,a,n,x,1,&bet,
                                                          y,1);

  cudaDeviceSynchronize();
  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(a);                                    // free  memory
  cudaFree(x);                                    // free  memory
  cudaFree(y);                                    // free  memory
  cublasDestroy(handle);              // destroy CUBLAS context
  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]
```

2.3.35 `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]
```

2.3.36 cublasChbmv - unified memory version

```
// nvcc 030Chbmv.cu -lcublas
#include <stdio.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){
  cublasHandle_t handle;                          // CUBLAS context
  int i,j;                             // i-row index, j-column index
                                                // lower triangle of a:
  cuComplex *a; //nxn matrix a                 //11
  cuComplex *x; //n-vector x                   //17,12
  cuComplex *y; //n-vector y                   //   18,13
                                               //      19,14
                                               //         20,15
                                               //            21,16
  cudaMallocManaged(&a,n*n*sizeof(cuComplex));//unif.memory a
  cudaMallocManaged(&x,n*sizeof(cuComplex));  //unif.memory x
  cudaMallocManaged(&y,n*sizeof(cuComplex));  //unif.memory 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
  cublasCreate(&handle);          // initialize CUBLAS context
  cuComplex al={1.0f,0.0f};                               // al=1
  cuComplex bet={1.0f,0.0f};                             // bet=1
// Hermitian banded matrix-vector multiplication:
// y = al*a*x + bet*y
// a - complex Hermitian banded nxn matrix;
// x,y -complex  n-vectors; al,bet - complex scalars

  cublasChbmv(handle,CUBLAS_FILL_MODE_LOWER,n,k,&al,a,n,x,1,
                                                   &bet,y,1);

  cudaDeviceSynchronize();
  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(a);                                   // free   memory
  cudaFree(x);                                   // free   memory
  cudaFree(y);                                   // free   memory
  cublasDestroy(handle);            // destroy CUBLAS context
  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]
```

2.3.37 `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]
```

2.3.38 cublasChpmv - unified memory version

```
// nvcc 031Chpmv.cu -lcublas
#include <stdio.h>
#include "cublas_v2.h"
#define n 6                   // number of rows and columns of a
int main(void){
  cublasHandle_t handle;                        // CUBLAS context
  int i,j,l,m;                   // i-row index, j-column index
// data preparation
  cuComplex *a;                  // lower triangle of a complex
                                                    //  nxn matrix
  cuComplex *x;                                // complex n-vector
  cuComplex *y;                                // complex n-vector
// unified memory for a,x,y
  cudaMallocManaged(&a,n*(n+1)/2*sizeof(cuComplex));
  cudaMallocManaged(&x,n*sizeof(cuComplex));
  cudaMallocManaged(&y,n*sizeof(cuComplex));
// 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
  cublasCreate(&handle);          // initialize CUBLAS context
  cuComplex al={1.0f,0.0f};                            //  al=1
  cuComplex bet={1.0f,0.0f};                           // bet=1
// Hermitian packed matrix-vector multiplication:
// y = al*a*x + bet*y;  a - nxn Hermitian  matrix
// in packed format; x,y - complex n-vectors;
// al,bet - complex scalars
```

```
  cublasChpmv(handle,CUBLAS_FILL_MODE_LOWER,n,&al,a,x,1,&bet,y,1);

  cudaDeviceSynchronize();
  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(a);                                     // free  memory
  cudaFree(x);                                     // free  memory
  cudaFree(y);                                     // free  memory
  cublasDestroy(handle);              // destroy CUBLAS context
  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]
```

2.3.39 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]
```

2.3.40 cublasCher - unified memory version

```
// nvcc 032cher.cu -lcublas
#include <stdio.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){
  cublasHandle_t handle;                          // CUBLAS context
  int i,j;                        // i-row index, j-column index
  cuComplex *a;                            //nxn complex matrix a
  cuComplex *x;                             //complex  n-vector x
  cudaMallocManaged(&a,n*n*sizeof(cuComplex));//unif.memory a
  cudaMallocManaged(&x,n*sizeof(cuComplex));  //unif.memory 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
  cublasCreate(&handle);          // initialize CUBLAS context
  float al=1.0f;                                        //  al=1
// rank-1 update of the Hermitian matrix d_a:
// a = al*x*x^H + a
// a - nxn Hermitian matrix; x - n-vector; al - scalar

  cublasCher(handle,CUBLAS_FILL_MODE_LOWER,n,&al,x,1,a,n);

  cudaDeviceSynchronize();
// 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(a);                                    // free  memory
  cudaFree(x);                                    // free  memory
  cublasDestroy(handle);             // destroy CUBLAS context
  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]
```

2.3.41 `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]
```

2.3.42 cublasCher2 - unified memory version

```
// nvcc 033cher2.cu -lcublas
#include <stdio.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){
  cublasHandle_t handle;                      // CUBLAS context
  int i,j;                         // i-row index, j-column index
  cuComplex *a;                                //nxn complex matrix
  cuComplex *x;                             //complex  n-vector
  cuComplex *y;                             //complex  n-vector
  cudaMallocManaged(&a,n*n*sizeof(cuComplex));//unif.memory a
  cudaMallocManaged(&x,n*sizeof(cuComplex));  //unif.memory x
  cudaMallocManaged(&y,n*sizeof(cuComplex));  //unif.memory 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
  cublasCreate(&handle);          // initialize CUBLAS context
  cuComplex al={1.0f,0.0f};                              // al=1
// rank-2 update of the Hermitian matrix a:
// a = al*x*y^H + \bar{al}*y*x^H + a
// a - nxn Hermitian matrix; x,y - n-vectors; al -scalar

  cublasCher2(handle,CUBLAS_FILL_MODE_LOWER,n,&al,x,1,y,1,a,n);

  cudaDeviceSynchronize();
// 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(a);                                  // free  memory
  cudaFree(x);                                  // free  memory
  cudaFree(y);                                  // free  memory
  cublasDestroy(handle);             // destroy CUBLAS context
  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]
```

2.3.43 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]
```

2.3.44 cublasChpr - unified memory version

```
// nvcc 034chpr.cu -lcublas
#include <stdio.h>
#include "cublas_v2.h"
#define n 6                  // number of rows and columns of a
int main(void){
  cublasHandle_t handle;                       // CUBLAS context
  int i,j,l,m;                    // i-row index, j-column index
  cuComplex *a;   // lower triangle of a complex nxn matrix a
  cuComplex *x;                          // complex n-vector x
// unified memory for a,x
  cudaMallocManaged(&a,n*(n+1)/2*sizeof(cuComplex));
  cudaMallocManaged(&x,n*sizeof(cuComplex));
// 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
  cublasCreate(&handle);         // initialize CUBLAS context
  float al=1.0f;                                      // al=1
// rank-1 update of a Hermitian packed complex matrix a:
// a = al*x*x^H + a;  a - Hermitian  nxn complex
// matrix in packed format; x - n-vector; al - scalar

  cublasChpr(handle,CUBLAS_FILL_MODE_LOWER,n,&al,x,1,a);

  cudaDeviceSynchronize();
// 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(a);                                  // free  memory
  cudaFree(x);                                  // free  memory
  cublasDestroy(handle);            // destroy CUBLAS context
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]
```

2.3.45 `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]
```

2.3.46 cublasChpr2 - unified memory version

```
// nvcc 035chpr2.cu -lcublas
#include <stdio.h>
#include "cublas_v2.h"
#define n 6                // number of rows and columns of a
int main(void){
  cublasHandle_t handle;                     // CUBLAS context
  int i,j,l,m;                     // i-row index, j-column index
  cuComplex *a;                    // lower triangle of a complex
                                                // nxn matrix
  cuComplex *x;                               // complex n-vector
  cuComplex *y;                               // complex n-vector
// unified memory for a,x,y
  cudaMallocManaged(&a,n*(n+1)/2*sizeof(cuComplex));
  cudaMallocManaged(&x,n*sizeof(cuComplex));
  cudaMallocManaged(&y,n*sizeof(cuComplex));
// 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
  cublasCreate(&handle);          // initialize CUBLAS context
  cuComplex al={1.0f,0.0f};                            //al=1
// rank-2 update of a Hermitian matrix a:
// a = al*x*y^H + \bar{al}*y*x^H + a; a -Hermitian
// nxn matrix in packed format; x,y - n-vectors; al -scalar

  cublasChpr2(handle,CUBLAS_FILL_MODE_LOWER,n,&al,x,1,y,1,a);

  cudaDeviceSynchronize();
// 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(a);                                   // free  memory
  cudaFree(x);                                   // free  memory
  cudaFree(y);                                   // free  memory
  cublasDestroy(handle);              // destroy CUBLAS context
  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]
```

2.4 CUBLAS Level-3. Matrix-matrix operations

2.4.1 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
```

2.4.2 `cublasSgemm` - unified memory version

```
// nvcc 036sgemm.cu -lcublas
#include <stdio.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){
  cublasHandle_t handle;                      // CUBLAS context
  int i,j;                     // i-row index,j-column index
  float* a;                                       // mxk matrix
  float* b;                                       // kxn matrix
  float* c;                                       // mxn matrix
// unified memory for a,b,c
  cudaMallocManaged(&a,m*k*sizeof(cuComplex));
  cudaMallocManaged(&b,k*n*sizeof(cuComplex));
  cudaMallocManaged(&c,m*n*sizeof(cuComplex));
// 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");
   }
  cublasCreate(&handle);          // initialize CUBLAS context
  float al=1.0f;                                          // al=1
  float bet=1.0f;                                         //bet=1
// matrix-matrix multiplication: c = al*a*b + bet*c
// a -mxk matrix, b -kxn matrix, c -mxn matrix;
// al,bet -scalars

  cublasSgemm(handle,CUBLAS_OP_N,CUBLAS_OP_N,m,n,k,&al,a,m,b,k,
                                                    &bet,c,m);

  cudaDeviceSynchronize();
  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(a);                                     // free  memory
  cudaFree(b);                                     // free  memory
  cudaFree(c);                                     // free  memory
  cublasDestroy(handle);             // destroy CUBLAS context
  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
```

2.4.3 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
```

2.4.4 cublasSsymm - unified memory version

```
// nvcc 037ssymm.cu -lcublas
#include <stdio.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){
  cublasHandle_t handle;                        // CUBLAS context
  int i,j;                            // i-row ind., j-column ind.
  float* a;                                       // mxm matrix a
  float* b;                                       // mxn matrix b
  float* c;                                       // mxn matrix c
// unified memory for a,b,c
  cudaMallocManaged(&a,m*m*sizeof(cuComplex));
  cudaMallocManaged(&b,m*n*sizeof(cuComplex));
  cudaMallocManaged(&c,m*n*sizeof(cuComplex));
// 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");
   }
  cublasCreate(&handle);          // initialize CUBLAS context
  float al=1.0f;                                       // al=1
  float bet=1.0f;                                     // bet=1
// symmetric matrix-matrix multiplication:
```

```
// c = al*a*b + bet*c; a - mxm symmetric matrix;
// b,c - mxn general matrices; al,bet - scalars

   cublasSsymm(handle,CUBLAS_SIDE_LEFT,CUBLAS_FILL_MODE_LOWER,
                                  m,n,&al,a,m,b,m,&bet,c,m);

   cudaDeviceSynchronize();
   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(a);                                   // free  memory
   cudaFree(b);                                   // free  memory
   cudaFree(c);                                   // free  memory
   cublasDestroy(handle);             // destroy CUBLAS context
   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
```

2.4.5 `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: d_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
```

2.4.6 cublasSsyrk - unified memory version

```
// nvcc 038ssyrk.cu -lcublas
#include <stdio.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){
  cublasHandle_t handle;                       // CUBLAS context
  int i,j;                   // i-row index, j-column index
  float* a;                                       // nxk matrix
  float* c;                                       // nxn matrix
// unified memory for a,c
  cudaMallocManaged(&a,n*k*sizeof(cuComplex));
  cudaMallocManaged(&c,n*n*sizeof(cuComplex));
// 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");
   }
  cublasCreate(&handle);         // initialize CUBLAS context
  float al=1.0f;                                          // al=1
  float bet=1.0f;                                         //bet=1
// symmetric rank-k update: c = al*a*a^T + bet*c;
// c - symmetric nxn matrix, a - general nxk matrix;
// al,bet - scalars

  cublasSsyrk(handle,CUBLAS_FILL_MODE_LOWER, CUBLAS_OP_N,
                                    n,k,&al,a,n,&bet,c,n);
  cudaDeviceSynchronize();
  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(a);                                        // free  memory
  cudaFree(c);                                        // free  memory
  cublasDestroy(handle);              // destroy CUBLAS context
  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
```

2.4.7 `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
```

2.4.8 cublasSsyr2k - unified memory version

```
// nvcc 039ssyr2k.cu -lcublas
#include <stdio.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){
  cublasHandle_t handle;                            // CUBLAS context
  int i,j;                        // i-row index, j-col. index
  float* a;                                       // nxk matrix
  float* b;                                       // nxk matrix
  float* c;                                       // nxn matrix
// unified memory for a,b,c
  cudaMallocManaged((void**)&a,n*k*sizeof(float));
  cudaMallocManaged((void**)&b,n*k*sizeof(float));
  cudaMallocManaged((void**)&c,n*n*sizeof(float));
// 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++){
     for(j=0;j<n;j++){
       if(i>=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");
   }
  cublasCreate(&handle);         // initialize CUBLAS context
  float al=1.0f;                                       // al=1
  float bet=1.0f;                                      //bet=1
// symmetric rank-2k update:  c=al*(a*b^T+b*a^T)+bet*c
// c - symmetric nxn matrix, a,b - general nxk matrices
// al,bet - scalars

  cublasSsyr2k(handle,CUBLAS_FILL_MODE_LOWER,CUBLAS_OP_N,n,k,
                                        &al,a,n,b,n,&bet,c,n);
  cudaDeviceSynchronize();
  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(a);                                      // free   memory
  cudaFree(b);                                      // free   memory
  cudaFree(c);                                      // free   memory
  cublasDestroy(handle);                // destroy CUBLAS context
  return 0;
}
// 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
```

2.4.9 `cublasStrmm` - triangular matrix-matrix multiplication

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

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

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
```

2.4.10 cublasStrmm - unified memory version

```
// nvcc 040strmm.cu -lcublas
#include <stdio.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){
  cublasHandle_t handle;                            // CUBLAS context
  int i,j;                               // i-row index, j-col. index
  float* a;                                             // mxm matrix
  float* b;                                             // mxn matrix
  float* c;                                             // mxn matrix
// unified memory for a,b,c
  cudaMallocManaged(&a,m*m*sizeof(float));
  cudaMallocManaged(&b,m*n*sizeof(float));
  cudaMallocManaged(&c,m*n*sizeof(float));
// 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");
   }
  cublasCreate(&handle);          // initialize CUBLAS context
  float al=1.0f;
// triangular matrix-matrix multiplication: c = al*a*b;
// a - mxm triangular matrix in lower mode,
// b,c -mxn general matrices; al- scalar

  cublasStrmm(handle,CUBLAS_SIDE_LEFT,CUBLAS_FILL_MODE_LOWER,
    CUBLAS_OP_N,CUBLAS_DIAG_NON_UNIT,m,n,&al,a,m,b,m,c,m);
  cudaDeviceSynchronize();
  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(a);                                  // free  memory
  cudaFree(b);                                  // free  memory
  cudaFree(c);                                  // free  memory
  cublasDestroy(handle);             // destroy CUBLAS context
  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
```

2.4.11 `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
```

2.4.12 cublasStrsm - unified memory version

```
// nvcc 041strsm.cu -lcublas
#include <stdio.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){
  cublasHandle_t handle;                          // CUBLAS context
  int i,j;                            // i-row index, j-col. index
  float* a;                                         // mxm matrix
  float* b;                                         // mxn matrix
// unified memory for a,b
  cudaMallocManaged(&a,m*m*sizeof(float));
  cudaMallocManaged(&b,m*n*sizeof(float));
// 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");
   }
  cublasCreate(&handle);         // initialize CUBLAS context
  float al=1.0f;                                       // al=1
// solve a*x=al*b; the solution x overwrites rhs b;
// a - mxm triangular matrix in lower mode;
// b,x - mxn general matrices; al - scalar

  cublasStrsm(handle,CUBLAS_SIDE_LEFT,CUBLAS_FILL_MODE_LOWER,
        CUBLAS_OP_N,CUBLAS_DIAG_NON_UNIT,m,n,&al,a,m,b,m);

  cudaDeviceSynchronize();
  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(a);                                  // free  memory
  cudaFree(b);                                  // free  memory
  cublasDestroy(handle);             // destroy CUBLAS context
  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
```

2.4.13 `cublasChemm` - Hermitian matrix-matrix multiplication

This function performs the Hermitian matrix-matrix multiplication

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

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 //
```

2.4.14 `cublasChemm` - unified memory version

```
// nvcc 042chemm.cu -lcublas
#include <stdio.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){
  cublasHandle_t handle;                        // CUBLAS context
  int i,j;                             // i-row index, j-col. ind.
// data preparation
  cuComplex *a;                           // mxm complex matrix a
  cuComplex *b;                           // mxn complex matrix b
  cuComplex *c;                           // mxn complex matrix c
  cudaMallocManaged(&a,m*m*sizeof(cuComplex));//unif.memory a
  cudaMallocManaged(&b,m*n*sizeof(cuComplex));//unif.memory b
  cudaMallocManaged(&c,m*n*sizeof(cuComplex));//unif.memory 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");
   }
  cublasCreate(&handle);          // initialize CUBLAS context
  cuComplex al={1.0f,0.0f};                              // al=1
  cuComplex bet={1.0f,0.0f};                            // bet=1
// Hermitian matrix-matrix multiplication:
// c=al*a*b+bet*c;
// a - mxm hermitian matrix; b,c - mxn-general matices;
// al,bet - scalars

  cublasChemm(handle,CUBLAS_SIDE_LEFT,CUBLAS_FILL_MODE_LOWER,
                         m,n,&al,a,m,b,m,&bet,c,m);

  cudaDeviceSynchronize();
  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(a);                                 // free  memory
  cudaFree(b);                                 // free  memory
  cudaFree(c);                                 // free  memory
  cublasDestroy(handle);             // destroy CUBLAS context
  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 //
```

2.4.15 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
```

2.4.16 cublasCherk - unified memory version

```
// nvcc 043cherk.cu -lcublas
#include <stdio.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){
  cublasHandle_t handle;                      // CUBLAS context
  int i,j;                         // i-row index, j-col. index
// data preparation
  cuComplex *a;                          // nxk complex matrix a
  cuComplex *c;                          // nxn complex matrix c
  cudaMallocManaged(&a,n*k*sizeof(cuComplex));//unif.memory a
  cudaMallocManaged(&c,n*n*sizeof(cuComplex));//unif.memory 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");
    }

  cublasCreate(&handle);          // initialize CUBLAS context
  float al=1.0f;                                       // al=1
  float bet=1.0f;                                      //bet=1
// rank-k update of a Hermitian matrix: c=al*a*a^H +bet*c
// c - nxn, Hermitian matrix; a - nxk general matrix;
// al,bet - scalars

  cublasCherk(handle,CUBLAS_FILL_MODE_LOWER,CUBLAS_OP_N,n,k,
                          &al,a,n,&bet,c,n);

  cudaDeviceSynchronize();
  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(a);                                    // free   memory
  cudaFree(c);                                    // free   memory
  cublasDestroy(handle);             // destroy CUBLAS context
  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
```

2.4.17 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 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;             // nxk complex matrix a on the host
  cuComplex *b;             // nxk complex matrix b 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*d_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
```

2.4.18 cublasCher2k - unified memory version

```
// nvcc 044cher2k.cu -lcublas
#include <stdio.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 matrices
int main(void){
  cublasHandle_t handle;                        // CUBLAS context
  int i,j;                          // i-row index, j-col. ind.
  cuComplex *a;                          // nxk complex matrix
  cuComplex *b;                          // nxk complex matrix
  cuComplex *c;                          // nxn complex matrix
  cudaMallocManaged(&a,n*k*sizeof(cuComplex));//unif.memory a
  cudaMallocManaged(&b,n*k*sizeof(cuComplex));//unif.memory b
  cudaMallocManaged(&c,n*n*sizeof(cuComplex));//unif.memory 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");
   }
  cublasCreate(&handle);         // initialize CUBLAS context
  cuComplex al={1.0f,0.0f};                             // al=1
  float bet=1.0f;                                       //bet=1
// Hermitian rank-2k update: c=al*a*b^H+\bar{al}*b*a^H +bet*c
// c - nxn, hermitian matrix; a,b - nxk general matrices;
// al,bet - scalars

  cublasCher2k(handle,CUBLAS_FILL_MODE_LOWER,CUBLAS_OP_N,n,k,
                        &al,a,n,b,n,&bet,c,n);

  cudaDeviceSynchronize();
// 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(a);                                  // free  memory
  cudaFree(b);                                  // free  memory
  cudaFree(c);                                  // free  memory
  cublasDestroy(handle);             // destroy CUBLAS context
  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 3

CUSOLVER by example

3.1 General remarks on cuSolver

CUSOLVER is a part of CUDA environment and consists of three subsets of routines:

- cuSolverDn - dense matrix routines,
- cuSolverSp - sparse matrix routines,
- cuSolverRf - sparse re-factorization (useful when solving a sequence of matrices where only the coefficients are changed but the sparsity pattern remains the same).

The most complete source of information on cuSolver is `http://docs.nvidia.com/cuda/pdf/CUSOLVER_Library.pdf`. In this chapter we restrict ourselves to presentation of cuSolverDn, which includes the following topics.

- LU, QR and Cholesky factorization.
- Linear solvers based on LU, QR and Cholesky decompositions.
- Bunch-Kaufman factorization of symmetric indefinite matrices.
- Symmetric Eigenvalue and singular value problem solvers.
- Singular value decomposition.

All subprograms have four versions corresponding to four data types:

- `S - float` – real single-precision,
- `D - double` – real double-precision,
- `C -` complex single-precision,
- `Z -` complex double-precision.

Note on error checking. It is obvious that we should check for errors on every function call. Unfortunately, such an approach doubles the length of sample codes. Therefore we decided to compute the error codes only in the case of functions contained in subsections titles. A more careful error checking one can find in examples contained in [CUSOLVER].

3.1.1 Remarks on installation and compilation

As we already said, cuSolver is a part of CUDA, so after CUDA installation, cuSolver is ready to use. Compilation of examples needs two steps. Below we show how to compile our fist example with Openblas library.

```
nvcc -Wno-deprecated-gpu-targets -c -std=c++11 -I/usr/local/
cuda/include -I/usr/include 001dgetrf_exampleBigOnes.cpp

g++ -fopenmp 001dgetrf_exampleBigOnes.o -L/usr/local/cuda/lib64
-L/usr/lib -lcudart -lcublas -lcusolver -lopenblas
```

In the case of the Bunch-Kaufman decomposition example, the routines `ssytrs/dsytrs` are needed. Since we had problems with finding them in Openblas, we used MKL:

```
nvcc -I/opt/intel/mkl/include -Wno-deprecated-gpu-targets -c
-std=c++11 -I/usr/local/cuda/include
005dsytrf_dsytrs_exampleBigOnes.cpp

icpc -mkl -fopenmp 005dsytrf_dsytrs_exampleBigOnes.o
-L/usr/local/cuda/lib64 -lcudart -lcublas -lcusolver
```

3.1.2 Remarks on hardware used in examples

In all examples we have measured the computations times. The times were obtained on the machine with Ubuntu 16.04, Cuda 8.0 and

- Intel i7 6700 CPU, 4GHz, 16 GB RAM,
- Nvidia GeForce GTX 1080 GPU.

3.2 LU decomposition and solving general linear systems

3.2.1 `cusolverDnSgetrf` and `cusolverDnSgetrs` - solving general linear system using LU decomposition in single precision

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

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

where P is a permutation matrix, L is lower triangular with unit diagonal, and U is upper diagonal. The information on the interchanged rows is contained in `d_pivot`. 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 `cusolverDnSgetrs` uses the L, U factors defined by `cusolverDnSgetrf` to solve in single precision a general linear system

$$A\,X = B.$$

```
#include <cblas.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <cuda_runtime.h>
#include <cublas_v2.h>
#include <cusolverDn.h>
#define N 8192
#define BILLION 1000000000L;
using namespace std;
int main(int argc, char*argv[]){
  struct timespec start,stop;          // variables for timing
  double accum;                        // elapsed time variable
  cublasStatus_t stat;
  cudaError cudaStatus;
  cusolverStatus_t cusolverStatus;
  cusolverDnHandle_t handle;
// declare  arrays on the host
  float *A, *B1, *B; // A - NxN matrix, B1 - auxiliary N-vect.
                // B=A*B1 -  N-vector of rhs, all on the host
// declare arrays on the device
  float *d_A, *d_B, *d_Work;   // coeff.matrix, rhs, workspace
  int *d_pivot, *d_info, Lwork;  // pivots, info, worksp. size
  int info_gpu = 0;
// prepare memory on the host
  A = (float*)malloc(N*N*sizeof(float));
  B = (float*)malloc(N*sizeof(float));
  B1 = (float*)malloc(N*sizeof(float));
  for(int i=0;i<N*N;i++) A[i]=rand()/(float)RAND_MAX;// A-rand
  for(int i=0;i<N;i++) B[i] = 0.0;              // initialize B
  for(int i=0;i<N;i++) B1[i] = 1.0;    // B1 - N-vector of ones
  float al=1.0, bet=0.0;              // coefficients for sgemv
  int incx=1, incy=1;
  cblas_sgemv(CblasColMajor,CblasNoTrans,N,N,al,A,N,B1,incx,
                        bet,B,incy);          // multiply B=A*B1
  cudaStatus = cudaGetDevice(0);
  cusolverStatus = cusolverDnCreate(&handle);
// prepare memory on the device
  cudaStatus = cudaMalloc((void**)&d_A,N*N*sizeof(float));
  cudaStatus = cudaMalloc((void**)&d_B, N*sizeof(float));
  cudaStatus = cudaMalloc((void**)&d_pivot, N*sizeof(int));
  cudaStatus = cudaMalloc((void**)&d_info, sizeof(int));
  cudaStatus = cudaMemcpy(d_A, A, N*N*sizeof(float),
                   cudaMemcpyHostToDevice);     // copy d_A<-A
  cudaStatus = cudaMemcpy(d_B, B, N*sizeof(float),
                   cudaMemcpyHostToDevice);     // copy d_B<-B
  cusolverStatus = cusolverDnSgetrf_bufferSize(handle, N, N,
   d_A, N, &Lwork);     // compute buffer size and prep.memory
  cudaStatus=cudaMalloc((void**)&d_Work,Lwork*sizeof(float));
  clock_gettime(CLOCK_REALTIME,&start);         // timer start
// LU factorization of d_A, with partial pivoting and row
// interchanges; row i is interchanged with row d_pivot(i);

  cusolverStatus = cusolverDnSgetrf(handle,N,N,d_A,N,d_Work,
                             d_pivot, d_info);

// use the LU factorization to solve the system d_A*x=d_B;
// the solution overwrites d_B

  cusolverStatus = cusolverDnSgetrs(handle, CUBLAS_OP_N, N, 1,
                             d_A, N, d_pivot, d_B,N, d_info);

  cudaStatus = cudaDeviceSynchronize();
  clock_gettime(CLOCK_REALTIME,&stop);           // timer stop
  accum=(stop.tv_sec-start.tv_sec)+            // elapsed time
       (stop.tv_nsec-start.tv_nsec)/(double)BILLION;
  printf("getrf+getrs time: %lf sec.\n",accum);//print el.time
  cudaStatus = cudaMemcpy(&info_gpu, d_info, sizeof(int),
               cudaMemcpyDeviceToHost);   //d_info -> info_gpu
  printf("after getrf+getrs: info_gpu = %d\n", info_gpu);
  cudaStatus = cudaMemcpy(B1, d_B, N*sizeof(float),
                   cudaMemcpyDeviceToHost);    // copy d_B->B1
  printf("solution: ");
  for (int i = 0; i < 5; i++) printf("%g, ", B1[i]);
  printf(" ...");    // print first components of the solution
  printf("\n");
// free  memory
  cudaStatus = cudaFree(d_A);
  cudaStatus = cudaFree(d_B);
  cudaStatus = cudaFree(d_pivot);
  cudaStatus = cudaFree(d_info);
  cudaStatus = cudaFree(d_Work);
  free(A); free(B); free(B1);
  cusolverStatus = cusolverDnDestroy(handle);
  cudaStatus = cudaDeviceReset();
  return 0;
}
//getrf+getrs time: 0.267574 sec.
//after getrf+getrs: info_gpu = 0
//solution: 1.04225, 0.873826, 1.05703, 1.03822, 0.883831,  ...
```

3.2.2 cusolverDnSgetrf and cusolverDnSgetrs - unified memory version

```
#include <cblas.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <cuda_runtime.h>
#include <cublas_v2.h>
#include <cusolverDn.h>
#define N 8192
#define BILLION 1000000000L;
using namespace std;
int main(int argc, char*argv[]){
  struct timespec start,stop;          // variables for timing
  double accum;                        // elapsed time variable
  cublasStatus_t stat;
  cudaError cudaStatus;
  cusolverStatus_t cusolverStatus;
  cusolverDnHandle_t handle;
// declare arrays
  float *A, *B1, *B; // A - NxN matrix, B1 - auxiliary N-vect.
                               // B=A*B1 -  N-vector of rhs
  float  *Work;                                 //  workspace
  int *pivot, *info, Lwork;       // pivots, info, worksp. size
  cudaMallocManaged(&A,N*N*sizeof(float)); //unified mem.for A
  cudaMallocManaged(&B,N*sizeof(float));   //unified mem.for B
  cudaMallocManaged(&B1,N*sizeof(float)); //unified mem.for B1
  for(int i=0;i<N*N;i++) A[i]=rand()/(float)RAND_MAX; //A-rand
  for(int i=0;i<N;i++) B[i] = 0.0f;              // initialize B
  for(int i=0;i<N;i++) B1[i] = 1.0f;  // B1 - N-vector of ones
  float al=1.0, bet=0.0;             // coefficients for sgemv
  int incx=1, incy=1;
  cblas_sgemv(CblasColMajor,CblasNoTrans,N,N,al,A,N,B1,incx,
                      bet,B,incy);         // multiply B=A*B1
  cudaStatus = cudaGetDevice(0);
  cusolverStatus = cusolverDnCreate(&handle);
  cudaMallocManaged(&pivot,N*sizeof(int));//unif.mem.for pivot
  cudaMallocManaged(&info,sizeof(int));    //unif.mem.for info
  cusolverStatus = cusolverDnSgetrf_bufferSize(handle, N, N,
     A, N, &Lwork);     // compute buffer size and prep.memory
  cudaMallocManaged(&Work,Lwork*sizeof(float));
  clock_gettime(CLOCK_REALTIME,&start);          // timer start
// LU factorization of A, with partial pivoting and row
// interchanges; row i is interchanged with row pivot(i);

  cusolverStatus = cusolverDnSgetrf(handle,N,N,A,N,Work,
                                      pivot, info);

// use the LU factorization to solve the system A*x=B;
// the solution overwrites B

  cusolverStatus = cusolverDnSgetrs(handle, CUBLAS_OP_N, N, 1,
                                A, N, pivot, B,N, info);

  cudaStatus = cudaDeviceSynchronize();
  clock_gettime(CLOCK_REALTIME,&stop);            // timer stop
  accum=(stop.tv_sec-start.tv_sec)+             // elapsed time
        (stop.tv_nsec-start.tv_nsec)/(double)BILLION;
  printf("getrf+getrs time: %lf sec.\n",accum);//pr.elaps.time
  printf("after getrf+getrs: info = %d\n", *info);
  printf("solution: ");
  for (int i = 0; i < 5; i++) printf("%g, ", B[i]);
  printf(" ...");    // print first components of the solution
  printf("\n");
// free  memory
  cudaStatus = cudaFree(A);
  cudaStatus = cudaFree(B);
  cudaStatus = cudaFree(pivot);
  cudaStatus = cudaFree(info);
  cudaStatus = cudaFree(Work);
  cusolverStatus = cusolverDnDestroy(handle);
  cudaStatus = cudaDeviceReset();
  return 0;
}
//getrf+getrs time: 0.295533 sec.
//after getrf+getrs: info = 0
//solution: 1.04225 0.873826, 1.05703, 1.03822, 0.883831,  ...
```

3.2.3 `cusolverDnDgetrf` and `cusolverDnDgetrs` - solving general linear system using LU decomposition in double precision

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

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

where P is a permutation matrix, L is lower triangular with unit diagonal, and U is upper diagonal. The information on the interchanged rows is contained in `d_pivot`. 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 `cusolverDnDgetrs` uses the L, U factors defined by `cusolverDnDgetrf` to solve in double precision a general linear system

$$A\,X = B.$$

```
#include <cblas.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <cuda_runtime.h>
#include <cublas_v2.h>
#include <cusolverDn.h>
#define N 8192
#define BILLION 1000000000L;
using namespace std;
int main(int argc, char*argv[]){
  struct timespec start,stop;          // variables for timing
  double accum;                        // elapsed time variable
  cublasStatus_t stat;
  cudaError cudaStatus;
  cusolverStatus_t cusolverStatus;
  cusolverDnHandle_t handle;
// declare arrays on the host
  double *A, *B1, *B; // A - NxN matrix, B1 - auxiliary N-vect.
                 // B=A*B1 -  N-vector of rhs, all on the host
// declare arrays on the device
  double *d_A, *d_B, *d_Work;  // coeff.matrix, rhs, workspace
  int *d_pivot, *d_info, Lwork;  // pivots, info, worksp. size
  int info_gpu = 0;
// prepare memory on the host
  A = (double*)malloc(N*N*sizeof(double));
  B = (double*)malloc(N*sizeof(double));
  B1 = (double*)malloc(N*sizeof(double));
  for(int i=0;i<N*N;i++) A[i]=rand()/(double)RAND_MAX;//A-rand
  for(int i=0;i<N;i++) B[i] = 0.0;              // initialize B
  for(int i=0;i<N;i++) B1[i] = 1.0;   // B1 - N-vector of ones
  double al=1.0, bet=0.0;            // coefficients for dgemv
  int incx=1, incy=1;
  cblas_dgemv(CblasColMajor,CblasNoTrans,N,N,al,A,N,B1,incx,
                      bet,B,incy);        // multiply B=A*B1
  cudaStatus = cudaGetDevice(0);
  cusolverStatus = cusolverDnCreate(&handle);
// prepare memory on the device
  cudaStatus = cudaMalloc((void**)&d_A,N*N*sizeof(double));
  cudaStatus = cudaMalloc((void**)&d_B, N*sizeof(double));
  cudaStatus = cudaMalloc((void**)&d_pivot, N*sizeof(int));
  cudaStatus = cudaMalloc((void**)&d_info, sizeof(int));
  cudaStatus = cudaMemcpy(d_A, A, N*N*sizeof(double),
                    cudaMemcpyHostToDevice);     // copy d_A<-A
  cudaStatus = cudaMemcpy(d_B, B, N*sizeof(double),
                    cudaMemcpyHostToDevice);     // copy d_B<-B
  cusolverStatus = cusolverDnDgetrf_bufferSize(handle, N, N,
     d_A, N, &Lwork);   // compute buffer size and prep.memory
  cudaStatus=cudaMalloc((void**)&d_Work,Lwork*sizeof(double));
  clock_gettime(CLOCK_REALTIME,&start);          // timer start
// LU factorization of d_A, with partial pivoting and row
// interchanges; row i is interchanged with row d_pivot(i);

  cusolverStatus = cusolverDnDgetrf(handle,N,N,d_A,N,d_Work,
                              d_pivot, d_info);

// use the LU factorization to solve the system d_A*x=d_B;
// the solution overwrites d_B

  cusolverStatus = cusolverDnDgetrs(handle, CUBLAS_OP_N, N, 1,
                         d_A, N, d_pivot, d_B,N, d_info);

  cudaStatus = cudaDeviceSynchronize();
  clock_gettime(CLOCK_REALTIME,&stop);            // timer stop
  accum=(stop.tv_sec-start.tv_sec)+              // elapsed time
        (stop.tv_nsec-start.tv_nsec)/(double)BILLION;
  printf("getrf+getrs time: %lf sec.\n",accum);//pr.elaps.time
  cudaStatus = cudaMemcpy(&info_gpu, d_info, sizeof(int),
                cudaMemcpyDeviceToHost);   //d_info -> info_gpu
  printf("after getrf+getrs: info_gpu = %d\n", info_gpu);
  cudaStatus = cudaMemcpy(B1, d_B, N*sizeof(double),
                    cudaMemcpyDeviceToHost);    // copy d_B->B1
  printf("solution: ");
  for (int i = 0; i < 5; i++) printf("%g, ", B1[i]);
  printf(" ...");    // print first components of the solution
  printf("\n");
// free  memory
  cudaStatus = cudaFree(d_A);
  cudaStatus = cudaFree(d_B);
  cudaStatus = cudaFree(d_pivot);
  cudaStatus = cudaFree(d_info);
  cudaStatus = cudaFree(d_Work);
  free(A); free(B); free(B1);
  cusolverStatus = cusolverDnDestroy(handle);
  cudaStatus = cudaDeviceReset();
  return 0;
}
//getrf+getrs time: 1.511761 sec.
//after getrf+getrs: info_gpu = 0
//solution: 1, 1, 1, 1, 1,  ...
```

3.2.4 cusolverDnDgetrf and cusolverDnDgetrs - unified memory version

```
#include <cblas.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <cuda_runtime.h>
#include <cublas_v2.h>
#include <cusolverDn.h>
#define N 8192
#define BILLION 1000000000L;
using namespace std;
int main(int argc, char*argv[]){
  struct timespec start,stop;           // variables for timing
  double accum;                         // elapsed time variable
  cublasStatus_t stat;
  cudaError cudaStatus;
  cusolverStatus_t cusolverStatus;
  cusolverDnHandle_t handle;
// declare arrays
  double *A, *B1, *B; // A - NxN matrix, B1 - auxiliary N-vect.
                                   // B=A*B1 -  N-vector of rhs
  double  *Work;                                    //  workspace
  int *pivot, *info, Lwork;       // pivots, info, worksp. size
  cudaMallocManaged(&A,N*N*sizeof(double));//unified mem.for A
  cudaMallocManaged(&B,N*sizeof(double));  //unified mem.for B
  cudaMallocManaged(&B1,N*sizeof(double));//unified mem.for B1
  for(int i=0;i<N*N;i++) A[i]=rand()/(double)RAND_MAX;//A-rand
  for(int i=0;i<N;i++) B[i] = 0.0;               // initialize B
  for(int i=0;i<N;i++) B1[i] = 1.0;    // B1 - N-vector of ones
  double al=1.0, bet=0.0;             // coefficients for dgemv
  int incx=1, incy=1;
  cblas_dgemv(CblasColMajor,CblasNoTrans,N,N,al,A,N,B1,incx,
                      bet,B,incy);         // multiply B=A*B1
  cudaStatus = cudaGetDevice(0);
  cusolverStatus = cusolverDnCreate(&handle);
  cudaMallocManaged(&pivot,N*sizeof(int));//unif.mem.for pivot
  cudaMallocManaged(&info,sizeof(int));    //unif.mem.for info
  cusolverStatus = cusolverDnDgetrf_bufferSize(handle, N, N,
     A, N, &Lwork);      // compute buffer size and prep.memory
  cudaMallocManaged(&Work,Lwork*sizeof(double));//mem.for Work
  clock_gettime(CLOCK_REALTIME,&start);          // timer start
// LU factorization of A, with partial pivoting and row
// interchanges; row i is interchanged with row pivot(i);

  cusolverStatus = cusolverDnDgetrf(handle,N,N,A,N,Work,
                             pivot, info);

// use the LU factorization to solve the system A*x=B;
// the solution overwrites B

   cusolverStatus = cusolverDnDgetrs(handle, CUBLAS_OP_N, N, 1,
                            A, N, pivot, B, N, info);

  cudaStatus = cudaDeviceSynchronize();
  clock_gettime(CLOCK_REALTIME,&stop);           // timer stop
  accum=(stop.tv_sec-start.tv_sec)+             // elapsed time
        (stop.tv_nsec-start.tv_nsec)/(double)BILLION;
  printf("getrf+getrs time: %lf sec.\n",accum);//pr.elaps.time
  printf("after getrf+getrs: info = %d\n", *info);
  printf("solution: ");
  for (int i = 0; i < 5; i++) printf("%g, ", B[i]);
  printf(" ...");    // print first components of the solution
  printf("\n");
// free  memory
  cudaStatus = cudaFree(A);
  cudaStatus = cudaFree(B);
  cudaStatus = cudaFree(pivot);
  cudaStatus = cudaFree(info);
  cudaStatus = cudaFree(Work);
  cusolverStatus = cusolverDnDestroy(handle);
  cudaStatus = cudaDeviceReset();
  return 0;
}
//getrf+getrs time: 1.595864 sec.
//after getrf+getrs: info_gpu = 0
//solution: 1, 1, 1, 1, 1,  ...
```

3.3 QR decomposition and solving general linear systems

3.3.1 `cusolverDnSgeqrf` and `cusolverDnSorgqr` - QR decomposition and checking the orthogonality in single precision

The function `cusolverDnSgeqrf` computes in single precision the QR factorization:

$$A = Q\ R,$$

where A is an $m \times n$ general matrix, R is upper triangular and Q is orthogonal. On exit the upper triangle 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 `d_tau` and the nonzero components of vectors v_k are stored on exit in parts of columns of A corresponding to the lower triangular part of A. The function `cusolverDnSorgqr` computes the orthogonal matrix Q using elementary reflectors vectors stored in A and elementary reflectors scalars stored in `d_tau`.

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <cuda_runtime.h>
#include <cublas_v2.h>
#include <cusolverDn.h>
#define BILLION 1000000000L;
int main(int argc, char*argv[])
{
  struct timespec start,stop;           // variables for timing
  double accum;                         // elapsed time variable
  cusolverDnHandle_t cusolverH;
  cublasHandle_t cublasH;
  cublasStatus_t cublas_status = CUBLAS_STATUS_SUCCESS;
  cusolverStatus_t cusolver_status = CUSOLVER_STATUS_SUCCESS;
  cudaError_t cudaStat = cudaSuccess;
  const int m = 8192;                     // number of rows of A
  const int n = 8192;                  // number of columns of A
  const int lda = m;                   // leading dimension of A
// declare matrices A and Q,R  on the host
  float *A, *Q, *R;
// prepare host memeory
  A=(float*)malloc(lda*n*sizeof(float));// matr. A on the host
  Q=(float*)malloc(lda*n*sizeof(float)); //orthogonal factor Q
  R=(float*)malloc(n*n*sizeof(float));             // R=I-Q^T*Q
  for(int i=0;i<lda*n;i++) A[i]=rand()/(float)RAND_MAX; //rand
  float *d_A, *d_R ;               // matrices A, R on the device
  float *d_tau ; // scalars defining the elementary reflectors
  int *devInfo ;                            // info on the device
  float *d_work;                       // workspace on the device
// workspace sizes
  int lwork_geqrf = 0;
  int lwork_orgqr = 0;
  int lwork = 0;
  int info_gpu = 0;               // info copied from the device
  const float h_one = 1;                    // constants used in
  const float h_minus_one = -1;      // computations of I-Q^T*Q
// create cusolver and cublas handles
  cusolver_status = cusolverDnCreate(&cusolverH);
  cublas_status = cublasCreate(&cublasH);
// prepare device memory
  cudaStat = cudaMalloc ((void**)&d_A   , sizeof(float)*lda*n);
  cudaStat = cudaMalloc ((void**)&d_tau, sizeof(float)*n);
  cudaStat = cudaMalloc ((void**)&devInfo, sizeof(int));
  cudaStat = cudaMalloc ((void**)&d_R   , sizeof(float)*n*n);
  cudaStat = cudaMemcpy(d_A, A, sizeof(float)*lda*n,
             cudaMemcpyHostToDevice);           // copy d_A <- A
// compute working space for geqrf and orgqr
  cusolver_status = cusolverDnSgeqrf_bufferSize(cusolverH,
  m, n, d_A, lda,&lwork_geqrf);  // compute Sgeqrf buffer size
  cusolver_status = cusolverDnSorgqr_bufferSize(cusolverH,
  m, n, n, d_A, lda, d_tau, &lwork_orgqr); //and Sorgqr b.size
  lwork=(lwork_geqrf > lwork_orgqr)? lwork_geqrf: lwork_orgqr;
// device memory for workspace
  cudaStat = cudaMalloc((void**)&d_work, sizeof(float)*lwork);
// QR factorization for d_A
  clock_gettime(CLOCK_REALTIME,&start);         // start timer

  cusolver_status = cusolverDnSgeqrf(cusolverH,m, n, d_A, lda,
                         d_tau, d_work, lwork, devInfo);

  cudaStat = cudaDeviceSynchronize();
  clock_gettime(CLOCK_REALTIME,&stop);            // stop timer
  accum=(stop.tv_sec-start.tv_sec)+             // elapsed time
        (stop.tv_nsec-start.tv_nsec)/(double)BILLION;
  printf("Sgeqrf time: %lf sec.\n",accum);//print elapsed time

  cudaStat = cudaMemcpy(&info_gpu, devInfo, sizeof(int),
            cudaMemcpyDeviceToHost); // copy devInfo->info_gpu
// check geqrf error code
    printf("after geqrf: info_gpu = %d\n", info_gpu);
// apply orgqr function to compute the orthogonal matrix Q
// using elementary  reflectors vectors stored in d_A and
// elementary  reflectors  scalars stored in d_tau,

  cusolver_status= cusolverDnSorgqr(cusolverH, m, n, n, d_A,
                         lda, d_tau, d_work, lwork, devInfo);

  cudaStat = cudaDeviceSynchronize();
  cudaStat = cudaMemcpy(&info_gpu, devInfo, sizeof(int),
            cudaMemcpyDeviceToHost); // copy devInfo->info_gpu
// check orgqr error code
  printf("after orgqr: info_gpu = %d\n", info_gpu);
  cudaStat = cudaMemcpy(Q, d_A, sizeof(float)*lda*n,
            cudaMemcpyDeviceToHost);            // copy d_A->Q
  memset(R, 0, sizeof(double)*n*n);      // nxn matrix of zeros
  for(int j = 0 ; j < n ; j++){
      R[j + n*j] = 1.0f;                 // ones on the diagonal
    }
  cudaStat = cudaMemcpy(d_R, R, sizeof(float)*n*n,
               cudaMemcpyHostToDevice);         // copy R-> d_R
// compute R = -Q**T*Q + I
  cublas_status=cublasSgemm_v2(cublasH,CUBLAS_OP_T,CUBLAS_OP_N,
    n, n, m, &h_minus_one, d_A, lda, d_A, lda, &h_one, d_R,n);
  float dR_nrm2 = 0.0;                              // norm value
// compute the norm of R = -Q**T*Q + I
  cublas_status = cublasSnrm2_v2(cublasH,n*n,d_R,1,&dR_nrm2);
  printf("||I - Q^T*Q|| = %E\n", dR_nrm2);    // print the norm
// free memory
  cudaFree(d_A);
  cudaFree(d_tau);
  cudaFree(devInfo);
  cudaFree(d_work);
  cudaFree(d_R);
  cublasDestroy(cublasH);
  cusolverDnDestroy(cusolverH);
  cudaDeviceReset();
  return 0;
}
//Sqeqrf time: 0.434779 sec.
//after geqrf: info_gpu = 0
//after orgqr: info_gpu = 0
//|I - Q**T*Q| = 2.515004E-04
```

3.3.2 cusolverDnSgeqrf and cusolverDnSorgqr - unified memory version

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <cuda_runtime.h>
#include <cublas_v2.h>
#include <cusolverDn.h>
#define BILLION 1000000000L;
int main(int argc, char*argv[])
{
  struct timespec start,stop;          // variables for timing
  double accum;                        // elapsed time variable
  cusolverDnHandle_t cusolverH;
  cublasHandle_t cublasH;
  cublasStatus_t cublas_status = CUBLAS_STATUS_SUCCESS;
  cusolverStatus_t cusolver_status = CUSOLVER_STATUS_SUCCESS;
  cudaError_t cudaStat = cudaSuccess;
  const int m = 8192;                      // number of rows of A
  const int n = 8192;                   // number of columns of A
  const int lda = m;                    // leading dimension of A
// declare matrices A and Q,R
  float *A, *Q, *R;
  cudaMallocManaged(&A,lda*n*sizeof(float)); //unif. mem.for A
  cudaMallocManaged(&Q,lda*n*sizeof(float)); //unif. mem.for Q
  cudaMallocManaged(&R,n*n*sizeof(float)); //mem.for R=I-Q^T*Q
  for(int i=0;i<lda*n;i++) A[i]=rand()/(float)RAND_MAX; //rand
  float *tau ;   // scalars defining the elementary reflectors
  int *Info ;                                             // info
  float *work;                                      // workspace
// workspace sizes
  int lwork_geqrf = 0;
  int lwork_orgqr = 0;
  int lwork = 0;
  const float h_one = 1;                     // constants used in
  const float h_minus_one = -1;      // computations of I-Q^T*Q
// create cusolver and cublas handles
```

```
  cusolver_status = cusolverDnCreate(&cusolverH);
  cublas_status = cublasCreate(&cublasH);
// prepare  memory
  cudaMallocManaged(&tau,n*sizeof(float));  //unif.mem.for tau
  cudaMallocManaged(&Info,sizeof(int));    //unif.mem.for Info
// compute working space for geqrf and orgqr
  cusolver_status = cusolverDnSgeqrf_bufferSize(cusolverH,
  m, n, A, lda, &lwork_geqrf);   // compute Sgeqrf buffer size
  cusolver_status = cusolverDnSorgqr_bufferSize(cusolverH,
  m, n, n, A, lda, tau, &lwork_orgqr);     //and Sorgqr b.size
  lwork=(lwork_geqrf > lwork_orgqr)? lwork_geqrf: lwork_orgqr;
// memory for workspace
  cudaMallocManaged(&work,lwork*sizeof(float)); //mem.for work
// QR factorization for A
  clock_gettime(CLOCK_REALTIME,&start);          // start timer

  cusolver_status = cusolverDnSgeqrf(cusolverH, m, n, A, lda,
                           tau, work, lwork, Info);

  cudaStat = cudaDeviceSynchronize();
  clock_gettime(CLOCK_REALTIME,&stop);            // stop timer
  accum=(stop.tv_sec-start.tv_sec)+              // elapsed time
        (stop.tv_nsec-start.tv_nsec)/(double)BILLION;
  printf("Sgeqrf time :%lf\n",accum);    // print elapsed time
// check geqrf error code
  printf("after geqrf: info = %d\n", *Info);
// apply orgqr function to compute the orthogonal matrix Q
// using elementary  reflectors vectors stored in A and
// elementary  reflectors  scalars stored in tau,

  cusolver_status= cusolverDnSorgqr(cusolverH, m, n, n, A,
                       lda, tau, work, lwork, Info);

  cudaStat = cudaDeviceSynchronize();
// check orgqr error code
  printf("after orgqr: info = %d\n", *Info);
  memset(R, 0, sizeof(float)*n*n);      // nxn matrix of zeros
  for(int j = 0 ; j < n ; j++){
     R[j + n*j] = 1.0;                 // ones on the diagonal
    }
// compute R = -Q**T*Q + I
  cublas_status=cublasSgemm_v2(cublasH,CUBLAS_OP_T,CUBLAS_OP_N,
    n, n, m, &h_minus_one, A, lda, A, lda, &h_one, R, n);
  float nrm2 = 0.0;                                // norm value
// compute the norm of R = -Q**T*Q + I
  cublas_status = cublasSnrm2_v2(cublasH,n*n,R,1,&nrm2);
  printf("||I - Q^T*Q|| = %E\n", nrm2);      // print the norm
// free memory
  cudaFree(A);
  cudaFree(Q);
  cudaFree(R);
  cudaFree(tau);
  cudaFree(Info);
  cudaFree(work);
  cublasDestroy(cublasH);
  cusolverDnDestroy(cusolverH);
  cudaDeviceReset();
  return 0;
}
//Sgeqrf time :0.470025
//after geqrf: info = 0
//after orgqr: info = 0
//||I - Q^T*Q|| = 2.515004E-04
```

3.3.3 `cusolverDnDgeqrf` and `cusolverDnDorgqr` - QR decomposition and checking the orthogonality in double precision

The function `cusolverDnDgeqrf` 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. 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 `d_tau` 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. The function `cusolverDnDorgqr` computes the orthogonal matrix Q using elementary reflectors vectors stored in A and elementary reflectors scalars stored in `d_tau`.

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <cuda_runtime.h>
#include <cublas_v2.h>
#include <cusolverDn.h>
#define BILLION 1000000000L;
int main(int argc, char*argv[])
{
  struct timespec start,stop;          // variables for timing
  double accum;                       // elapsed time variable
  cusolverDnHandle_t cusolverH;
  cublasHandle_t cublasH;
  cublasStatus_t cublas_status = CUBLAS_STATUS_SUCCESS;
  cusolverStatus_t cusolver_status = CUSOLVER_STATUS_SUCCESS;
  cudaError_t cudaStat = cudaSuccess;
  const int m = 8192;                    // number of rows of A
  const int n = 8192;                 // number of columns of A
  const int lda = m;                  // leading dimension of A
// declare matrices A and Q,R on the host
  double *A, *Q, *R;
```

```
// prepare host memeory
  A=(double*)malloc(lda*n*sizeof(double));//matr.A on the host
  Q=(double*)malloc(lda*n*sizeof(double));//orthogonal matr. Q
  R=(double*)malloc(n*n*sizeof(double));          // R=I-Q^T*Q
  for(int i=0;i<lda*n;i++) A[i]=rand()/(double)RAND_MAX;//rand
  double *d_A, *d_R ;            // matrices A, R on the device
  double *d_tau ;// scalars defining the elementary reflectors
  int *devInfo ;                         // info on the device
  double *d_work;                   // workspace on the device
// workspace sizes
  int lwork_geqrf = 0;
  int lwork_orgqr = 0;
  int lwork = 0;
  int info_gpu = 0;              // info copied from the device
  const double h_one = 1;                  // constants used in
  const double h_minus_one = -1;     // computations of I-Q^T*Q
// create cusolver and cublas handles
  cusolver_status = cusolverDnCreate(&cusolverH);
  cublas_status = cublasCreate(&cublasH);
// prepare device memory
  cudaStat = cudaMalloc ((void**)&d_A  , sizeof(double)*lda*n);
  cudaStat = cudaMalloc ((void**)&d_tau, sizeof(double)*n);
  cudaStat = cudaMalloc ((void**)&devInfo, sizeof(int));
  cudaStat = cudaMalloc ((void**)&d_R  , sizeof(double)*n*n);
  cudaStat = cudaMemcpy(d_A, A, sizeof(double)*lda*n,
             cudaMemcpyHostToDevice);          // copy d_A <- A
// compute working space for geqrf and orgqr
  cusolver_status = cusolverDnDgeqrf_bufferSize(cusolverH,
  m, n, d_A, lda,&lwork_geqrf);  // compute Sgeqrf buffer size
  cusolver_status = cusolverDnDorgqr_bufferSize(cusolverH,
  m, n, n, d_A, lda, d_tau, &lwork_orgqr); //and Sorgqr b.size
  lwork=(lwork_geqrf > lwork_orgqr)? lwork_geqrf: lwork_orgqr;
// device memory for workspace
  cudaStat = cudaMalloc((void**)&d_work, sizeof(double)*lwork);
// QR factorization for d_A
  clock_gettime(CLOCK_REALTIME,&start);          // start timer

  cusolver_status = cusolverDnDgeqrf(cusolverH, m, n, d_A, lda,
                           d_tau, d_work, lwork, devInfo);

  cudaStat = cudaDeviceSynchronize();
  clock_gettime(CLOCK_REALTIME,&stop);            // stop timer
  accum=(stop.tv_sec-start.tv_sec)+             // elapsed time
       (stop.tv_nsec-start.tv_nsec)/(double)BILLION;
  printf("Dgeqrf time :%lf sec.\n",accum);//print elapsed time

  cudaStat = cudaMemcpy(&info_gpu, devInfo, sizeof(int),
            cudaMemcpyDeviceToHost); // copy devInfo->info_gpu
// check geqrf error code
  printf("after geqrf: info_gpu = %d\n", info_gpu);
// apply orgqr function to compute the orthogonal matrix Q
// using elementary  reflectors vectors stored in d_A and
// elementary  reflectors  scalars stored in d_tau,

  cusolver_status= cusolverDnDorgqr(cusolverH, m, n, n, d_A,
                       lda, d_tau, d_work, lwork, devInfo);

  cudaStat = cudaDeviceSynchronize();
  cudaStat = cudaMemcpy(&info_gpu, devInfo, sizeof(int),
            cudaMemcpyDeviceToHost); // copy devInfo->info_gpu
// check orgqr error code
  printf("after orgqr: info_gpu = %d\n", info_gpu);
  cudaStat = cudaMemcpy(Q, d_A, sizeof(double)*lda*n,
            cudaMemcpyDeviceToHost);            // copy d_A->Q
  memset(R, 0, sizeof(double)*n*n);     // nxn matrix of zeros
  for(int j = 0 ; j < n ; j++){
      R[j + n*j] = 1.0;                 // ones on the diagonal
    }
  cudaStat = cudaMemcpy(d_R, R, sizeof(double)*n*n,
               cudaMemcpyHostToDevice);         // copy R-> d_R
// compute R = -Q**T*Q + I
  cublas_status=cublasDgemm_v2(cublasH,CUBLAS_OP_T,CUBLAS_OP_N,
    n, n, m, &h_minus_one, d_A, lda, d_A, lda, &h_one, d_R,n);
  double dR_nrm2 = 0.0;                            // norm value
// compute the norm of R = -Q**T*Q + I
  cublas_status = cublasDnrm2_v2(cublasH,n*n,d_R,1,&dR_nrm2);
  printf("||I - Q^T*Q|| = %E\n", dR_nrm2);   // print the norm
// free memory
  cudaFree(d_A);
  cudaFree(d_tau);
  cudaFree(devInfo);
  cudaFree(d_work);
  cudaFree(d_R);
  cublasDestroy(cublasH);
  cusolverDnDestroy(cusolverH);
  cudaDeviceReset();
  return 0;
}
//Dgeqrf time: 3.324072 sec.
//after geqrf: info_gpu = 0
//after orgqr: info_gpu = 0
//|I - Q**T*Q| = 4.646390E-13
```

3.3.4 cusolverDnDgeqrf and cusolverDnDorgqr - unified memory version

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <cuda_runtime.h>
#include <cublas_v2.h>
```

```
#include <cusolverDn.h>
#define BILLION 1000000000L;
int main(int argc, char*argv[])
{
  struct timespec start,stop;          // variables for timing
  double accum;                        // elapsed time variable
  cusolverDnHandle_t cusolverH;
  cublasHandle_t cublasH;
  cublasStatus_t cublas_status = CUBLAS_STATUS_SUCCESS;
  cusolverStatus_t cusolver_status = CUSOLVER_STATUS_SUCCESS;
  cudaError_t cudaStat = cudaSuccess;
  const int m = 8192;                    // number of rows of A
  const int n = 8192;                 // number of columns of A
  const int lda = m;                  // leading dimension of A
// declare matrices A and Q,R
  double *A, *Q, *R;
  cudaMallocManaged(&A,lda*n*sizeof(double));//unif. mem.for A
  cudaMallocManaged(&Q,lda*n*sizeof(double));//unif. mem.for Q
  cudaMallocManaged(&R,n*n*sizeof(double));//mem.for R=I-Q^T*Q
  for(int i=0;i<lda*n;i++) A[i]=rand()/(double)RAND_MAX;//rand
  double *tau ;  // scalars defining the elementary reflectors
  int *Info ;                                          // info
  double *work;                                   // workspace
// workspace sizes
  int lwork_geqrf = 0;
  int lwork_orgqr = 0;
  int lwork = 0;
  const double h_one = 1;                   // constants used in
  const double h_minus_one = -1;    // computations of I-Q^T*Q
// create cusolver and cublas handles
  cusolver_status = cusolverDnCreate(&cusolverH);
  cublas_status = cublasCreate(&cublasH);
// prepare  memory
  cudaMallocManaged(&tau,n*sizeof(double)); //unif.mem.for tau
  cudaMallocManaged(&Info,sizeof(int));    //unif.mem.for Info
// compute working space for geqrf and orgqr
  cusolver_status = cusolverDnDgeqrf_bufferSize(cusolverH,
  m, n, A, lda, &lwork_geqrf);   // compute Sgeqrf buffer size
  cusolver_status = cusolverDnDorgqr_bufferSize(cusolverH,
  m, n, n, A, lda, tau, &lwork_orgqr);  //and Sorgqr buff.size
  lwork=(lwork_geqrf > lwork_orgqr)? lwork_geqrf: lwork_orgqr;
// memory for workspace
  cudaMallocManaged(&work,lwork*sizeof(double));//mem.for work
// QR factorization for A
  clock_gettime(CLOCK_REALTIME,&start);         // start timer

  cusolver_status = cusolverDnDgeqrf(cusolverH,m, n, A, lda,
                            tau, work, lwork, Info);

  cudaStat = cudaDeviceSynchronize();
  clock_gettime(CLOCK_REALTIME,&stop);           // stop timer
  accum=(stop.tv_sec-start.tv_sec)+            // elapsed time
```

```
        (stop.tv_nsec-start.tv_nsec)/(double)BILLION;
  printf("Dgeqrf time :%lf sec.\n",accum);//print elapsed time
// check geqrf error code
  printf("after geqrf: info = %d\n", *Info);
// apply orgqr function to compute the orthogonal matrix Q
// using elementary  reflectors vectors stored in A and
// elementary  reflectors  scalars stored in tau,

  cusolver_status= cusolverDnDorgqr(cusolverH, m, n, n, A,
                          lda, tau, work, lwork, Info);

  cudaStat = cudaDeviceSynchronize();
// check orgqr error code
  printf("after orgqr: info = %d\n", *Info);
  memset(R, 0, sizeof(double)*n*n);      // nxn matrix of zeros
  for(int j = 0 ; j < n ; j++){
      R[j + n*j] = 1.0;                 // ones on the diagonal
    }
// compute R = -Q**T*Q + I
  cublas_status=cublasDgemm_v2(cublasH,CUBLAS_OP_T,CUBLAS_OP_N,
    n, n, m, &h_minus_one, A, lda, A, lda, &h_one, R, n);
  double nrm2 = 0.0;                                 // norm value
// compute the norm of R = -Q**T*Q + I
  cublas_status = cublasDnrm2_v2(cublasH,n*n,R,1,&nrm2);
  printf("||I - Q^T*Q|| = %E\n", nrm2);       // print the norm
// free memory
  cudaFree(A);
  cudaFree(Q);
  cudaFree(R);
  cudaFree(tau);
  cudaFree(Info);
  cudaFree(work);
  cublasDestroy(cublasH);
  cusolverDnDestroy(cusolverH);
  cudaDeviceReset();
  return 0;
}
//Dgeqrf time :3.398122 sec.
//after geqrf: info = 0
//after orgqr: info = 0
//||I - Q^T*Q|| = 4.646390E-13
```

3.3.5 `cusolverDnSgeqrf` and `cusolverDnSormqr`, `cublasStrsm` - QR decomposition and solving a linear system in single precision

The function `cusolverDnSgeqrf` computes in single precision the QR factorization:

$$A = Q\,R,$$

where A is an $m \times n$ general matrix, R is upper triangular and Q is orthogonal. On exit the upper triangle 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 `d_tau` and the nonzero components of vectors v_k are stored on exit in parts of columns of A corresponding to the lower triangular part of A. The function `cusolverDnSormqr` computes $Q^T * B$, the original system $A * X = (Q * R) * X = B$ can be written in the form $R * X = Q^T * B$ and `cublasStrsm` solves the obtained triangular system.

```
#include <cblas.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include <cublas_v2.h>
#include <cusolverDn.h>
#define BILLION 1000000000L;
int main(int argc, char*argv[])
{
  struct timespec start,stop;          // variables for timing
  double accum;                        // elapsed time variable
  cusolverDnHandle_t cusolverH;              // cusolver handle
  cublasHandle_t cublasH;                      // cublas handle
  cublasStatus_t cublas_status = CUBLAS_STATUS_SUCCESS;
  cusolverStatus_t cusolver_status = CUSOLVER_STATUS_SUCCESS;
// variables for error checking in  cudaMalloc
  cudaError_t cudaStat1 = cudaSuccess;
  cudaError_t cudaStat2 = cudaSuccess;
  cudaError_t cudaStat3 = cudaSuccess;
  cudaError_t cudaStat4 = cudaSuccess;
  const int m = 8192;                      // number of rows of A
  const int lda = m;                    // leading dimension of A
  const int ldb = m;                    // leading dimension of B
  const int nrhs = 1;             // number of right hand sides
// A - mxm coeff. matr.,  B=A*B1 -right hand side, B1 - mxnrhs
  float *A, *B, *B1, *X;        // - auxil.matrix, X - solution
// prepare memory on the host
  A=(float*)malloc(lda*m*sizeof(float));
  B=(float*)malloc(ldb*nrhs*sizeof(float));
  B1=(float*)malloc(ldb*nrhs*sizeof(float));
  X=(float*)malloc(ldb*nrhs*sizeof(float));
  for(int i=0;i<lda*m;i++) A[i]=rand()/(float)RAND_MAX;
  for(int i=0;i<ldb*nrhs;i++) B[i]=0.0;;
  for(int i=0;i<ldb*nrhs;i++) B1[i]=1.0;
  float al=1.0,bet=0.0;                    // constants for sgemv
  int incx=1, incy=1;
  cblas_sgemv(CblasColMajor,CblasNoTrans,m,m,al,A,m,B1,incx,
                                   bet,B,incy);       //B=A*B1
// declare arrays on the device
  float *d_A, *d_B, *d_tau, *d_work ;
  int *devInfo ;                        // device version of info
  int  lwork = 0;                                // workspace size
  int info_gpu = 0;               // device info copied to host
  const float one = 1;
// create cusolver and cublas handles
  cusolver_status = cusolverDnCreate(&cusolverH);
  cublas_status = cublasCreate(&cublasH);
// prepare memory on the device
  cudaStat1 = cudaMalloc((void**)&d_A,sizeof(float)*lda*m);
  cudaStat2 = cudaMalloc((void**)&d_tau, sizeof(float) * m);
  cudaStat3 = cudaMalloc((void**)&d_B,sizeof(float)*ldb*nrhs);
  cudaStat4 = cudaMalloc((void**)&devInfo, sizeof(int));
// copy A,B from host to device
  cudaStat1 = cudaMemcpy(d_A,A,sizeof(float)*lda*m,
                             cudaMemcpyHostToDevice); // A->d_A
  cudaStat2 = cudaMemcpy(d_B,B,sizeof(float)*ldb*nrhs,
                             cudaMemcpyHostToDevice); // B->d_B
// compute buffer size for geqrf and prepare worksp. on device
  cusolver_status = cusolverDnSgeqrf_bufferSize(cusolverH, m,
                                   m, d_A, lda, &lwork);
  cudaStat1 = cudaMalloc((void**)&d_work,sizeof(float)*lwork);
  clock_gettime(CLOCK_REALTIME,&start);         // start timer
// QR factorization for d_A;    R  stored in upper triangle of
// d_A, elementary reflectors vectors stored in lower triangle
// of d_A, elementary  reflectors  scalars stored in d_tau

  cusolver_status = cusolverDnSgeqrf(cusolverH, m, m, d_A, lda,
                       d_tau, d_work, lwork, devInfo);

  cudaStat1 = cudaDeviceSynchronize();           // stop timer
  clock_gettime(CLOCK_REALTIME,&stop);
  accum=(stop.tv_sec-start.tv_sec)+            // elapsed time
        (stop.tv_nsec-start.tv_nsec)/(double)BILLION;
  printf("Sgeqrf time: %lf sec.\n",accum);//print elapsed time

  cudaStat1 = cudaMemcpy(&info_gpu,devInfo,sizeof(int),
         cudaMemcpyDeviceToHost);       // devInfo -> info_gpu
// check error code of geqrf function
  printf("after geqrf: info_gpu = %d\n", info_gpu);
// compute d_B=Q^T*B using ormqr function

  cusolver_status=cusolverDnSormqr(cusolverH,CUBLAS_SIDE_LEFT,
  CUBLAS_OP_T, m, nrhs, m, d_A, lda, d_tau, d_B, ldb, d_work,
                           lwork, devInfo);

  cudaStat1 = cudaDeviceSynchronize();
  cudaStat1 = cudaMemcpy(&info_gpu, devInfo, sizeof(int),
         cudaMemcpyDeviceToHost);       // devInfo -> info_gpu
// check error code of ormqr function
  printf("after ormqr: info_gpu = %d\n", info_gpu);
// write the original system A*X=(Q*R)*X=B in the form
// R*X=Q^T*B and solve the obtained triangular system
```

```
  cublas_status = cublasStrsm(cublasH,CUBLAS_SIDE_LEFT,
  CUBLAS_FILL_MODE_UPPER,CUBLAS_OP_N, CUBLAS_DIAG_NON_UNIT,
                 m, nrhs, &one, d_A, lda, d_B, ldb);

  cudaStat1 = cudaDeviceSynchronize();
  cudaStat1 = cudaMemcpy(X,d_B,sizeof(float)*ldb*nrhs,
              cudaMemcpyDeviceToHost);          // copy d_B->X
  printf("solution: ");//show first components of the solution
  for (int i = 0; i < 5; i++) printf("%g, ", X[i]);
  printf(" ...");
  printf("\n");
// free memory
  cudaFree(d_A);
  cudaFree(d_tau);
  cudaFree(d_B);
  cudaFree(devInfo);
  cudaFree(d_work);
  cublasDestroy(cublasH);
  cusolverDnDestroy(cusolverH);
  cudaDeviceReset();
  return 0;
}
//Sgeqrf time: 0.435715 sec.
//after geqrf: info_gpu = 0
//after ormqr: info_gpu = 0
//solution: 1.00008, 1.02025, 1.00586, 0.999749, 1.00595,  ...
```

3.3.6 cusolverDnSgeqrf and cusolverDnSormqr, cublasStrsm - unified memory version

```
#include <cblas.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include <cublas_v2.h>
#include <cusolverDn.h>
#define BILLION 1000000000L;
int main(int argc, char*argv[])
{
  struct timespec start,stop;          // variables for timing
  double accum;                        // elapsed time variable
  cusolverDnHandle_t cusolverH;              // cusolver handle
  cublasHandle_t cublasH;                      // cublas handle
  cublasStatus_t cublas_status = CUBLAS_STATUS_SUCCESS;
  cusolverStatus_t cusolver_status = CUSOLVER_STATUS_SUCCESS;
  cudaError_t cudaStat1 = cudaSuccess;
  const int m = 8192;                    // number of rows of A
  const int lda = m;                  // leading dimension of A
  const int ldb = m;                  // leading dimension of B
  const int nrhs = 1;             // number of right hand sides
// A - mxm coeff. matr.,  B=A*B1 -right hand side, B1 - mxnrhs
  float *A, *B, *B1, *X;        // - auxil.matrix, X - solution
// prepare unified memory
  cudaMallocManaged(&A,lda*m*sizeof(float)); //unif. mem.for A
  cudaMallocManaged(&B,ldb*nrhs*sizeof(float));//uni.mem.for A
  cudaMallocManaged(&B1,ldb*nrhs*sizeof(float));//u.mem.for B1
  for(int i=0;i<lda*m;i++) A[i]=rand()/(float)RAND_MAX;
  for(int i=0;i<ldb*nrhs;i++) B[i]=0.0f;
  for(int i=0;i<ldb*nrhs;i++) B1[i]=1.0f;
  float al=1.0,bet=0.0;                 // constants for sgemv
  int incx=1, incy=1;
  cblas_sgemv(CblasColMajor,CblasNoTrans,m,m,al,A,m,B1,incx,
                                  bet,B,incy);        //B=A*B1
  float *tau, *work ;   //elem. reflectors scalars,  workspace
  int *Info ;                                        //  info
  int  lwork = 0;                           // workspace size
  const float one = 1;
// create cusolver and cublas handles
  cusolver_status = cusolverDnCreate(&cusolverH);
  cublas_status = cublasCreate(&cublasH);
  cudaMallocManaged(&tau,m*sizeof(float));//unif. mem. for tau
  cudaMallocManaged(&Info,sizeof(int));  //unif. mem. for Info
// compute buffer size for geqrf and prepare workspace
  cusolver_status=cusolverDnSgeqrf_bufferSize(cusolverH, m, m,
                                  A, lda, &lwork);
  cudaMallocManaged(&work,lwork*sizeof(float)); //mem.for work
  clock_gettime(CLOCK_REALTIME,&start);          // start timer
// QR factorization for A;   R  stored in upper triangle of  A
// elementary reflectors vectors stored in lower triangle of A
// elementary  reflectors  scalars stored in tau

  cusolver_status = cusolverDnSgeqrf(cusolverH, m, m, A, lda,
                    tau, work, lwork, Info);

  cudaStat1 = cudaDeviceSynchronize();            // stop timer
  clock_gettime(CLOCK_REALTIME,&stop);
  accum=(stop.tv_sec-start.tv_sec)+             // elapsed time
        (stop.tv_nsec-start.tv_nsec)/(double)BILLION;
  printf("Sgeqrf time: %lf sec.\n",accum);//print elapsed time
// check error code of geqrf function
  printf("after geqrf: info = %d\n", *Info);
// compute B=Q^T*B using ormqr function

  cusolver_status=cusolverDnSormqr(cusolverH,CUBLAS_SIDE_LEFT,
      CUBLAS_OP_T,m,nrhs,m,A,lda,tau,B,ldb,work,lwork,Info);

  cudaStat1 = cudaDeviceSynchronize();
// check error code of ormqr function
  printf("after ormqr: info = %d\n", *Info);
// write the original system A*X=(Q*R)*X=B in the form
// R*X=Q^T*B and solve the obtained triangular system

  cublas_status = cublasStrsm(cublasH,CUBLAS_SIDE_LEFT,
  CUBLAS_FILL_MODE_UPPER,CUBLAS_OP_N, CUBLAS_DIAG_NON_UNIT,
                    m, nrhs, &one, A, lda, B, ldb);

  cudaStat1 = cudaDeviceSynchronize();
  printf("solution: ");//show first components of the solution
  for (int i = 0; i < 5; i++) printf("%g, ", B[i]);
  printf(" ...");
  printf("\n");
// free memory
  cudaFree(A);
  cudaFree(tau);
  cudaFree(B);
  cudaFree(B1);
  cudaFree(Info);
  cudaFree(work);
  cublasDestroy(cublasH);
  cusolverDnDestroy(cusolverH);
  cudaDeviceReset();
  return 0;
}
//Sgeqrf time: 0.465168 sec.
//after geqrf: info = 0
//after ormqr: info = 0
//solution: 1.00008, 1.02025, 1.00586, 0.999749, 1.00595,  ...
```

3.3.7 `cusolverDnDgeqrf` and `cusolverDnDormqr`, `cublasDtrsm` - QR decomposition and solving a linear system in doble precision

The function `cusolverDnDgeqrf` computes in double precision the QR factorization:

$$A = Q\,R,$$

where A is an $m \times n$ general matrix, R is upper triangular and Q is orthogonal. On exit the upper triangle 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 `d_tau` and the nonzero components of vectors v_k are stored on exit in parts of columns of A corresponding to the lower triangular part of A. The function `cusolverDnDormqr` computes $Q^T * B$, the original system $A * X = (Q * R) * X = B$ can be written in the form $R * X = Q^T * B$ and `cublasDtrsm` solves the obtained triangular system.

```
#include <cblas.h>
#include <time.h>
#include <stdio.h>
```

```
#include <stdlib.h>
#include <cuda_runtime.h>
#include <cublas_v2.h>
#include <cusolverDn.h>
#define BILLION 1000000000L;
int main(int argc, char*argv[])
{
  struct timespec start,stop;          // variables for timing
  double accum;                        // elapsed time variable
  cusolverDnHandle_t cusolverH;              // cusolver handle
  cublasHandle_t cublasH;                      // cublas handle
  cublasStatus_t cublas_status = CUBLAS_STATUS_SUCCESS;
  cusolverStatus_t cusolver_status = CUSOLVER_STATUS_SUCCESS;
// variables for error checking in  cudaMalloc
  cudaError_t cudaStat1 = cudaSuccess;
  cudaError_t cudaStat2 = cudaSuccess;
  cudaError_t cudaStat3 = cudaSuccess;
  cudaError_t cudaStat4 = cudaSuccess;
  const int m = 8192;                     // number of rows of A
  const int lda = m;                   // leading dimension of A
  const int ldb = m;                   // leading dimension of B
  const int nrhs = 1;             // number of right hand sides
// A - mxm coeff. matr.,  B=A*B1 -right hand side, B1 - mxnrhs
  double *A, *B, *B1, *X;       // - auxil.matrix, X - solution
// prepare memory on the host
  A=(double*)malloc(lda*m*sizeof(double));
  B=(double*)malloc(ldb*nrhs*sizeof(double));
  B1=(double*)malloc(ldb*nrhs*sizeof(double));
  X=(double*)malloc(ldb*nrhs*sizeof(double));
  for(int i=0;i<lda*m;i++) A[i]=rand()/(double)RAND_MAX;
  for(int i=0;i<ldb*nrhs;i++) B[i]=0.0;;
  for(int i=0;i<ldb*nrhs;i++) B1[i]=1.0;
  double al=1.0,bet=0.0;                  // constants for dgemv
  int incx=1, incy=1;
  cblas_dgemv(CblasColMajor,CblasNoTrans,m,m,al,A,m,B1,incx,
                                 bet,B,incy);       //B=A*B1
// declare arrays on the device
  double *d_A, *d_B, *d_tau, *d_work ;
  int *devInfo ;                       // device version of info
  int  lwork = 0;                              // workspace size
  int info_gpu = 0;               // device info copied to host
  const double one = 1;
// create cusolver and cublas handles
  cusolver_status = cusolverDnCreate(&cusolverH);
  cublas_status = cublasCreate(&cublasH);
// prepare memory on the device
  cudaStat1 = cudaMalloc((void**)&d_A,sizeof(double)*lda*m);
  cudaStat2 = cudaMalloc((void**)&d_tau, sizeof(double) * m);
  cudaStat3 = cudaMalloc((void**)&d_B,sizeof(double)*ldb*nrhs);
  cudaStat4 = cudaMalloc((void**)&devInfo, sizeof(int));
// copy A,B from host to device
  cudaStat1 = cudaMemcpy(d_A,A,sizeof(double)*lda*m,
                                   cudaMemcpyHostToDevice); // A->d_A
  cudaStat2 = cudaMemcpy(d_B,B,sizeof(double)*ldb*nrhs,
                                   cudaMemcpyHostToDevice); // B->d_B
// compute buffer size for geqrf and prepare worksp. on device
  cusolver_status=cusolverDnDgeqrf_bufferSize(cusolverH, m, m,
                                       d_A, lda, &lwork);
  cudaStat1=cudaMalloc((void**)&d_work,sizeof(double)*lwork);
  clock_gettime(CLOCK_REALTIME,&start);         // start timer
// QR factorization for d_A;    R  stored in upper triangle of
// d_A, elementary reflectors vectors stored in lower triangle
// of d_A, elementary  reflectors  scalars stored in d_tau

  cusolver_status = cusolverDnDgeqrf(cusolverH, m, m, d_A, lda,
                     d_tau, d_work, lwork, devInfo);

  cudaStat1 = cudaDeviceSynchronize();           // stop timer
  clock_gettime(CLOCK_REALTIME,&stop);
  accum=(stop.tv_sec-start.tv_sec)+             // elapsed time
       (stop.tv_nsec-start.tv_nsec)/(double)BILLION;
  printf("Dgeqrf time: %lf\n",accum);    // print elapsed time

  cudaStat1 = cudaMemcpy(&info_gpu,devInfo,sizeof(int),
         cudaMemcpyDeviceToHost);       // devInfo -> info_gpu
// check error code of geqrf function
  printf("after geqrf: info_gpu = %d\n", info_gpu);
// compute d_B=Q^T*B using ormqr function

  cusolver_status=cusolverDnDormqr(cusolverH,CUBLAS_SIDE_LEFT,
  CUBLAS_OP_T, m, nrhs, m, d_A, lda, d_tau, d_B, ldb, d_work,
                       lwork, devInfo);

  cudaStat1 = cudaDeviceSynchronize();
  cudaStat1 = cudaMemcpy(&info_gpu, devInfo, sizeof(int),
         cudaMemcpyDeviceToHost);       // devInfo -> info_gpu
// check error code of ormqr function
  printf("after ormqr: info_gpu = %d\n", info_gpu);
// write the original system A*X=(Q*R)*X=B in the form
// R*X=Q^T*B and solve the obtained triangular system

  cublas_status = cublasDtrsm(cublasH,CUBLAS_SIDE_LEFT,
  CUBLAS_FILL_MODE_UPPER,CUBLAS_OP_N, CUBLAS_DIAG_NON_UNIT,
               m, nrhs, &one, d_A, lda, d_B, ldb);

  cudaStat1 = cudaDeviceSynchronize();
  cudaStat1 = cudaMemcpy(X,d_B,sizeof(double)*ldb*nrhs,
              cudaMemcpyDeviceToHost);          // copy d_B->X
  printf("solution: ");//show first components of the solution
  for (int i = 0; i < 5; i++) printf("%g, ", X[i]);
  printf(" ...");
  printf("\n");
// free memory
  cudaFree(d_A);
  cudaFree(d_tau);
  cudaFree(d_B);
  cudaFree(devInfo);
  cudaFree(d_work);
  cublasDestroy(cublasH);
  cusolverDnDestroy(cusolverH);
  cudaDeviceReset();
  return 0;
}
//Dgeqrf time: 3.333913 sec.
//after geqrf: info_gpu = 0
//after ormqr: info_gpu = 0
//solution: 1, 1, 1, 1, 1,  ...
```

3.3.8 cusolverDnDgeqrf and cusolverDnDormqr, cublasDtrsm - unified memory version

```
#include <cblas.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include <cublas_v2.h>
#include <cusolverDn.h>
#define BILLION 1000000000L;
int main(int argc, char*argv[])
{
  struct timespec start,stop;          // variables for timing
  double accum;                       // elapsed time variable
  cusolverDnHandle_t cusolverH;              // cusolver handle
  cublasHandle_t cublasH;                      // cublas handle
  cublasStatus_t cublas_status = CUBLAS_STATUS_SUCCESS;
  cusolverStatus_t cusolver_status = CUSOLVER_STATUS_SUCCESS;
  cudaError_t cudaStat1 = cudaSuccess;
  const int m = 8192;                       // number of rows of A
  const int lda = m;                     // leading dimension of A
  const int ldb = m;                     // leading dimension of B
  const int nrhs = 1;              // number of right hand sides
// A - mxm coeff. matr.,  B=A*B1 -right hand side, B1 - mxnrhs
  double *A, *B, *B1, *X;       // - auxil.matrix, X - solution
// prepare unified memory
  cudaMallocManaged(&A,lda*m*sizeof(double));//unif. mem.for A
  cudaMallocManaged(&B,ldb*nrhs*sizeof(double));//un.mem.for A
  cudaMallocManaged(&B1,ldb*nrhs*sizeof(double)); //mem.for B1
  for(int i=0;i<lda*m;i++) A[i]=rand()/(double)RAND_MAX;
  for(int i=0;i<ldb*nrhs;i++) B[i]=0.0;;
  for(int i=0;i<ldb*nrhs;i++) B1[i]=1.0;
  double al=1.0,bet=0.0;                    // constants for dgemv
  int incx=1, incy=1;
  cblas_dgemv(CblasColMajor,CblasNoTrans,m,m,al,A,m,B1,incx,
                                  bet,B,incy);        //B=A*B1
  double *tau, *work ;  //elem. reflectors scalars,  workspace
  int *Info ;                                          //  info
  int  lwork = 0;                               // workspace size
  const double one = 1;
// create cusolver and cublas handles
  cusolver_status = cusolverDnCreate(&cusolverH);
  cublas_status = cublasCreate(&cublasH);
  cudaMallocManaged(&tau,m*sizeof(double)); //unif.mem.for tau
  cudaMallocManaged(&Info,sizeof(int));    //unif.mem.for Info
// compute buffer size for geqrf and prepare workspace
  cusolver_status=cusolverDnDgeqrf_bufferSize(cusolverH, m, m,
                                A, lda, &lwork);
  cudaMallocManaged(&work,lwork*sizeof(double));//mem.for work
  clock_gettime(CLOCK_REALTIME,&start);          // start timer
// QR factorization for A;    R  stored in upper triangle of A
// elementary reflectors vectors stored in lower triangle of A
// elementary  reflectors  scalars stored in tau

  cusolver_status = cusolverDnDgeqrf(cusolverH, m, m, A, lda,
                         tau, work, lwork, Info);

  cudaStat1 = cudaDeviceSynchronize();            // stop timer
  clock_gettime(CLOCK_REALTIME,&stop);
  accum=(stop.tv_sec-start.tv_sec)+             // elapsed time
        (stop.tv_nsec-start.tv_nsec)/(double)BILLION;
  printf("Dgeqrf time: %lf sec.\n",accum);//print elapsed time
// check error code of geqrf function
  printf("after geqrf: info = %d\n", *Info);
// compute B=Q^T*B using ormqr function

  cusolver_status=cusolverDnDormqr(cusolverH,CUBLAS_SIDE_LEFT,
      CUBLAS_OP_T,m,nrhs,m,A,lda,tau,B,ldb,work,lwork,Info);

  cudaStat1 = cudaDeviceSynchronize();
// check error code of ormqr function
  printf("after ormqr: info = %d\n", *Info);
// write the original system A*X=(Q*R)*X=B in the form
// R*X=Q^T*B and solve the obtained triangular system

  cublas_status = cublasDtrsm(cublasH,CUBLAS_SIDE_LEFT,
  CUBLAS_FILL_MODE_UPPER,CUBLAS_OP_N, CUBLAS_DIAG_NON_UNIT,
                      m, nrhs, &one, A, lda, B, ldb);

  cudaStat1 = cudaDeviceSynchronize();
  printf("solution: ");//show first components of the solution
  for (int i = 0; i < 5; i++) printf("%g, ", B[i]);
  printf(" ...");
  printf("\n");
// free memory
  cudaFree(A);
  cudaFree(tau);
  cudaFree(B);
  cudaFree(B1);
  cudaFree(Info);
  cudaFree(work);
  cublasDestroy(cublasH);
  cusolverDnDestroy(cusolverH);
  cudaDeviceReset();
  return 0;
}
//Dgeqrf time: 3.386223 sec.
//after geqrf: info = 0
//after ormqr: info = 0
//solution: 1, 1, 1, 1, 1,  ...
```

3.4 Cholesky decomposition and solving positive definite linear systems

3.4.1 `cusolverDnSpotrf` and `cusolverDnSpotrs` - Choleski decomposition and solving positive definite systems in single precision

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

$$A = \begin{cases} U^T\,U & \text{in } \texttt{CUBLAS\_FILL\_MODE\_UPPER} \text{ case,} \\ L\,L^T & \text{in } \texttt{CUBLAS\_FILL\_MODE\_LOWER} \text{ case,} \end{cases}$$

where U is an upper triangular matrix and L is a lower triangular matrix. Using the obtained factorization the function `cusolverDnSpotrs` computes in single precision the solution of the linear system

$$A\,X = B,$$

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

```
#include <time.h>
#include <cblas.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <cuda_runtime.h>
#include <cusolverDn.h>
#define N 8192
#define BILLION 1000000000L;
using namespace std;

int main(int argc, char*argv[]){
  struct timespec start,stop;          // variables for timing
  double accum;                          // elapsed time variable
  float *A, *B, *B1;                 // declare arrays on the host
// prepare memory on the host
  A = (float*)malloc(N*N*sizeof(float));  // NxN coeff. matrix
  B = (float*)malloc(N*sizeof(float));  // N-vector rhs B=A*B1
  B1 = (float*)malloc(N*sizeof(float));  //  auxiliary N-vect.
  for(int i=0;i<N*N;i++) A[i] = rand()/(float)RAND_MAX;
  for(int i=0;i<N;i++) B[i] = 0.0;
  for(int i=0;i<N;i++) B1[i] = 1.0;        // N-vector of ones
  for(int i=0;i<N;i++){
     A[i*N+i]=A[i*N+i]+(float)N;   // make A positive definite
     for(int j=0;j<i;j++) A[i*N+j]=A[j*N+i];   //and symmetric
  }
  float al=1.0,bet=0.0;                  // constants for sgemv
  int incx=1, incy=1;
  cblas_sgemv(CblasColMajor,CblasNoTrans,N,N,al,A,N,B1,incx,
                      bet,B,incy);                    // B=A*B1
  cudaError cudaStatus;
  cusolverStatus_t cusolverStatus;
  cusolverDnHandle_t handle;               // device versions of
  float *d_A, *d_B, *Work;       // matrix A, rhs B and worksp.
  int  *d_info, Lwork;  // device version of info, worksp.size
  int info_gpu = 0;              // device info copied to host
  cudaStatus = cudaGetDevice(0);
  cusolverStatus = cusolverDnCreate(&handle); // create handle
  cublasFillMode_t uplo = CUBLAS_FILL_MODE_LOWER;
// prepare memory on the device
  cudaStatus = cudaMalloc((void**)&d_A, N*N*sizeof(float));
  cudaStatus = cudaMalloc((void**)&d_B, N*sizeof(float));
  cudaStatus = cudaMalloc((void**)&d_info, sizeof(int));
  cudaStatus = cudaMemcpy(d_A, A, N*N*sizeof(float),
                     cudaMemcpyHostToDevice);    // copy A->d_A
  cudaStatus = cudaMemcpy(d_B, B, N*sizeof(float),
                     cudaMemcpyHostToDevice);    // copy B->d_B
// compute workspace size and prepare workspace
  cusolverStatus = cusolverDnSpotrf_bufferSize(handle,
                    uplo,N,d_A,N,&Lwork );
  cudaStatus = cudaMalloc((void**)&Work,Lwork*sizeof(float));
  clock_gettime(CLOCK_REALTIME,&start);          // start timer
// Cholesky decomposition  d_A=L*L^T, lower triangle of d_A is
// replaced by the factor L

  cusolverStatus = cusolverDnSpotrf(handle,uplo,N,d_A,N,Work,
                             Lwork,d_info);

// solve d_A*X=d_B,  where d_A is factorized by potrf function
// d_B is overwritten by the solution

  cusolverStatus = cusolverDnSpotrs(handle,uplo,N, 1,d_A,N,
                             d_B,N,d_info);

  cudaStatus = cudaDeviceSynchronize();
  clock_gettime(CLOCK_REALTIME,&stop);           // stop timer
  accum=(stop.tv_sec-start.tv_sec)+            // elapsed time
        (stop.tv_nsec-start.tv_nsec)/(double)BILLION;
  printf("Spotrf+Spotrs time: %lf sec.\n",accum); //pr.el.time
  cudaStatus = cudaMemcpy(&info_gpu, d_info, sizeof(int),
         cudaMemcpyDeviceToHost);   // copy d_info -> info_gpu
  printf("after Spotrf+Spotrs: info_gpu = %d\n", info_gpu);
  cudaStatus = cudaMemcpy(B, d_B, N*sizeof(float),
     cudaMemcpyDeviceToHost);  // copy solution to host d_B->B
  printf("solution: ");
  for (int i = 0; i < 5; i++)  printf("%g, ", B[i]);  // print
  printf(" ...");          // first components of the solution
  printf("\n");
// free memory
  cudaStatus = cudaFree(d_A);
  cudaStatus = cudaFree(d_B);
  cudaStatus = cudaFree(d_info);
  cudaStatus = cudaFree(Work);
  cusolverStatus = cusolverDnDestroy(handle);
  cudaStatus = cudaDeviceReset();
  return 0;
}
//Spotrf+Spotrs time: 0.057328
//after Spotrf+Spotrs: info_gpu = 0
//solution: 1, 1, 1, 0.999999, 1,  ...
```

3.4.2 cusolverDnSpotrf and cusolverDnSpotrs - unified memory version

```
#include <time.h>
#include <cblas.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <cuda_runtime.h>
#include <cusolverDn.h>
#define N 8192
#define BILLION 1000000000L;
using namespace std;

int main(int argc, char*argv[]){
  struct timespec start,stop;          // variables for timing
  double accum;                       // elapsed time variable
  float *A, *B, *B1;                          // declare arrays
// prepare unified memory
  cudaMallocManaged(&A,N*N*sizeof(float));// unified mem.for A
  cudaMallocManaged(&B,N*sizeof(float));  // unified mem.for B
  cudaMallocManaged(&B1,N*sizeof(float));// unified mem.for B1
  for(int i=0;i<N*N;i++) A[i] = rand()/(float)RAND_MAX;
  for(int i=0;i<N;i++) B[i] = 0.0;
  for(int i=0;i<N;i++) B1[i] = 1.0;          // N-vector of ones
  for(int i=0;i<N;i++){
     A[i*N+i]=A[i*N+i]+(float)N;    // make A positive definite
     for(int j=0;j<i;j++) A[i*N+j]=A[j*N+i];   //and symmetric
  }
  float al=1.0,bet=0.0;                       // constants for sgemv
  int incx=1, incy=1;
  cblas_sgemv(CblasColMajor,CblasNoTrans,N,N,al,A,N,B1,incx,
                       bet,B,incy);                      // B=A*B1
  cudaError cudaStatus;
  cusolverStatus_t cusolverStatus;
  cusolverDnHandle_t handle;
  float  *Work;                                         //  workspace
  int  *info, Lwork;                        //  info, workspace size
  cudaStatus = cudaGetDevice(0);
  cusolverStatus = cusolverDnCreate(&handle); // create handle
  cublasFillMode_t uplo = CUBLAS_FILL_MODE_LOWER;
  cudaMallocManaged(&info,sizeof(int));//unified mem. for info
// compute workspace size and prepare workspace
  cusolverStatus = cusolverDnSpotrf_bufferSize(handle,
                     uplo,N,A,N,&Lwork );
  cudaMallocManaged(&Work,Lwork*sizeof(float)); //mem.for Work
  clock_gettime(CLOCK_REALTIME,&start);          // start timer
// Cholesky decomposition  d_A=L*L^T, lower triangle of d_A is
// replaced by the factor L

  cusolverStatus = cusolverDnSpotrf(handle,uplo,N,A,N,Work,
                         Lwork,info);

  cudaStatus = cudaDeviceSynchronize();
// solve A*X=B,  where A is factorized by potrf function
// B is overwritten by the solution

  cusolverStatus = cusolverDnSpotrs(handle,uplo,N,1,A,N,
                         B,N,info);

  cudaStatus = cudaDeviceSynchronize();
  clock_gettime(CLOCK_REALTIME,&stop);            // stop timer
  accum=(stop.tv_sec-start.tv_sec)+             // elapsed time
        (stop.tv_nsec-start.tv_nsec)/(double)BILLION;
  printf("Spotrf+Spotrs time: %lf sec.\n",accum); //pr.el.time
  printf("after Spotrf+Spotrs: info = %d\n", *info);
  printf("solution: ");
  for (int i = 0; i < 5; i++)  printf("%g, ", B[i]);  // print
  printf(" ...");          // first components of the solution
  printf("\n");
// free memory
  cudaStatus = cudaFree(A);
  cudaStatus = cudaFree(B);
  cudaStatus = cudaFree(B1);
  cudaStatus = cudaFree(info);
  cudaStatus = cudaFree(Work);
  cusolverStatus = cusolverDnDestroy(handle);
  cudaStatus = cudaDeviceReset();
  return 0;
}
//Spotrf+Spotrs time: 0.094803 sec.
//after Spotrf+Spotrs: info = 0
//solution: 1, 1, 1, 0.999999, 1,  ...
```

3.4.3 `cusolverDnDpotrf` and `cusolverDnDpotrs` - Choleski decomposition and solving positive definite systems in double precision

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

$$A = \begin{cases} U^T\,U & \text{in } \texttt{CUBLAS\_FILL\_MODE\_UPPER} \text{ case,} \\ L\,L^T & \text{in } \texttt{CUBLAS\_FILL\_MODE\_LOWER} \text{ case,} \end{cases}$$

where U is an upper triangular matrix and L is a lower triangular matrix. Using the obtained factorization the function `cusolverDnDpotrs` computes in double precision the solution of the linear system

$$A\,X = B,$$

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

```
#include <time.h>
#include <cblas.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <cuda_runtime.h>
#include <cusolverDn.h>
#define N 8192
#define BILLION 1000000000L;
using namespace std;

int main(int argc, char*argv[]){
  struct timespec start,stop;          // variables for timing
  double accum;                        // elapsed time variable
  double *A, *B, *B1;             // declare arrays on the host
// prepare memory on the host
  A = (double*)malloc(N*N*sizeof(double));// NxN coeff. matrix
  B = (double*)malloc(N*sizeof(double));// N-vector rhs B=A*B1
  B1 = (double*)malloc(N*sizeof(double));//  auxiliary N-vect.
  for(int i=0;i<N*N;i++) A[i] = rand()/(double)RAND_MAX;
  for(int i=0;i<N;i++) B[i] = 0.0;
  for(int i=0;i<N;i++) B1[i] = 1.0;          // N-vector of ones
  for(int i=0;i<N;i++){
     A[i*N+i]=A[i*N+i]+(double)N;  // make A positive definite
     for(int j=0;j<i;j++) A[i*N+j]=A[j*N+i];   //and symmetric
  }
  double al=1.0,bet=0.0;                 // constants for dgemv
  int incx=1, incy=1;
  cblas_dgemv(CblasColMajor,CblasNoTrans,N,N,al,A,N,B1,incx,
                         bet,B,incy);                  // B=A*B1
  cudaError cudaStatus;
  cusolverStatus_t cusolverStatus;
  cusolverDnHandle_t handle;              // device versions of
  double *d_A, *d_B, *Work;      // matrix A, rhs B and worksp.
  int  *d_info, Lwork;  // device version of info, worksp.size
  int info_gpu = 0;               // device info copied to host
  cudaStatus = cudaGetDevice(0);
  cusolverStatus = cusolverDnCreate(&handle); // create handle
  cublasFillMode_t uplo = CUBLAS_FILL_MODE_LOWER;
// prepare memory on the device
  cudaStatus = cudaMalloc((void**)&d_A, N*N*sizeof(double));
  cudaStatus = cudaMalloc((void**)&d_B, N*sizeof(double));
  cudaStatus = cudaMalloc((void**)&d_info, sizeof(int));
  cudaStatus = cudaMemcpy(d_A, A, N*N*sizeof(double),
                       cudaMemcpyHostToDevice);    // copy A->d_A
  cudaStatus = cudaMemcpy(d_B, B, N*sizeof(double),
                       cudaMemcpyHostToDevice);    // copy B->d_B
// compute workspace size and prepare workspace
  cusolverStatus = cusolverDnDpotrf_bufferSize(handle,
                      uplo,N,d_A,N,&Lwork );
  cudaStatus = cudaMalloc((void**)&Work,Lwork*sizeof(double));
  clock_gettime(CLOCK_REALTIME,&start);          // start timer
// Cholesky decomposition  d_A=L*L^T, lower triangle of d_A is
// replaced by the factor L

  cusolverStatus = cusolverDnDpotrf(handle,uplo,N,d_A,N,Work,
                         Lwork,d_info);

// solve d_A*X=d_B,  where d_A is factorized by potrf function
// d_B is overwritten by the solution

  cusolverStatus = cusolverDnDpotrs(handle, uplo,N, 1,d_A, N,
                         d_B,N,d_info);

  cudaStatus = cudaDeviceSynchronize();
  clock_gettime(CLOCK_REALTIME,&stop);            // stop timer
  accum=(stop.tv_sec-start.tv_sec)+             // elapsed time
        (stop.tv_nsec-start.tv_nsec)/(double)BILLION;
  printf("solution: ");
  printf("Dpotrf+Dpotrs time: %lf sec.\n",accum); //pr.el.time
  cudaStatus = cudaMemcpy(&info_gpu, d_info, sizeof(int),
         cudaMemcpyDeviceToHost);   // copy d_info -> info_gpu
  printf("after Dpotrf+Dpotrs: info_gpu = %d\n", info_gpu);
  cudaStatus = cudaMemcpy(B, d_B, N*sizeof(double),
     cudaMemcpyDeviceToHost);  // copy solution to host d_B->B
```

```
  for (int i = 0; i < 5; i++)  printf("%g, ", B[i]);  // print
  printf(" ...");            // first components of the solution
  printf("\n");
// free memory
  cudaStatus = cudaFree(d_A);
  cudaStatus = cudaFree(d_B);
  cudaStatus = cudaFree(d_info);
  cudaStatus = cudaFree(Work);
  cusolverStatus = cusolverDnDestroy(handle);
  cudaStatus = cudaDeviceReset();
  return 0;
}

//Dpotrf+Dpotrs time: 0.754091 sec.
//after potrf: info_gpu = 0
//solution: 1, 1, 1, 1, 1,  ...
```

3.4.4 cusolverDnDpotrf and cusolverDnDpotrs - unified memory version

```
#include <time.h>
#include <cblas.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <cuda_runtime.h>
#include <cusolverDn.h>
#define N 8192
#define BILLION 1000000000L;
using namespace std;

int main(int argc, char*argv[]){
  struct timespec start,stop;           // variables for timing
  double accum;                        // elapsed time variable
  double *A, *B, *B1;                            // declare arrays
// prepare unified memory
  cudaMallocManaged(&A,N*N*sizeof(double));//unified mem.for A
  cudaMallocManaged(&B,N*sizeof(double));  //unified mem.for B
  cudaMallocManaged(&B1,N*sizeof(double));//unified mem.for B1
  for(int i=0;i<N*N;i++) A[i] = rand()/(double)RAND_MAX;
  for(int i=0;i<N;i++) B[i] = 0.0;
  for(int i=0;i<N;i++) B1[i] = 1.0;         // N-vector of ones
  for(int i=0;i<N;i++){
     A[i*N+i]=A[i*N+i]+(double)N;  // make A positive definite
     for(int j=0;j<i;j++) A[i*N+j]=A[j*N+i];   //and symmetric
  }
  double al=1.0,bet=0.0;                  // constants for dgemv
  int incx=1, incy=1;
  cblas_dgemv(CblasColMajor,CblasNoTrans,N,N,al,A,N,B1,incx,
                      bet,B,incy);                    // B=A*B1
  cudaError cudaStatus;
  cusolverStatus_t cusolverStatus;
  cusolverDnHandle_t handle;
  double  *Work;                                   //  workspace
  int  *info, Lwork;                        //  info, workspace size
  cudaStatus = cudaGetDevice(0);
  cusolverStatus = cusolverDnCreate(&handle); // create handle
  cublasFillMode_t uplo = CUBLAS_FILL_MODE_LOWER;
  cudaMallocManaged(&info,sizeof(int));//unified mem. for info
// compute workspace size and prepare workspace
  cusolverStatus = cusolverDnDpotrf_bufferSize(handle,
                      uplo,N,A,N,&Lwork );
  cudaMallocManaged(&Work,Lwork*sizeof(double));//mem.for Work
  clock_gettime(CLOCK_REALTIME,&start);         // start timer
// Cholesky decomposition  d_A=L*L^T, lower triangle of d_A is
// replaced by the factor L

  cusolverStatus = cusolverDnDpotrf(handle,uplo,N,A,N,Work,
                           Lwork,info);

  cudaStatus = cudaDeviceSynchronize();
// solve A*X=B,  where A is factorized by potrf function
// B is overwritten by the solution

  cusolverStatus = cusolverDnDpotrs(handle,uplo,N,1,A,N,
                           B,N,info);

  cudaStatus = cudaDeviceSynchronize();
  clock_gettime(CLOCK_REALTIME,&stop);           // stop timer
  accum=(stop.tv_sec-start.tv_sec)+             // elapsed time
       (stop.tv_nsec-start.tv_nsec)/(double)BILLION;
  printf("Dpotrf+Dpotrs time: %lf sec.\n",accum); //pr.el.time
  printf("after Dpotrf+Dpotrs: info = %d\n", *info);
  printf("solution: ");
  for (int i = 0; i < 5; i++)  printf("%g, ", B[i]);  // print
  printf(" ...");          // first components of the solution
  printf("\n");
// free memory
  cudaStatus = cudaFree(A);
  cudaStatus = cudaFree(B);
  cudaStatus = cudaFree(B1);
  cudaStatus = cudaFree(info);
  cudaStatus = cudaFree(Work);
  cusolverStatus = cusolverDnDestroy(handle);
  cudaStatus = cudaDeviceReset();
  return 0;
}

//Dpotrf+Dpotrs time: 0.807432 sec.
//after Dpotrf+Dpotrs: info = 0
//solution: 1, 1, 1, 1, 1,  ...
```

3.5 Bunch-Kaufman decomposition and solving symmetric linear systems

3.5.1 `cusolverDnSsytrf` and `ssytrs` - Bunch-Kaufman decomposition and solving symmetric systems in single precision

The function `cusolverDnSsytrf` computes Bunch-Kaufman factorization of a symmetric indefinite matrix

$$A = L * D * L^T,$$

where D is symmetric, block-diagonal, with 1×1 or 2×2 blocks, L is a product of permutation and triangular matrices. The function `ssytrs` solves the system $A * X = B$, where $A = L * D * L^T$ is the matrix factored using Bunch-Kaufman method, B is overwritten by the solution.

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <cuda_runtime.h>
#include <cusolverDn.h>
#include <mkl.h>
#define BILLION 1000000000L;
using namespace std;
int main(int argc, char*argv[]){
  int N=8192;
  int nrhs=1;
  struct timespec start,stop;          // variables for timing
  double accum;                        // elapsed time variable
  float *A;                            // NxN coefficient matrix
  float *B, *B1;                       // N-vectors, rhs B=A*B1
// prepare memory on the host
  A = (float*)malloc(N*N*sizeof(float));
  B1 = (float*)malloc(N*sizeof(float));
  B = (float*)malloc(N*sizeof(float));
  for(int i=0;i<N*N;i++) A[i] = rand()/(float)RAND_MAX;
  for(int i=0;i<N;i++) B[i] = 0.0;
  for(int i=0;i<N;i++) B1[i] = 1.0;         // N-vector of ones
  for(int i=0;i<N;i++){                     // make A symmetric
    for(int j=0;j<i;j++) A[i*N+j]=A[j*N+i];
  }
  float al=1.0,bet=0.0;                     // constants for sgemv
  int incx=1, incy=1;
  const char tr='N';
  sgemv(&tr,&N,&N,&al,A,&N,B1,&incx,&bet,B,&incy);   // B=A*B1
  cudaError cudaStatus;
  cusolverStatus_t cusolverStatus;
  cusolverDnHandle_t handle;
  float *d_A, *Work;  // coeff. matrix and workspace on device
  int *d_pivot, *d_info, Lwork;   // pivots and info on device
  int *piv, info;                // pivots and info on the host
  int info_gpu = 0;               // device info copied to host
  cudaStatus = cudaGetDevice(0);
  cusolverStatus = cusolverDnCreate(&handle); // create handle
  cublasFillMode_t uplo=CUBLAS_FILL_MODE_LOWER;
  const char upl='L';          //use lower triangular part of A
// prepare memory on the device
  cudaStatus = cudaMalloc((void**)&d_A, N*N*sizeof(float));
  cudaStatus = cudaMalloc((void**)&d_pivot, N*sizeof(int));
  cudaStatus = cudaMalloc((void**)&d_info, sizeof(int));
  cudaStatus = cudaMemcpy(d_A, A, N*N*sizeof(float),
               cudaMemcpyHostToDevice);        // copy A->d_A
  piv=(int*)malloc(N*sizeof(int));
  cusolverStatus=cusolverDnSsytrf_bufferSize(handle,N,d_A,N,
        &Lwork );    // compute buffer size and prepare memory
  cudaStatus = cudaMalloc((void**)&Work, Lwork*sizeof(float));
  clock_gettime(CLOCK_REALTIME,&start);          // start timer
// Bunch-Kaufman factorization of an NxN symmetric indefinite
// matrix  d_A=L*D*L^T,  where D is symmetric, block-diagonal,
// with 1x1 or 2x2 blocks,  L is a product of permutation  and
// triangular matrices

  cusolverStatus = cusolverDnSsytrf(handle,uplo,N,d_A,N,d_pivot,
                         Work,Lwork,d_info );

  cudaStatus = cudaDeviceSynchronize();
  cudaStatus = cudaMemcpy(A, d_A, N*N*sizeof(float),
               cudaMemcpyDeviceToHost);          // copy d_A->A
  cudaStatus = cudaMemcpy(piv,d_pivot , N*sizeof(int),
               cudaMemcpyDeviceToHost);   // copy d_pivot->piv
// solve the system A*X=B , where A=L*D*L^T - symmetric
// coefficient matrix factored using Bunch-Kaufman method,
// B is overwritten by the solution

  ssytrs(&upl,&N,&nrhs,A,&N,piv,B,&N,&info);

  clock_gettime(CLOCK_REALTIME,&stop);            // stop timer
  accum=(stop.tv_sec-start.tv_sec)+             // elapsed time
        (stop.tv_nsec-start.tv_nsec)/(double)BILLION;
  printf("Ssytrf+ssytrs time: %lf sec.\n",accum); //pr.el.time
  cudaStatus = cudaMemcpy(&info_gpu, d_info, sizeof(int),
             cudaMemcpyDeviceToHost); // copy d_info->info_gpu
  printf("after Sytrf: info_gpu = %d\n", info_gpu);
  printf("solution: ");
  for (int i = 0; i < 5; i++) printf("%g, ", B[i]);
  printf("...\n");         // first components of the solution
// free memory
  cudaStatus = cudaFree(d_A);
  cudaStatus = cudaFree(d_pivot);
  cudaStatus = cudaFree(d_info);
  cudaStatus = cudaFree(Work);
  cusolverStatus = cusolverDnDestroy(handle);
  cudaStatus = cudaDeviceReset();
  return 0;
}
//Ssytrf+Ssytrs time: 0.397637 sec.
//after Sytrf: info_gpu = 0
//solution: 1.01025, 1.0031, 0.994385, 1.00684, 0.986153, ...
```

3.5.2 cusolverDnSsytrf and ssytrs - unified memory version

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <cuda_runtime.h>
#include <cusolverDn.h>
#include <mkl.h>
#define BILLION 1000000000L;
using namespace std;
int main(int argc, char*argv[]){
  int N=8192;
  int nrhs=1;
  struct timespec start,stop;          // variables for timing
  double accum;                        // elapsed time variable
  float *A;                           // NxN coefficient matrix
  float *B, *B1;                       // N-vectors, rhs B=A*B1
// prepare unified memory
  cudaMallocManaged(&A,N*N*sizeof(float)); //unified mem.for A
  cudaMallocManaged(&B,N*sizeof(float));   //unified mem.for B
  cudaMallocManaged(&B1,N*sizeof(float)); //unified mem.for B1
  for(int i=0;i<N*N;i++) A[i] = rand()/(float)RAND_MAX;
  for(int i=0;i<N;i++) B[i] = 0.0;
  for(int i=0;i<N;i++) B1[i] = 1.0;         // N-vector of ones
  for(int i=0;i<N;i++){                     // make A symmetric
    for(int j=0;j<i;j++) A[i*N+j]=A[j*N+i];
  }
  float al=1.0,bet=0.0;                   // constants for sgemv
  int incx=1, incy=1;
  const char tr='N';
  sgemv(&tr,&N,&N,&al,A,&N,B1,&incx,&bet,B,&incy);    // B=A*B1
  cudaError cudaStatus;
  cusolverStatus_t cusolverStatus;
  cusolverDnHandle_t handle;
  float  *Work;                                       // workspace
  int *pivot, *info, Lwork;    // pivots, info, workspace size
  cudaStatus = cudaGetDevice(0);
  cusolverStatus = cusolverDnCreate(&handle); // create handle
  cublasFillMode_t uplo=CUBLAS_FILL_MODE_LOWER;
  const char upl='L';         //use lower triangular part of A
  cudaMallocManaged(&pivot,N*sizeof(int));//unif.mem.for pivot
  cudaMallocManaged(&info,sizeof(int));    //unif.mem.for info
  cusolverStatus=cusolverDnSsytrf_bufferSize(handle,N,A,N,
        &Lwork );    // compute buffer size and prepare memory
  cudaMallocManaged(&Work,Lwork*sizeof(float)); //mem.for Work
  clock_gettime(CLOCK_REALTIME,&start);         // start timer
// Bunch-Kaufman factorization of an NxN symmetric indefinite
// matrix  A=L*D*L^T,  where D is symmetric, block-diagonal,
// with 1x1 or 2x2 blocks,  L is a product of permutation  and
// triangular matrices

  cusolverStatus = cusolverDnSsytrf(handle,uplo,N,A,N, pivot,
                          Work,Lwork, info );

  cudaStatus = cudaDeviceSynchronize();
// solve the system A*X=B , where A=L*D*L^T - symmetric
// coefficient matrix factored using Bunch-Kaufman method,
// B is overwritten by the solution

  ssytrs(&upl,&N,&nrhs,A,&N,pivot,B,&N,info);

  clock_gettime(CLOCK_REALTIME,&stop);           // stop timer
  accum=(stop.tv_sec-start.tv_sec)+            // elapsed time
        (stop.tv_nsec-start.tv_nsec)/(double)BILLION;
  printf("Ssytrf+ssytrs time: %lf sec.\n",accum); //pr.el.time
  printf("after Ssytrf: info = %d\n", *info);
  printf("solution: ");
  for (int i = 0; i < 5; i++) printf("%g, ", B[i]);
  printf("...\n");          // first components of the solution
// free memory
  cudaStatus = cudaFree(A);
  cudaStatus = cudaFree(B);
  cudaStatus = cudaFree(B1);
  cudaStatus = cudaFree(pivot);
  cudaStatus = cudaFree(info);
  cudaStatus = cudaFree(Work);
  cusolverStatus = cusolverDnDestroy(handle);
  cudaStatus = cudaDeviceReset();
  return 0;
}

//Ssytrf+ssytrs time: 0.544929 sec.
//after Ssytrf: info = 0
//solution: 1.01025 1.0031, 0.994385, 1.00684, 0.986153, ...
```

3.5.3 `cusolverDnDsytrf` and `dsytrs` - Bunch-Kaufman decomposition and solving symmetric systems in double precision

The function `cusolverDnDsytrf` computes Bunch-Kaufman factorization of a symmetric indefinite matrix

$$A = L * D * L^T,$$

where D is symmetric, block-diagonal, with 1×1 or 2×2 blocks, L is a product of permutation and triangular matrices. The function `dsytrs` solves the system $A * X = B$, where $A = L * D * L^T$ is the matrix factored using Bunch-Kaufman method, B is overwritten by the solution.

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <cuda_runtime.h>
#include <cusolverDn.h>
#include <mkl.h>
#define BILLION 1000000000L;
using namespace std;
int main(int argc, char*argv[]){
  int N=8192;
  int nrhs=1;
  struct timespec start,stop;          // variables for timing
  double accum;                        // elapsed time variable
  double *A;                          // NxN coefficient matrix
  double *B, *B1;                      // N-vectors, rhs B=A*B1
// prepare memory on the host
  A = (double*)malloc(N*N*sizeof(double));
  B1 = (double*)malloc(N*sizeof(double));
  B = (double*)malloc(N*sizeof(double));
  for(int i=0;i<N*N;i++) A[i] = rand()/(double)RAND_MAX;
  for(int i=0;i<N;i++) B[i] = 0.0;
  for(int i=0;i<N;i++) B1[i] = 1.0;         // N-vector of ones
  for(int i=0;i<N;i++){                     // make A symmetric
    for(int j=0;j<i;j++) A[i*N+j]=A[j*N+i];
  }
  double al=1.0,bet=0.0;                      // constants for dgemv
  int incx=1, incy=1;
  const char tr='N';
  dgemv(&tr,&N,&N,&al,A,&N,B1,&incx,&bet,B,&incy);   // B=A*B1
  cudaError cudaStatus;
  cusolverStatus_t cusolverStatus;
  cusolverDnHandle_t handle;
  double *d_A, *Work; // coeff. matrix and workspace on device
  int *d_pivot, *d_info, Lwork;   // pivots and info on device
  int *piv, info;                // pivots and info on the host
  int info_gpu = 0;               // device info copied to host
  cudaStatus = cudaGetDevice(0);
  cusolverStatus = cusolverDnCreate(&handle); // create handle
  cublasFillMode_t uplo=CUBLAS_FILL_MODE_LOWER;
  const char upl='L';          //use lower triangular part of A
// prepare memory on the device
  cudaStatus = cudaMalloc((void**)&d_A, N*N*sizeof(double));
  cudaStatus = cudaMalloc((void**)&d_pivot, N*sizeof(int));
  cudaStatus = cudaMalloc((void**)&d_info, sizeof(int));
  cudaStatus = cudaMemcpy(d_A, A, N*N*sizeof(double),
                 cudaMemcpyHostToDevice);        // copy A->d_A
  piv=(int*)malloc(N*sizeof(int));
  cusolverStatus=cusolverDnDsytrf_bufferSize(handle,N,d_A,N,
        &Lwork );    // compute buffer size and prepare memory
  cudaStatus = cudaMalloc((void**)&Work,Lwork*sizeof(double));
  clock_gettime(CLOCK_REALTIME,&start);         // start timer
// Bunch-Kaufman factorization of an NxN symmetric indefinite
// matrix  d_A=L*D*L^T,  where D is symmetric, block-diagonal,
// with 1x1 or 2x2 blocks,  L is a product of permutation  and
// triangular matrices

  cusolverStatus = cusolverDnDsytrf(handle,uplo,N,d_A,N,d_pivot,
                         Work,Lwork,d_info );

  cudaStatus = cudaDeviceSynchronize();
  cudaStatus = cudaMemcpy(A, d_A, N*N*sizeof(double),
               cudaMemcpyDeviceToHost);         // copy d_A->A
  cudaStatus = cudaMemcpy(piv,d_pivot , N*sizeof(int),
               cudaMemcpyDeviceToHost);   // copy d_pivot->piv
// solve the system A*X=B , where A=L*D*L^T - symmetric
// coefficient matrix factored using Bunch-Kaufman method,
// B is overwritten by the solution

  dsytrs(&upl,&N,&nrhs,A,&N,piv,B,&N,&info);

  clock_gettime(CLOCK_REALTIME,&stop);           // stop timer
  accum=(stop.tv_sec-start.tv_sec)+            // elapsed time
        (stop.tv_nsec-start.tv_nsec)/(double)BILLION;
  printf("Dsytrf+dsytrs time: %lf sec.\n",accum); //pr.el.time
  cudaStatus = cudaMemcpy(&info_gpu, d_info, sizeof(int),
             cudaMemcpyDeviceToHost); // copy d_info->info_gpu
  printf("after Dsytrf: info_gpu = %d\n", info_gpu);
  printf("solution: ");
  for (int i = 0; i < 5; i++) printf("%g, ", B[i]);
  printf("...\n");          // first components of the solution
// free memory
  cudaStatus = cudaFree(d_A);
  cudaStatus = cudaFree(d_pivot);
  cudaStatus = cudaFree(d_info);
  cudaStatus = cudaFree(Work);
  cusolverStatus = cusolverDnDestroy(handle);
  cudaStatus = cudaDeviceReset();
  return 0;
}
//Dsytrf+dsytrs time: 1.173202 sec.
//after Dsytrf: info_gpu = 0
//solution: 1, 1, 1, 1, 1, ...
```

3.5.4 `cusolverDnDsytrf` and `dsytrs` - unified memory version

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <cuda_runtime.h>
#include <cusolverDn.h>
#include <mkl.h>
#define BILLION 1000000000L;
using namespace std;
int main(int argc, char*argv[]){
  int N=8192;
  int nrhs=1;
  struct timespec start,stop;          // variables for timing
  double accum;                        // elapsed time variable
  double *A;                          // NxN coefficient matrix
  double *B, *B1;                      // N-vectors, rhs B=A*B1
// prepare unified memory
  cudaMallocManaged(&A,N*N*sizeof(double));//unified mem.for A
  cudaMallocManaged(&B,N*sizeof(double));  //unified mem.for B
  cudaMallocManaged(&B1,N*sizeof(double));//unified mem.for B1
  for(int i=0;i<N*N;i++) A[i] = rand()/(double)RAND_MAX;
  for(int i=0;i<N;i++) B[i] = 0.0;
  for(int i=0;i<N;i++) B1[i] = 1.0;        // N-vector of ones
  for(int i=0;i<N;i++){                    // make A symmetric
    for(int j=0;j<i;j++) A[i*N+j]=A[j*N+i];
  }
  double al=1.0,bet=0.0;                  // constants for dgemv
  int incx=1, incy=1;
  const char tr='N';
  dgemv(&tr,&N,&N,&al,A,&N,B1,&incx,&bet,B,&incy);   // B=A*B1
  cudaError cudaStatus;
  cusolverStatus_t cusolverStatus;
  cusolverDnHandle_t handle;
  double  *Work;                                     // workspace
  int *pivot, *info, Lwork;    // pivots, info, workspace size
  cudaStatus = cudaGetDevice(0);
  cusolverStatus = cusolverDnCreate(&handle); // create handle
  cublasFillMode_t uplo=CUBLAS_FILL_MODE_LOWER;
  const char upl='L';         //use lower triangular part of A
  cudaMallocManaged(&pivot,N*sizeof(int));//unif.mem.for pivot
  cudaMallocManaged(&info,sizeof(int));    //unif.mem.for info
  cusolverStatus=cusolverDnDsytrf_bufferSize(handle,N,A,N,
        &Lwork );    // compute buffer size and prepare memory
  cudaMallocManaged(&Work,Lwork*sizeof(double));//mem.for Work
  clock_gettime(CLOCK_REALTIME,&start);         // start timer
// Bunch-Kaufman factorization of an NxN symmetric indefinite
// matrix  A=L*D*L^T,  where D is symmetric, block-diagonal,
// with 1x1 or 2x2 blocks,  L is a product of permutation  and
// triangular matrices
```

```
  cusolverStatus = cusolverDnDsytrf(handle,uplo,N,A,N,pivot,
                         Work,Lwork, info );

  cudaStatus = cudaDeviceSynchronize();
// solve the system A*X=B , where A=L*D*L^T - symmetric
// coefficient matrix factored using Bunch-Kaufman method,
// B is overwritten by the solution

  dsytrs(&upl,&N,&nrhs,A,&N,pivot,B,&N,info);

  clock_gettime(CLOCK_REALTIME,&stop);           // stop timer
  accum=(stop.tv_sec-start.tv_sec)+             // elapsed time
        (stop.tv_nsec-start.tv_nsec)/(double)BILLION;
  printf("Dsytrf+dsytrs time: %lf sec.\n",accum); //pr.el.time
  printf("after Dsytrf: info = %d\n", *info);
  printf("solution: ");
  for (int i = 0; i < 5; i++) printf("%g, ", B[i]);
  printf("...\n");          // first components of the solution
// free memory
  cudaStatus = cudaFree(A);
  cudaStatus = cudaFree(B);
  cudaStatus = cudaFree(B1);
  cudaStatus = cudaFree(pivot);
  cudaStatus = cudaFree(info);
  cudaStatus = cudaFree(Work);
  cusolverStatus = cusolverDnDestroy(handle);
  cudaStatus = cudaDeviceReset();
  return 0;
}
//Dsytrf+dsytrs time: 1.279214 sec.
//after Dsytrf: info = 0
//solution: 1, 1, 1, 1, 1, ...
```

3.6 SVD decomposition

3.6.1 `cusolverDnSgesvd` - SVD decomposition in single precision

This function computes in single precision the singular value decomposition of an $m \times n$ matrix:

$$A = u\,\sigma\,v^T,$$

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 respectively.

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>
#include <cuda_runtime.h>
#include <cublas_v2.h>
#include <cusolverDn.h>
#define BILLION 1000000000L;
int main(int argc, char*argv[])
{
  struct timespec start,stop;          // variables for timing
  double accum;                         // elapsed time variable
  cusolverDnHandle_t cusolverH;              // cusolver handle
  cublasHandle_t cublasH;                      // cublas handle
  cublasStatus_t cublas_status = CUBLAS_STATUS_SUCCESS;
  cusolverStatus_t cusolver_status = CUSOLVER_STATUS_SUCCESS;
  cudaError_t cudaStat = cudaSuccess;
  const int m = 2048;                       // number of rows of A
  const int n = 2048;                    // number of columns of A
  const int lda = m;                     // leading dimension of A
// declare the factorized matrix A, orthogonal matrices  U, VT
  float *A, *U, *VT, *S; // and sing.val. matrix S on the host
  A=(float*)malloc(lda*n*sizeof(float));
  U=(float*)malloc(lda*m*sizeof(float));
  VT=(float*)malloc(lda*n*sizeof(float));
  S= (float*)malloc(n*sizeof(float));
  for(int i=0;i<lda*n;i++) A[i]=rand()/(float)RAND_MAX;
// the factorized matrix d_A, orthogonal matrices  d_U, d_VT
  float *d_A, *d_U, *d_VT, *d_S;   // and sing.val. matrix d_S
  int *devInfo;                                // on the device
  float *d_work, *d_rwork ;          // workspace on the device
  float *d_W;      // auxiliary device array   (d_W = d_S*d_VT)
  int lwork = 0;
  int info_gpu = 0;          // info copied from device to host
  const float h_one = 1;
  const float h_minus_one = -1;
// create cusolver and cublas handle
  cusolver_status = cusolverDnCreate(&cusolverH);
  cublas_status = cublasCreate(&cublasH);
// prepare memory on the device
  cudaStat = cudaMalloc((void**)&d_A,sizeof(float)*lda*n);
  cudaStat = cudaMalloc((void**)&d_S,sizeof(float)*n);
  cudaStat = cudaMalloc((void**)&d_U,sizeof(float)*lda*m);
  cudaStat = cudaMalloc((void**)&d_VT,sizeof(float)*lda*n);
  cudaStat = cudaMalloc((void**)&devInfo,sizeof(int));
  cudaStat = cudaMalloc((void**)&d_W, sizeof(float)*lda*n);
  cudaStat = cudaMemcpy(d_A, A, sizeof(float)*lda*n,
             cudaMemcpyHostToDevice);           // copy A->d_A
// compute buffer size and prepare workspace
  cusolver_status = cusolverDnSgesvd_bufferSize(cusolverH,m,n,
                                                     &lwork );
  cudaStat = cudaMalloc((void**)&d_work,sizeof(float)*lwork);
// compute the singular value decomposition of d_A
// and optionally the left and right singular vectors:
// d_A = d_U*d_S*d_VT; the diagonal elements of d_S
// are the singular values of d_A in descending order
// the first min(m,n) columns of d_U contain the left sing.vec.
// the first min(m,n) cols of d_VT contain the right sing.vec.
  signed char jobu = 'A';     // all m columns of d_U returned
  signed char jobvt = 'A';   // all n columns of d_VT returned
  clock_gettime(CLOCK_REALTIME,&start);          // start timer

  cusolver_status = cusolverDnSgesvd (cusolverH, jobu, jobvt,
  m, n, d_A, lda, d_S, d_U, lda, d_VT, lda, d_work, lwork,
                        d_rwork, devInfo);

  cudaStat = cudaDeviceSynchronize();
  clock_gettime(CLOCK_REALTIME,&stop);            // stop timer
  accum=(stop.tv_sec-start.tv_sec)+             // elapsed time
        (stop.tv_nsec-start.tv_nsec)/(double)BILLION;
  printf("SVD time: %lf sec.\n",accum);  // print elapsed time
  cudaStat = cudaMemcpy(U,d_U,sizeof(float)*lda*m,
             cudaMemcpyDeviceToHost);            // copy d_U->U
  cudaStat = cudaMemcpy(VT,d_VT,sizeof(float)*lda*n,
             cudaMemcpyDeviceToHost);          // copy d_VT->VT
  cudaStat = cudaMemcpy(S,d_S,sizeof(float)*n,
             cudaMemcpyDeviceToHost);            // copy d_S->S
  cudaStat = cudaMemcpy(&info_gpu,devInfo,sizeof(int),
               cudaMemcpyDeviceToHost);   // devInfo->info_gpu
  printf("after gesvd: info_gpu = %d\n", info_gpu);
// multiply d_VT by the diagonal matrix corresponding to d_S
  cublas_status = cublasSdgmm(cublasH,CUBLAS_SIDE_LEFT,n,n,
  d_VT, lda, d_S, 1, d_W, lda);                 // d_W=d_S*d_VT
  cudaStat = cudaMemcpy(d_A,A,sizeof(float)*lda*n,
               cudaMemcpyHostToDevice);          // copy A->d_A
// compute the difference d_A-d_U*d_S*d_VT
  cublas_status=cublasSgemm_v2(cublasH,CUBLAS_OP_N,CUBLAS_OP_N,
  m, n, n, &h_minus_one,d_U, lda, d_W, lda, &h_one, d_A, lda);
  float dR_fro = 0.0;                  // variable for the norm
// compute the norm of the difference d_A-d_U*d_S*d_VT
  cublas_status = cublasSnrm2_v2(cublasH,lda*n,d_A,1,&dR_fro);
  printf("|A - U*S*VT| = %E \n", dR_fro);    // print the norm
// free memory
  cudaFree(d_A);
  cudaFree(d_S);
  cudaFree(d_U);
  cudaFree(d_VT);
  cudaFree(devInfo);
  cudaFree(d_work);
  cudaFree(d_rwork);
  cudaFree(d_W);
  cublasDestroy(cublasH);
  cusolverDnDestroy(cusolverH);
  cudaDeviceReset();
  return 0;
}
```

```
//SVD time: 18.911288 sec.
//after gesvd: info_gpu = 0
//|A - U*S*VT| = 5.613920E-03
```

3.6.2 cusolverDnSgesvd - unified memory version

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <cuda_runtime.h>
#include <cublas_v2.h>
#include <cusolverDn.h>
#define BILLION 1000000000L;
int main(int argc, char*argv[])
{
  struct timespec start,stop;          // variables for timing
  double accum;                        // elapsed time variable
  cusolverDnHandle_t cusolverH;              // cusolver handle
  cublasHandle_t cublasH;                      // cublas handle
  const int m = 2048;                   // number of rows of A
  const int n = 2048;                // number of columns of A
  const int lda = m;                 // leading dimension of A
// declare the factorized matrix A, orthogonal matrices U, VT,
// sing.val. vector S  and auxiliary matr. W = S*VT
  float *A, *A1, *U, *VT, *S, *W;
  cudaMallocManaged(&A,lda*n*sizeof(float)); //unif. mem.for A
  cudaMallocManaged(&A1,lda*n*sizeof(float));//unif.mem.for A1
  cudaMallocManaged(&U,lda*m*sizeof(float)); //unif. mem.for U
  cudaMallocManaged(&VT,lda*n*sizeof(float));//unif.mem.for VT
  cudaMallocManaged(&S,n*sizeof(float));   //unified mem.for S
  cudaMallocManaged(&W,lda*n*sizeof(float)); //unif. mem.for W
  for(int i=0;i<lda*n;i++) A[i]=rand()/(float)RAND_MAX;
  int *Info;                           // info for gesvd  fun.
  float *work, *rwork ;                            // workspace
  int lwork = 0;                              // workspace size
  const float h_one = 1;      // constants used in SVD checking
  const float h_minus_one = -1;
// create cusolver and cublas handle
  cusolverDnCreate(&cusolverH);
  cublasCreate(&cublasH);
  cudaMallocManaged(&Info,sizeof(int)); //unified mem.for Info
  cudaMemcpy(A1, A, sizeof(float)*lda*n,
             cudaMemcpyHostToDevice);               // copy A->A1
// compute buffer size and prepare workspace
  cusolverDnSgesvd_bufferSize(cusolverH,m,n,&lwork );
  cudaMallocManaged(&work,lwork*sizeof(float)); //mem.for work
// compute the singular value decomposition of A
// and optionally the left and right singular vectors:
// A = U*S*VT; the diagonal elements of S
// 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) cols of VT contain the right sing.vec.
  signed char jobu = 'A';       // all m columns of U returned
  signed char jobvt = 'A';   // all n columns of d_VT returned
  clock_gettime(CLOCK_REALTIME,&start);         // start timer

  cusolverDnSgesvd (cusolverH, jobu, jobvt,
  m, n, A1, lda, S, U, lda, VT, lda, work,lwork, rwork, Info);

  cudaDeviceSynchronize();
  clock_gettime(CLOCK_REALTIME,&stop);            // stop timer
  accum=(stop.tv_sec-start.tv_sec)+             // elapsed time
        (stop.tv_nsec-start.tv_nsec)/(double)BILLION;
  printf("SVD time: %lf sec.\n",accum); // print  elapsed time
  printf("after gesvd: info = %d\n", *Info);
// multiply VT by the diagonal matrix corresponding to S
  cublasSdgmm(cublasH,CUBLAS_SIDE_LEFT,n,n,
                     VT, lda, S, 1, W, lda);       // W=S*VT
  cudaMemcpy(A1,A,sizeof(float)*lda*n,
             cudaMemcpyHostToDevice);            // copy A->A1
// compute the difference A1-U*S*VT
  cublasSgemm_v2(cublasH,CUBLAS_OP_N,CUBLAS_OP_N,
  m, n, n, &h_minus_one, U, lda, W, lda, &h_one, A1, lda);
  float nrm = 0.0;                      // variable for the norm
// compute the norm of the difference A1-U*S*VT
  cublasSnrm2_v2(cublasH,lda*n,A1,1,&nrm);
  printf("|A - U*S*VT| = %E \n", nrm);       // print the norm
// free memory
  cudaFree(A);
  cudaFree(A1);
  cudaFree(U);
  cudaFree(VT);
  cudaFree(S);
  cudaFree(W);
  cudaFree(Info);
  cudaFree(work);
  cudaFree(rwork);
  cublasDestroy(cublasH);
  cusolverDnDestroy(cusolverH);
  cudaDeviceReset();
  return 0;
}
//SVD time: 19.200704 sec.
//after gesvd: info = 0
//|A - U*S*VT| = 5.613920E-03
```

3.6.3 `cusolverDnDgesvd` - SVD decomposition in double precision

This function computes in double precision the singular value decomposition of an $m \times n$ matrix:

$$A = u\,\sigma\,v^T,$$

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 respectively.

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <cuda_runtime.h>
#include <cublas_v2.h>
#include <cusolverDn.h>
#define BILLION 1000000000L;
int main(int argc, char*argv[])
{
  struct timespec start,stop;          // variables for timing
  double accum;                       // elapsed time variable
  cusolverDnHandle_t cusolverH;             // cusolver handle
  cublasHandle_t cublasH;                     // cublas handle
  cublasStatus_t cublas_status = CUBLAS_STATUS_SUCCESS;
  cusolverStatus_t cusolver_status = CUSOLVER_STATUS_SUCCESS;
  cudaError_t cudaStat = cudaSuccess;
  const int m = 2048;                     // number of rows of A
  const int n = 2048;                  // number of columns of A
  const int lda = m;                   // leading dimension of A
// declare the factorized matrix A, orthogonal matrices  U, VT
  double *A, *U, *VT, *S;// and sing.val. matrix S on the host
  A=(double*)malloc(lda*n*sizeof(double));
  U=(double*)malloc(lda*m*sizeof(double));
  VT=(double*)malloc(lda*n*sizeof(double));
  S= (double*)malloc(n*sizeof(double));
  for(int i=0;i<lda*n;i++) A[i]=rand()/(double)RAND_MAX;
// the factorized matrix d_A, orthogonal matrices  d_U, d_VT
  double *d_A, *d_U, *d_VT, *d_S;  // and sing.val. matrix d_S
  int *devInfo;                               // on the device
  double *d_work, *d_rwork ;         // workspace on the device
  double *d_W;    // auxiliary device array   (d_W = d_S*d_VT)
  int lwork = 0;
  int info_gpu = 0;          // info copied from device to host
  const double h_one = 1;
  const double h_minus_one = -1;
// create cusolver and cublas handle
  cusolver_status = cusolverDnCreate(&cusolverH);
  cublas_status = cublasCreate(&cublasH);
// prepare memory on the device
  cudaStat = cudaMalloc((void**)&d_A,sizeof(double)*lda*n);
  cudaStat = cudaMalloc((void**)&d_S,sizeof(double)*n);
  cudaStat = cudaMalloc((void**)&d_U,sizeof(double)*lda*m);
  cudaStat = cudaMalloc((void**)&d_VT,sizeof(double)*lda*n);
  cudaStat = cudaMalloc((void**)&devInfo,sizeof(int));
  cudaStat = cudaMalloc((void**)&d_W, sizeof(double)*lda*n);
  cudaStat = cudaMemcpy(d_A, A, sizeof(double)*lda*n,
             cudaMemcpyHostToDevice);            // copy A->d_A
// compute buffer size and prepare workspace
  cusolver_status = cusolverDnDgesvd_bufferSize(cusolverH,m,n,
                                                     &lwork );
  cudaStat = cudaMalloc((void**)&d_work,sizeof(double)*lwork);
// compute the singular value decomposition of d_A
// and optionally the left and right singular vectors:
// d_A = d_U*d_S*d_VT; the diagonal elements of d_S
// are the singular values of d_A in descending order
// the first min(m,n) columns of d_U contain the left sing.vec.
// the first min(m,n) cols of d_VT contain the right sing.vec.
  signed char jobu = 'A';      // all m columns of d_U returned
  signed char jobvt = 'A';    // all n columns of d_VT returned
  clock_gettime(CLOCK_REALTIME,&start);         // start timer

  cusolver_status = cusolverDnDgesvd (cusolverH, jobu, jobvt,
  m, n, d_A, lda, d_S, d_U, lda, d_VT, lda, d_work, lwork,
                        d_rwork, devInfo);

  cudaStat = cudaDeviceSynchronize();
  clock_gettime(CLOCK_REALTIME,&stop);            // stop timer
  accum=(stop.tv_sec-start.tv_sec)+             // elapsed time
        (stop.tv_nsec-start.tv_nsec)/(double)BILLION;
  printf("SVD time: %lf sec.\n",accum);  // print elapsed time
  cudaStat = cudaMemcpy(U,d_U,sizeof(double)*lda*m,
             cudaMemcpyDeviceToHost);            // copy d_U->U
  cudaStat = cudaMemcpy(VT,d_VT,sizeof(double)*lda*n,
             cudaMemcpyDeviceToHost);          // copy d_VT->VT
  cudaStat = cudaMemcpy(S,d_S,sizeof(double)*n,
             cudaMemcpyDeviceToHost);            // copy d_S->S
  cudaStat = cudaMemcpy(&info_gpu,devInfo,sizeof(int),
               cudaMemcpyDeviceToHost);   // devInfo->info_gpu
  printf("after gesvd: info_gpu = %d\n", info_gpu);
// multiply d_VT by the diagonal matrix corresponding to d_S
  cublas_status = cublasDdgmm(cublasH,CUBLAS_SIDE_LEFT,n,n,
  d_VT, lda, d_S, 1, d_W, lda);                 // d_W=d_S*d_VT
  cudaStat = cudaMemcpy(d_A,A,sizeof(double)*lda*n,
               cudaMemcpyHostToDevice);          // copy A->d_A
// compute the difference d_A-d_U*d_S*d_VT
  cublas_status=cublasDgemm_v2(cublasH,CUBLAS_OP_N,CUBLAS_OP_N,
  m, n, n, &h_minus_one,d_U, lda, d_W, lda, &h_one, d_A, lda);
  double dR_fro = 0.0;                    // variable for the norm
// compute the norm of the difference d_A-d_U*d_S*d_VT
  cublas_status = cublasDnrm2_v2(cublasH,lda*n,d_A,1,&dR_fro);
  printf("|A - U*S*VT| = %E \n", dR_fro);     // print the norm
// free memory
```

```
  cudaFree(d_A);
  cudaFree(d_S);
  cudaFree(d_U);
  cudaFree(d_VT);
  cudaFree(devInfo);
  cudaFree(d_work);
  cudaFree(d_rwork);
  cudaFree(d_W);
  cublasDestroy(cublasH);
  cusolverDnDestroy(cusolverH);
  cudaDeviceReset();
  return 0;
}
//SVD time: 22.178122 sec.
//after gesvd: info_gpu = 0
//|A - U*S*VT| = 8.710823E-12
```

3.6.4 cusolverDnDgesvd - unified memory version

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <cuda_runtime.h>
#include <cublas_v2.h>
#include <cusolverDn.h>
#define BILLION 1000000000L;
int main(int argc, char*argv[])
{
  struct timespec start,stop;           // variables for timing
  double accum;                        // elapsed time variable
  cusolverDnHandle_t cusolverH;              // cusolver handle
  cublasHandle_t cublasH;                      // cublas handle
  const int m = 2048;                      // number of rows of A
  const int n = 2048;                   // number of columns of A
  const int lda = m;                    // leading dimension of A
// declare the factorized matrix A, orthogonal matrices U, VT,
// sing.val. vector S  and auxiliary matr. W = S*VT
  double *A, *A1, *U, *VT, *S, *W;
  cudaMallocManaged(&A,lda*n*sizeof(double));//unif. mem.for A
  cudaMallocManaged(&A1,lda*n*sizeof(double));//uni.mem.for A1
  cudaMallocManaged(&U,lda*m*sizeof(double));//unif. mem.for U
  cudaMallocManaged(&VT,lda*n*sizeof(double));//uni.mem.for VT
  cudaMallocManaged(&S,n*sizeof(double));  //unified mem.for S
  cudaMallocManaged(&W,lda*n*sizeof(double));//unif. mem.for W
  for(int i=0;i<lda*n;i++) A[i]=rand()/(double)RAND_MAX;
  int *Info;                            // info for gesvd  fun.
  double *work, *rwork ;                             // workspace
  int lwork = 0;                                // workspace size
  const double h_one = 1;    // constants used in SVD checking
  const double h_minus_one = -1;
// create cusolver and cublas handle
  cusolverDnCreate(&cusolverH);
  cublasCreate(&cublasH);
  cudaMallocManaged(&Info,sizeof(int)); //unified mem.for Info
  cudaMemcpy(A1, A, sizeof(double)*lda*n,
             cudaMemcpyHostToDevice);             // copy A->A1
// compute buffer size and prepare workspace
  cusolverDnDgesvd_bufferSize(cusolverH,m,n,&lwork );
  cudaMallocManaged(&work,lwork*sizeof(double));//mem.for work
// compute the singular value decomposition of A
// and optionally the left and right singular vectors:
// A = U*S*VT; the diagonal elements of S
// 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) cols of VT contain the right sing.vec.
  signed char jobu = 'A';       // all m columns of U returned
  signed char jobvt = 'A';   // all n columns of d_VT returned
  clock_gettime(CLOCK_REALTIME,&start);          // start timer

  cusolverDnDgesvd (cusolverH, jobu, jobvt, m, n, A1, lda, S, U,
                    lda, VT, lda, work, lwork, rwork, Info);

  cudaDeviceSynchronize();
  clock_gettime(CLOCK_REALTIME,&stop);            // stop timer
  accum=(stop.tv_sec-start.tv_sec)+             // elapsed time
       (stop.tv_nsec-start.tv_nsec)/(double)BILLION;
  printf("SVD time: %lf sec.\n",accum);  // print elapsed time
  printf("after gesvd: info = %d\n", *Info);
// multiply VT by the diagonal matrix corresponding to S
  cublasDdgmm(cublasH,CUBLAS_SIDE_LEFT,n,n,
                     VT, lda, S, 1, W, lda);       // W=S*VT
  cudaMemcpy(A1,A,sizeof(double)*lda*n,
               cudaMemcpyHostToDevice);           // copy A->A1
// compute the difference A1-U*S*VT
  cublasDgemm_v2(cublasH,CUBLAS_OP_N,CUBLAS_OP_N,
  m, n, n, &h_minus_one, U, lda, W, lda, &h_one, A1, lda);
  double nrm = 0.0;                     // variable for the norm
// compute the norm of the difference A1-U*S*VT
  cublasDnrm2_v2(cublasH,lda*n,A1,1,&nrm);
  printf("|A - U*S*VT| = %E \n", nrm);        // print the norm
// free memory
  cudaFree(A);
  cudaFree(A1);
  cudaFree(U);
  cudaFree(VT);
  cudaFree(S);
  cudaFree(W);
  cudaFree(Info);
  cudaFree(work);
  cudaFree(rwork);
  cublasDestroy(cublasH);
  cusolverDnDestroy(cusolverH);
  cudaDeviceReset();
  return 0;
}
//SVD time: 22.325408 sec.
//after gesvd: info = 0
//|A - U*S*VT| = 8.710823E-12
```

3.7 Eigenvalues and eigenvectors for symmetric matrices

3.7.1 cusolverDnSsyevd - eigenvalues and eigenvectors for symmetric matrices in single precision

This function computes in single precision all eigenvalues and, optionally, eigenvectors of a real symmetric matrix A. The second parameter can take the values CUSOLVER_EIG_MODE_VECTOR or CUSOLVER_EIG_MODE_NOVECTOR and answers the question whether the eigenvectors are desired. The symmetric matrix A can be stored in lower (CUBLAS_FILL_MODE_LOWER) or upper (CUBLAS_FILL_MODE_UPPER) mode. If the eigenvectors are desired, then on exit A contains orthonormal eigenvectors. The eigenvalues are stored in an array W.

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include <cusolverDn.h>
#define BILLION 1000000000L;
int main(int argc, char*argv[])
{
  struct timespec start,stop;          // variables for timing
  double accum;                        // elapsed time variable
  cusolverDnHandle_t cusolverH;
  cusolverStatus_t cusolver_status = CUSOLVER_STATUS_SUCCESS;
  cudaError_t cudaStat = cudaSuccess;
  const int m = 2048;        // number of rows and columns of A
  const int lda = m;                  // leading dimension of A
  float *A;                                       // mxm matrix
  float *V;                       // mxm matrix of eigenvectors
  float *W;                         //  m-vector of eigenvalues
// prepare memory on the host
  A = (float*)malloc(lda*m*sizeof(float));
  V = (float*)malloc(lda*m*sizeof(float));
  W = (float*)malloc(m*sizeof(float));
// define random A
  for(int i=0;i<lda*m;i++) A[i] = rand()/(float)RAND_MAX;
// declare arrays on the device
  float *d_A;                       // mxm matrix A on the device
  float *d_W;           // m-vector of eigenvalues on the device
  int *devInfo;                            // info on the device
  float *d_work;                      // workspace on the device
  int  lwork = 0;                                // workspace size
  int info_gpu = 0;          // info copied from device to host
// create cusolver handle
  cusolver_status = cusolverDnCreate(&cusolverH);
// prepare memory on the device
  cudaStat = cudaMalloc ((void**)&d_A,sizeof(float)*lda*m);
  cudaStat = cudaMalloc ((void**)&d_W,sizeof(float)*m);
  cudaStat = cudaMalloc ((void**)&devInfo,sizeof(int));
  cudaStat = cudaMemcpy(d_A,A,sizeof(float)*lda*m,
             cudaMemcpyHostToDevice);            // copy A->d_A
// compute eigenvalues and eigenvectors
  cusolverEigMode_t jobz = CUSOLVER_EIG_MODE_VECTOR;
// use lower left triangle of the matrix
  cublasFillMode_t uplo = CUBLAS_FILL_MODE_LOWER;
// compute buffer size and prepare workspace
  cusolver_status = cusolverDnSsyevd_bufferSize(cusolverH,
                      jobz, uplo, m, d_A, lda, d_W, &lwork);
  cudaStat = cudaMalloc((void**)&d_work,sizeof(float)*lwork);
  clock_gettime(CLOCK_REALTIME,&start);         // start timer
// compute the eigenvalues and eigenvectors for a symmetric,
// real mxm matrix (only the lower left triangle af A is used)

  cusolver_status = cusolverDnSsyevd(cusolverH, jobz, uplo, m,
                      d_A, lda, d_W, d_work, lwork, devInfo);

  cudaStat = cudaDeviceSynchronize();
  clock_gettime(CLOCK_REALTIME,&stop);            // stop timer
  accum=(stop.tv_sec-start.tv_sec)+             // elapsed time
        (stop.tv_nsec-start.tv_nsec)/(double)BILLION;
  printf("Ssyevd time: %lf sec.\n",accum);//print elapsed time
  cudaStat = cudaMemcpy(W, d_W, sizeof(float)*m,
             cudaMemcpyDeviceToHost);            // copy d_W->W
  cudaStat = cudaMemcpy(V, d_A, sizeof(float)*lda*m,
             cudaMemcpyDeviceToHost);            // copy d_A->V
  cudaStat = cudaMemcpy(&info_gpu, devInfo, sizeof(int),
             cudaMemcpyDeviceToHost);// copy devInfo->info_gpu
  printf("after syevd: info_gpu = %d\n", info_gpu);
  printf("eigenvalues:\n");          // print first eigenvalues
  for(int i = 0 ; i < 3 ; i++){
     printf("W[%d] = %E\n", i+1, W[i]);
  }
// free memory
  cudaFree(d_A);
  cudaFree(d_W);
  cudaFree(devInfo);
  cudaFree(d_work);
  cusolverDnDestroy(cusolverH);
  cudaDeviceReset();
  return 0;
}
```

```
//Ssyevd time: 2.110875 sec.
//after syevd: info_gpu = 0
//eigenvalues:
//W[1] = -2.582270E+01
//W[2] = -2.566824E+01
//W[3] = -2.563596E+01
```

3.7.2 cusolverDnSsyevd - unified memory version

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include <cusolverDn.h>
#define BILLION 1000000000L;
int main(int argc, char*argv[])
{
  struct timespec start,stop;          // variables for timing
  double accum;                        // elapsed time variable
  cusolverDnHandle_t cusolverH;
  const int m = 2048;        // number of rows and columns of A
  const int lda = m;                   // leading dimension of A
  float *A;                                          // mxm matrix
  float *W;                          //  m-vector of eigenvalues
// prepare memory
  cudaMallocManaged(&A,lda*m*sizeof(float)); //unif. mem.for A
  cudaMallocManaged(&W,m*sizeof(float));     //unif. mem.for W
// define random A
  for(int i=0;i<lda*m;i++) A[i] = rand()/(float)RAND_MAX;
  int *Info;                                             // info
  float *work;                                      // workspace
  int  lwork = 0;                              // workspace size
// create cusolver handle
  cusolverDnCreate(&cusolverH);
  cudaMallocManaged(&Info,sizeof(int));   // unif.mem.for Info
// compute eigenvalues and eigenvectors
  cusolverEigMode_t jobz = CUSOLVER_EIG_MODE_VECTOR;
// use lower left triangle of the matrix
  cublasFillMode_t uplo = CUBLAS_FILL_MODE_LOWER;
// compute buffer size and prepare workspace
  cusolverDnSsyevd_bufferSize(cusolverH,
                      jobz, uplo, m, A, lda, W, &lwork);
  cudaMallocManaged(&work,lwork*sizeof(float)); //mem.for work
  clock_gettime(CLOCK_REALTIME,&start);          // start timer
// compute the eigenvalues and eigenvectors for a symmetric,
// real mxm matrix (only the lower left triangle af A is used)

  cusolverDnSsyevd(cusolverH, jobz, uplo, m, A, lda, W, work,
                   lwork, Info);
  cudaDeviceSynchronize();
  clock_gettime(CLOCK_REALTIME,&stop);            // stop timer
  accum=(stop.tv_sec-start.tv_sec)+              // elapsed time
        (stop.tv_nsec-start.tv_nsec)/(double)BILLION;
  printf("syevd time: %lf sec.\n",accum);// print elapsed time
  printf("after syevd: info = %d\n", *Info);
  printf("eigenvalues:\n");          // print first eigenvalues
  for(int i = 0 ; i < 3 ; i++){
     printf("W[%d] = %E\n", i+1, W[i]);
  }
// free memory
  cudaFree(A);
  cudaFree(W);
  cudaFree(Info);
  cudaFree(work);
  cusolverDnDestroy(cusolverH);
  cudaDeviceReset();
  return 0;
}
//syevd time: 2.246703 sec.
//after syevd: info = 0
//eigenvalues:
//W[1] = -2.582270E+01
//W[2] = -2.566824E+01
//W[3] = -2.563596E+01
```

3.7.3 cusolverDnDsyevd - eigenvalues and eigenvectors for symmetric matrices in double precision

This function computes in double precision all eigenvalues and, optionally, eigenvectors of a real symmetric matrix A. The second parameter can take the values CUSOLVER_EIG_MODE_VECTOR or CUSOLVER_EIG_MODE_NOVECTOR and answers the question whether the eigenvectors are desired. The symmetric matrix A can be stored in lower (CUBLAS_FILL_MODE_LOWER) or upper (CUBLAS_FILL_MODE_UPPER) mode. If the eigenvectors are desired, then on exit A contains orthonormal eigenvectors. The eigenvalues are stored in an array W.

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include <cusolverDn.h>
#define BILLION 1000000000L;
int main(int argc, char*argv[])
{
  struct timespec start,stop;          // variables for timing
  double accum;                        // elapsed time variable
  cusolverDnHandle_t cusolverH;
  cusolverStatus_t cusolver_status = CUSOLVER_STATUS_SUCCESS;
  cudaError_t cudaStat = cudaSuccess;
  const int m = 2048;       // number of rows and columns of A
  const int lda = m;                  // leading dimension of A
  double *A;                                      // mxm matrix
  double *V;                    // mxm matrix of eigenvectors
  double *W;                     //  m-vector of eigenvalues
// prepare memory on the host
  A = (double*)malloc(lda*m*sizeof(double));
  V = (double*)malloc(lda*m*sizeof(double));
  W = (double*)malloc(m*sizeof(double));
// define random A
  for(int i=0;i<lda*m;i++) A[i] = rand()/(double)RAND_MAX;
// declare arrays on the device
  double *d_A;                  // mxm matrix A on the device
  double *d_W;         // m-vector of eigenvalues on the device
  int *devInfo;                           // info on the device
  double *d_work;                    // workspace on the device
  int  lwork = 0;                                // workspace size
  int info_gpu = 0;         // info copied from device to host
// create cusolver handle
  cusolver_status = cusolverDnCreate(&cusolverH);
// prepare memory on the device
  cudaStat = cudaMalloc ((void**)&d_A,sizeof(double)*lda*m);
  cudaStat = cudaMalloc ((void**)&d_W,sizeof(double)*m);
  cudaStat = cudaMalloc ((void**)&devInfo,sizeof(int));
  cudaStat = cudaMemcpy(d_A,A,sizeof(double)*lda*m,
             cudaMemcpyHostToDevice);           // copy A->d_A
// compute eigenvalues and eigenvectors
  cusolverEigMode_t jobz = CUSOLVER_EIG_MODE_VECTOR;
// use lower left triangle of the matrix
  cublasFillMode_t uplo = CUBLAS_FILL_MODE_LOWER;
// compute buffer size and prepare workspace
  cusolver_status = cusolverDnDsyevd_bufferSize(cusolverH,
                      jobz, uplo, m, d_A, lda, d_W, &lwork);
  cudaStat = cudaMalloc((void**)&d_work,sizeof(double)*lwork);
  clock_gettime(CLOCK_REALTIME,&start);         // start timer
// compute the eigenvalues and eigenvectors for a symmetric,
// real mxm matrix (only the lower left triangle af A is used)

  cusolver_status = cusolverDnDsyevd(cusolverH, jobz, uplo, m,
                       d_A, lda, d_W, d_work, lwork, devInfo);

  cudaStat = cudaDeviceSynchronize();
  clock_gettime(CLOCK_REALTIME,&stop);           // stop timer
  accum=(stop.tv_sec-start.tv_sec)+            // elapsed time
        (stop.tv_nsec-start.tv_nsec)/(double)BILLION;
  printf("Dsyevd time: %lf sec.\n",accum);//print elapsed time
  cudaStat = cudaMemcpy(W, d_W, sizeof(double)*m,
             cudaMemcpyDeviceToHost);           // copy d_W->W
  cudaStat = cudaMemcpy(V, d_A, sizeof(double)*lda*m,
             cudaMemcpyDeviceToHost);           // copy d_A->V
  cudaStat = cudaMemcpy(&info_gpu, devInfo, sizeof(int),
              cudaMemcpyDeviceToHost);// copy devInfo->info_gpu
  printf("after syevd: info_gpu = %d\n", info_gpu);
  printf("eigenvalues:\n");          // print first eigenvalues
  for(int i = 0 ; i < 3 ; i++){
     printf("W[%d] = %E\n", i+1, W[i]);
  }
// free memory
  cudaFree(d_A);
  cudaFree(d_W);
  cudaFree(devInfo);
  cudaFree(d_work);
  cusolverDnDestroy(cusolverH);
  cudaDeviceReset();
  return 0;
}
//Dsyevd time: 3.279903 sec.
//after syevd: info_gpu = 0
//eigenvalues:
//W[1] = -2.582273E+01
//W[2] = -2.566824E+01
//W[3] = -2.563596E+01
```

3.7.4 cusolverDnDsyevd - unified memory version

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include <cusolverDn.h>
#define BILLION 1000000000L;
int main(int argc, char*argv[])
{
  struct timespec start,stop;          // variables for timing
  double accum;                        // elapsed time variable
  cusolverDnHandle_t cusolverH;
  const int m = 2048;        // number of rows and columns of A
  const int lda = m;                  // leading dimension of A
  double *A;                                      // mxm matrix
  double *W;                        //  m-vector of eigenvalues
// prepare memory
  cudaMallocManaged(&A,lda*m*sizeof(double)); //unif.mem.for A
  cudaMallocManaged(&W,m*sizeof(double));     //unif.mem.for W
// define random A
  for(int i=0;i<lda*m;i++) A[i] = rand()/(double)RAND_MAX;
  int *Info;                                           // info
  double *work;                                   // workspace
  int  lwork = 0;                            // workspace size
// create cusolver handle
  cusolverDnCreate(&cusolverH);
  cudaMallocManaged(&Info,sizeof(int));//unified mem. for Info
// compute eigenvalues and eigenvectors
  cusolverEigMode_t jobz = CUSOLVER_EIG_MODE_VECTOR;
// use lower left triangle of the matrix
  cublasFillMode_t uplo = CUBLAS_FILL_MODE_LOWER;
// compute buffer size and prepare workspace
  cusolverDnDsyevd_bufferSize(cusolverH,
                          jobz, uplo, m, A, lda, W, &lwork);
  cudaMallocManaged(&work,lwork*sizeof(double));//mem.for work
  clock_gettime(CLOCK_REALTIME,&start);          // start timer
// compute the eigenvalues and eigenvectors for a symmetric,
// real mxm matrix (only the lower left triangle af A is used)

  cusolverDnDsyevd(cusolverH, jobz, uplo, m, A, lda, W, work,
                          lwork, Info);

  cudaDeviceSynchronize();
  clock_gettime(CLOCK_REALTIME,&stop);            // stop timer
  accum=(stop.tv_sec-start.tv_sec)+             // elapsed time
        (stop.tv_nsec-start.tv_nsec)/(double)BILLION;
  printf("Dsyevd time: %lf sec.\n",accum);//print elapsed time
  printf("after syevd: info = %d\n", *Info);
  printf("eigenvalues:\n");           // print first eigenvalues
  for(int i = 0 ; i < 3 ; i++){
     printf("W[%d] = %E\n", i+1, W[i]);
  }
// free memory
  cudaFree(A);
  cudaFree(W);
  cudaFree(Info);
  cudaFree(work);
  cusolverDnDestroy(cusolverH);
  cudaDeviceReset();
  return 0;
}

//Dsyevd time: 3.395064 sec.
//after syevd: info = 0
//eigenvalues:
//W[1] = -2.582273E+01
//W[2] = -2.566824E+01
//W[3] = -2.563596E+01
```

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 linear algebra routines for dense and sparse matrices. It is a successor of Lapack and ScaLapack, specially developed for heterogeneous CPU-GPU architectures. Magma is an open-source project developed by Innovative Computing Laboratory (ICL), University of Tennessee, Knoxville, USA.
The main ingredients of (dense) Magma are:

- LU, QR and Cholesky factorization.
- 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.

Magma Sparse contains (among other things):

- Sparse linear solvers.
- Sparse eigenvalues.
- Sparse preconditioners.

There is also Magma Batched, which allows for parallel computations on a set of small matrices.
A more detailed information on procedures contained in Magma can be

found in *MAGMA Users's Guide*: `http://icl.cs.utk.edu/projectsfiles/magma/doxygen/`.
Let us notice that the source files in Magma `src` directory contain precise syntax descriptions of Magma functions, so we do not repeat that information in our text (the syntax is also easily available on the Internet).
Instead, we present a series of examples how to use the (dense part of) 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.

To be precise, there exist also some mixed precision routines of the type `sc, dz, ds, zc`, but we have decided to omit the corresponding examples.

- 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 single and double precision.
- 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 over two hundred 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 descriptions 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 has to 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` and `libmagma_sparse.a` in Magma `lib` subdirectory and testing drivers in `testing` directory.

The method of compilation of examples depends on the libraries specified in `make.inc`. In the present version of our text we used `Openblas` and Magma directory was a subdirectory of `$HOME` directory. We compiled examples in two steps:

```
g++ -O3 -fopenmp -std=c++11 -DHAVE_CUBLAS -I/usr/local/cuda/
include -I../include -c -o 001isamax_v2u.o 001isamax_v2u.cpp

g++ -fopenmp -o 001isamax_v2u 001isamax_v2u.o -L../lib -lm
-lmagma -L/usr/local/cuda/lib64 -L/usr/lib -lopenblas -lcublas
-lcudart
```

Let us remark, that only two examples of the present chapter contain the `cudaDeviceSynchronize()` function. The function `magma_sync_wtime` used in the remaining examples contains the sychronization command and `cudaDeviceSynchronize()` is not necessary. Note however that, for example in subsection 4.2.4 (vectors swapping with unified memory), omitting `cudaDeviceSynchronize()` leads to wrong results (vectors are not swapped).

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 Ubuntu 16.04, CUDA 8.0, magma-2.2.0 compiled with Openblas library

- Intel(R) Core(TM) i7-6700K CPU, 4.00GHz
- Nvidia(R) GeForce GTX 1080

4.2 Magma BLAS

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_v2.h"
int main( int argc, char** argv ){
    magma_init();                                  // initialize Magma
    magma_queue_t queue=NULL;
    magma_int_t dev=0;
    magma_queue_create(dev,&queue);
    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
// allocate  array on the host
    magma_smalloc_cpu( &a , m );             // host memory for a
// allocate  array on the device
    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, queue);  // 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, queue );

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

4.2.2 magma_isamax - unified memory version

```
#include <stdlib.h>
#include <stdio.h>
#include "cuda_runtime.h"
#include "magma_v2.h"
int main( int argc, char** argv ){
    magma_init();                              // initialize Magma
    magma_queue_t queue=NULL;
    magma_int_t dev=0;
    magma_queue_create(dev,&queue);
    magma_int_t m = 1024;                           // length of a
    float *a;                                          // m-vector
// allocate array a in unified memory
    cudaMallocManaged(&a,m*sizeof(float));
                               // a={sin(0),sin(1),...,sin(m-1)}
    for(int j=0;j<m;j++) a[j]=sin((float)j);
// find the smallest index of the element of a with maximum
// absolute value

    int i = magma_isamax( m, a, 1, queue );

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

4.2.3 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_v2.h"
int main( int argc, char** argv ){
    magma_init();                              // initialize Magma
    magma_queue_t queue=NULL;
    magma_int_t dev=0;
    magma_queue_create(dev,&queue);
    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_int_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,queue);  // copy a -> d_a
    magma_ssetvector( m, b, 1, d_b,1,queue);  // copy b -> d_b
// swap the vectors

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

    magma_sgetvector( m, d_a, 1, a, 1,queue); // copy d_a -> a
    magma_sgetvector( m, d_b, 1, b, 1,queue); // 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_queue_destroy(queue);
    magma_finalize();
    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,...
```

4.2.4 magma_sswap - unified memory version

```
#include <stdlib.h>
#include <stdio.h>
#include <cuda_runtime.h>
#include "magma_v2.h"
int main( int argc, char** argv ){
    magma_init();                           // initialize Magma
    magma_queue_t queue=NULL;
    magma_int_t dev=0;
    magma_queue_create(dev,&queue);
    magma_int_t m = 1024;                       // length of a,b
    float *a;                                    // a- m-vector
    float *b;                                    // b- m-vector
    cudaMallocManaged(&a,m*sizeof(float));// unif.memory for a
    cudaMallocManaged(&b,m*sizeof(float));// unif.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");
// swap the vectors

    magma_sswap( m, a, 1, b, 1, queue );

    cudaDeviceSynchronize();
    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");
    magma_free(a);                              // free  memory
    magma_free(b);                              // free  memory
    magma_queue_destroy(queue);
    magma_finalize();
    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,...
```

4.2.5 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 case), A^T (transposition) in MagmaTrans case or A^H (conjugate transposition) in MagmaConjTrans case.

```
#include <stdio.h>
#include <cuda.h>
#include "magma_v2.h"
```

```
#include "magma_lapack.h"
int main( int argc, char** argv ){
    magma_init();                                  // initialize Magma
    magma_queue_t queue=NULL;
    magma_int_t dev=0;
    magma_queue_create(dev,&queue);
    real_Double_t  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;     //random uniform distr. in (0,1)
    magma_int_t ISEED[4] = { 0,1,2,3 };                    // seed
    magma_int_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);        // randomize a
    lapackf77_slarnv(&ione,ISEED,&n,b);         // randomize b
    lapackf77_slarnv(&ione,ISEED,&m,c);         // randomize c
// copy data from host to device
    magma_ssetmatrix( m, n, a,m,d_a,m,queue); // copy a -> d_a
    magma_ssetvector( n, b, 1, d_b, 1,queue); // copy b -> d_b
    magma_ssetvector( m, c, 1, d_c, 1,queue); // 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
    gpu_time = magma_sync_wtime(NULL);

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

    gpu_time = magma_sync_wtime(NULL)-gpu_time;
    printf("magma_sgemv time: %7.5f sec.\n",gpu_time);
// copy data from device to host
    magma_sgetvector( m, d_c, 1, c2, 1,queue);// 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_queue_destroy(queue);
    magma_finalize();                           // finalize Magma
    return 0;
}
//magma_sgemv time: 0.00016 sec.
//after magma_sgemv:
//c2:  507.9388, 498.1866, 503.1055, 508.1643,...
```

4.2.6 magma_sgemv - unified memory version

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
    magma_init();                             // initialize Magma
    magma_queue_t queue=NULL;
    magma_int_t dev=0;
    magma_queue_create(dev,&queue);
    real_Double_t  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
    float *b;                                      // b- n-vector
    float *c;                                      // c- m-vector
    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;     //random uniform distr. in (0,1)
    magma_int_t ISEED[4] = { 0,1,2,3 };                   // seed
    cudaMallocManaged(&a,m*n*sizeof(float)); // unif.mem.for a
    cudaMallocManaged(&b,n*sizeof(float));   // unif.mem.for b
    cudaMallocManaged(&c,m*sizeof(float));   // unif.mem.for c
// generate random matrix a and vectors b,c
    lapackf77_slarnv(&ione,ISEED,&mn,a);           // randomize a
    lapackf77_slarnv(&ione,ISEED,&n,b);            // randomize b
    lapackf77_slarnv(&ione,ISEED,&m,c);            // randomize c
// matrix-vector multiplication:
// c = alpha*a*b + beta*c;
// a- mxn matrix; b -  n-vector; c - m-vector
```

```
    gpu_time = magma_sync_wtime(NULL);

    magma_sgemv(MagmaNoTrans,m,n,alpha,a,m,b,1,beta,c,1,queue);

    gpu_time = magma_sync_wtime(NULL)-gpu_time;
    printf("magma_sgemv time: %7.5f sec.\n",gpu_time);
    printf("after magma_sgemv:\n");
    printf("c: ");
    for(int j=0;j<4;j++) printf("%9.4f,",c[j]);
    printf("...\n");
    magma_free(a);                                  // free  memory
    magma_free(b);                                  // free  memory
    magma_free(c);                                  // free  memory
    magma_queue_destroy(queue);
    magma_finalize();                             // finalize Magma
    return 0;
}
//magma_sgemv time: 0.00504 sec.
//after magma_sgemv:
//c:  507.9388, 498.1866, 503.1055, 508.1643,...
```

4.2.7 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) or upper (MagmaUpper) mode.

```
#include <stdio.h>
#include <cuda.h>
#include "magma_v2.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                               // initialize Magma
  magma_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  real_Double_t  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  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;       //random uniform distr. in (0,1)
  magma_int_t ISEED[4] = { 0,1,2,3 };                  // seed
  magma_int_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);          // randomize a
  lapackf77_slarnv(&ione,ISEED,&m,b);           // randomize b
  lapackf77_slarnv(&ione,ISEED,&m,c);           // randomize c
// copy data from host to device
  magma_ssetmatrix( m, m, a,  m, d_a,m,queue);// copy a -> d_a
  magma_ssetvector( m, b, 1, d_b,1,queue);    // copy b -> d_b
  magma_ssetvector( m, c, 1, d_c,1,queue);    // copy c -> d_c
// symmetric matrix-vector multiplication:
// d_c = alpha*d_a*d_b + beta*d_c;
// d_a- mxm symmetric matrix; b - m-vector; c - m-vector
  gpu_time = magma_sync_wtime(NULL);

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

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("magma_ssymv time: %7.5f sec.\n",gpu_time);
// copy data from device to host
  magma_sgetvector( m, d_c, 1, c2,1,queue);   // 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_queue_destroy(queue);
  magma_finalize();                            // finalize Magma
  return 0;
}
//magma_ssymv time: 0.00033 sec.
//after magma_ssymv:
//c2:  1003.9608, 1029.2787, 1008.7328, 1042.9585,...
```

4.2.8 magma_ssymv - unified memory version

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                              // initialize Magma
  magma_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  real_Double_t  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
// lower triangular part of a contains the lower triangular
// part of some  matrix
  float *b;                                      // b- m-vector
  float *c;                                      // c- m-vector
  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;       //random uniform distr. in (0,1)
  magma_int_t ISEED[4] = { 0,1,2,3 };                   // seed
  cudaMallocManaged(&a,mm*sizeof(float));// unified mem. for a
  cudaMallocManaged(&b,m*sizeof(float)); // unified mem. for b
  cudaMallocManaged(&c,m*sizeof(float)); // unified mem. 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);           // randomize a
  lapackf77_slarnv(&ione,ISEED,&m,b);            // randomize b
  lapackf77_slarnv(&ione,ISEED,&m,c);            // randomize c
// symmetric matrix-vector multiplication:
// c = alpha*a*b + beta*c;
// a- mxm symmetric matrix; b - m-vector; c - m-vector
  gpu_time = magma_sync_wtime(NULL);

  magma_ssymv( MagmaLower,m,alpha,a,m,b,1,beta,c,1,queue);

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("magma_ssymv time: %7.5f sec.\n",gpu_time);
  printf("after magma_ssymv:\n");
  printf("c: ");
  for(int j=0;j<4;j++) printf("%10.4f,",c[j]);
  printf("...\n");
  magma_free(a);                                 // free memory
  magma_free(b);                                 // free memory
  magma_free(c);                                 // free memory
  magma_queue_destroy(queue);
  magma_finalize();                          // finalize Magma
  return 0;
}
```

```
//magma_ssymv time: 0.01379 sec.
//after magma_ssymv:
//c:  1003.9608, 1029.2787, 1008.7328, 1042.9585,...
```

4.2.9 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 in MagmaNoTrans case, A^T (transposition) in MagmaTrans case, or A^H (conjugate transposition) in MagmaConjTrans case and similarly for $op(B)$.

```
#include <stdio.h>
#include <cuda.h>
#include "magma_v2.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                                // initialize Magma
  magma_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  real_Double_t  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;       //random uniform distr. in (0,1)
  magma_int_t ISEED[4] = { 0,1,2,3 };                    // seed
  magma_int_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 matrices and  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);          // randomize a
  lapackf77_slarnv(&ione,ISEED,&kn,b);          // randomize b
  lapackf77_slarnv(&ione,ISEED,&mn,c);          // randomize c
// copy data from host to device
  magma_ssetmatrix( m, k, a,m,d_a,m,queue);   // copy a -> d_a
  magma_ssetmatrix( k, n, b,k,d_b,k,queue);   // copy b -> d_b
  magma_ssetmatrix( m, n, c,m,d_c,m,queue);   // 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
  gpu_time = magma_sync_wtime(NULL);

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

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  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,queue);// 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_queue_destroy(queue);                    // destroy queue
  magma_finalize();                            // finalize Magma
  return 0;
}
//magma_sgemm time: 0.01936 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,...
//  ............................................
```

4.2.10 magma_sgemm - unified memory version

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.h"
```

```
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                               // initialize Magma
  magma_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  real_Double_t  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
  float *b;                                      // b- kxn matrix
  float *c;                                      // c- mxn matrix
  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;       //random uniform distr. in (0,1)
  magma_int_t ISEED[4] = { 0,1,2,3 };                     // seed
  cudaMallocManaged(&a,mk*sizeof(float));// unified mem. for a
  cudaMallocManaged(&b,kn*sizeof(float));// unified mem. for b
  cudaMallocManaged(&c,mn*sizeof(float));// unified mem. for c
// generate random matrices a, b, c;
  lapackf77_slarnv(&ione,ISEED,&mk,a);          // randomize a
  lapackf77_slarnv(&ione,ISEED,&kn,b);          // randomize b
  lapackf77_slarnv(&ione,ISEED,&mn,c);          // randomize c
// matrix-matrix multiplication: c = al*a*b + bet*c
// a -mxk matrix, b -kxn matrix, c -mxn matrix;
// al,bet - scalars
  gpu_time = magma_sync_wtime(NULL);

  magma_sgemm(MagmaNoTrans,MagmaNoTrans,m,n,k,alpha,a,m,b,k,
                     beta,c,m,queue);

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("magma_sgemm time: %7.5f sec.\n",gpu_time);
  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(a);                                // free  memory
  magma_free(b);                                // free  memory
  magma_free(c);                                // free  memory
  magma_queue_destroy(queue);                  // destroy queue
  magma_finalize();                           // finalize Magma
  return 0;
}
//magma_sgemm time: 0.05634 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,...
//..........................................
```

4.2.11 magma_ssymm - symmetric matrix-matrix multiplication

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

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

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) or upper (MagmaUpper) mode.

```
#include <stdio.h>
#include <cuda.h>
#include "magma_v2.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                              // initialize Magma
  magma_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  real_Double_t  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;       //random uniform distr. in (0,1)
  magma_int_t ISEED[4] = { 0,1,2,3 };                    // seed
  magma_int_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 matrices 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);           // randomize a
// lower triangular part of a is the lower triangular part
// of some  matrix, the strictly upper triangular
// part of a is not referenced
  lapackf77_slarnv(&ione,ISEED,&mn,b);           // randomize b
  lapackf77_slarnv(&ione,ISEED,&mn,c);           // randomize c
// copy data from host to device
  magma_ssetmatrix( m, m, a,m, d_a,m,queue ); // copy a -> d_a
  magma_ssetmatrix( m, n, b,m, d_b,m,queue ); // copy b -> d_b
  magma_ssetmatrix( m, n, c,m, d_c,m,queue ); // copy c -> d_c
// 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
  gpu_time = magma_sync_wtime(NULL);

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

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  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,queue); // 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_queue_destroy(queue);                    // destroy queue
  magma_finalize();                             // finalize Magma
  return 0;
}
//magma_ssymm time: 0.20599 sec.
//after magma_ssymm:
//c:
// 2021.3811, 2045.4391, 2048.6990, 2019.2104,...
// 2037.0023, 2050.8364, 2047.5414, 2031.6825,...
// 2053.6797, 2084.0034, 2077.5015, 2068.3196,...
// 2023.3375, 2045.9795, 2051.4314, 2013.8230,...
//...............................................
```

4.2.12 magma_ssymm - unified memory version

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                             // initialize Magma
  magma_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  real_Double_t  gpu_time;
  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
  float *b;                                      // b- mxn matrix
  float *c;                                      // c- mxn matrix
  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;        //random uniform distr. in (0,1)
  magma_int_t ISEED[4] = { 0,1,2,3 };                   // seed
  cudaMallocManaged(&a,mm*sizeof(float)); // unified mem.for a
  cudaMallocManaged(&b,mn*sizeof(float)); // unified mem.for b
  cudaMallocManaged(&c,mn*sizeof(float)); // unified mem.for c
// generate random matrices a, b, c;
  lapackf77_slarnv(&ione,ISEED,&mm,a);          // randomize a
// lower triangular part of a is the lower triangular part
// of some  matrix, the strictly upper triangular
// part of a is not referenced
  lapackf77_slarnv(&ione,ISEED,&mn,b);          // randomize b
  lapackf77_slarnv(&ione,ISEED,&mn,c);          // randomize c
// matrix-matrix multiplication: c = al*a*b + bet*c
// a -mxm symmetric matrix, b, c -mxn matrices;
// al,bet - scalars
  gpu_time = magma_sync_wtime(NULL);

  magma_ssymm( MagmaLeft,MagmaLower,m,n,alpha,a,m,b,m,beta,c,m,
                         queue);
  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("magma_ssymm time: %7.5f sec.\n",gpu_time);
  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(a);                                // free  memory
  magma_free(b);                                // free  memory
  magma_free(c);                                // free  memory
  magma_queue_destroy(queue);                      // destroy queue
  magma_finalize();                               // finalize Magma
  return 0;
}
//magma_ssymm time: 0.27643 sec.
//after magma_ssymm:
//c:
// 2021.3811, 2045.4391, 2048.6990, 2019.2104,...
// 2037.0023, 2050.8364, 2047.5414, 2031.6825,...
// 2053.6797, 2084.0034, 2077.5015, 2068.3196,...
// 2023.3375, 2045.9795, 2051.4314, 2013.8230,...
//...............................................
```

4.2.13 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) or upper (MagmaUpper) mode and α, β are scalars. The value of $op(A)$ can be equal to A in MagmaNoTrans case or A^T (transposition) in MagmaTrans case.

```
#include <stdio.h>
#include <cuda.h>
#include "magma_v2.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                                 // initialize Magma
  magma_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  real_Double_t  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;        //random uniform distr. in (0,1)
  magma_int_t ISEED[4] = { 0,1,2,3 };                       // seed
  magma_int_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 matrices 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);          // randomize a
  lapackf77_slarnv(&ione,ISEED,&mm,c);          // randomize c
// upper triangular part of c is the upper triangular part
// of some  matrix, the strictly lower triangular
// part of c is not referenced
// copy data from host to device
  magma_ssetmatrix( m, k, a, m, d_a,m,queue); // copy a -> d_a
  magma_ssetmatrix( m, m, c, m, d_c,m,queue); // 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
  gpu_time = magma_sync_wtime(NULL);

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

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  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,queue);  // 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_queue_destroy(queue);                     // destroy queue
  magma_finalize();                              // finalize Magma
  return 0;
}
//magma_ssyrk time: 0.03725 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,...
//........................................
```

4.2.14 magma_ssyrk - unified memory version

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                              // initialize Magma
  magma_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  real_Double_t  gpu_time;
  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
  float *c;                                      // c- mxm matrix
  float alpha = 1.0;                                  // alpha=1
  float beta  = 1.0;                                   // beta=1
  magma_int_t ione = 1;       //random uniform distr. in (0,1)
  magma_int_t ISEED[4] = { 0,1,2,3 };                    // seed
  cudaMallocManaged(&a,mk*sizeof(float)); // unified mem.for a
  cudaMallocManaged(&c,mm*sizeof(float)); // unified mem.for c
// generate random matrices a, c;
  lapackf77_slarnv(&ione,ISEED,&mk,a);           // randomize a
  lapackf77_slarnv(&ione,ISEED,&mm,c);           // randomize c
// upper triangular part of c is the upper triangular part
// of some  matrix, the strictly lower triangular
// part of c is not referenced
// symmetric rank-k update:  c=alpha*a*a^T+beta*c
// c -mxm symmetric matrix,  a -mxk matrix;
// alpha,beta - scalars
  gpu_time = magma_sync_wtime(NULL);

  magma_ssyrk( MagmaUpper,MagmaNoTrans,m,k,alpha,a,m,beta,
                         c,m,queue);

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("magma_ssyrk time: %7.5f sec.\n",gpu_time);
  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(a);                                // free  memory
  magma_free(c);                                // free  memory
  magma_queue_destroy(queue);                  // destroy queue
  magma_finalize();                           // finalize Magma
  return 0;
}
//magma_ssyrk time: 0.09162 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,...
//..............................................
```

4.2.15 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`) or upper (`MagmaUpper`) mode and α, β are scalars. The value of $op(A)$ can be equal to A in `MagmaNoTrans` case or A^T (transposition) in `MagmaTrans` case and similarly for $op(B)$.

```
#include <stdio.h>
#include <cuda.h>
#include "magma_v2.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                                // initialize Magma
  magma_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  real_Double_t  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;        //random uniform distr. in (0,1)
  magma_int_t ISEED[4] = { 0,1,2,3 };                     // seed
  magma_int_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 matrices 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);           // randomize a
  lapackf77_slarnv(&ione,ISEED,&mk,b);           // randomize b
  lapackf77_slarnv(&ione,ISEED,&mm,c);           // randomize c
// upper triangular part of c is the upper triangular part
// of some  matrix, the strictly lower triangular
// part of c is not referenced
// copy data from host to device
  magma_ssetmatrix( m, k, a, m, d_a,m,queue); // copy a -> d_a
  magma_ssetmatrix( m, k, a, m, d_b,m,queue); // copy b -> d_b
  magma_ssetmatrix( m, m, c, m, d_c,m,queue); // copy c -> d_c
// symmetric rank-2k update:
// d_c=alpha*d_a*d_b^T+ alpha*d_b*d_a^T+beta*d_c
// d_c -mxm symmetric matrix,  d_a,d_b -mxk matrices;
// alpha,beta - scalars
  gpu_time = magma_sync_wtime(NULL);

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

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  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,queue); // 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_queue_destroy(queue);                     // destroy queue
  magma_finalize();                             // finalize Magma
  return 0;
}
//magma_ssyr2k time: 0.07446 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,...
//...............................................
```

4.2.16 magma_ssyr2k - unified memory version

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                              // initialize Magma
  magma_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  real_Double_t  gpu_time;
  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
  float *b;                                     // b- mxk matrix
  float *c;                                     // c- mxm matrix
  float alpha = 1.0;                                   // alpha=1
  float beta  = 1.0;                                    // beta=1
  magma_int_t ione = 1;       //random uniform distr. in (0,1)
  magma_int_t ISEED[4] = { 0,1,2,3 };                     // seed
  cudaMallocManaged(&a,mk*sizeof(float)); // unified mem.for a
  cudaMallocManaged(&b,mk*sizeof(float)); // unified mem.for b
  cudaMallocManaged(&c,mm*sizeof(float)); // unified mem.for c
// generate random matrices a,b,c;
  lapackf77_slarnv(&ione,ISEED,&mk,a);          // randomize a
  lapackf77_slarnv(&ione,ISEED,&mk,b);          // randomize b
  lapackf77_slarnv(&ione,ISEED,&mm,c);          // randomize c
// upper triangular part of c is the upper triangular part
// of some  matrix, the strictly lower triangular
// part of c is not referenced

// symmetric rank-2k update:
// c=alpha*a*b^T+ alpha*b*a^T+beta*c
// c -mxm symmetric matrix,  a,b -mxk matrices;
// alpha,beta - scalars
  gpu_time = magma_sync_wtime(NULL);

  magma_ssyr2k( MagmaUpper,MagmaNoTrans,m,k,alpha,a,m,b,m,
                          beta,c,m,queue);

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("magma_ssyr2k time: %7.5f sec.\n",gpu_time);
  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(a);                                          // free  memory
  magma_free(b);                                          // free  memory
  magma_free(c);                                          // free  memory
  magma_queue_destroy(queue);                            // destroy queue
  magma_finalize();                                     // finalize Magma
  return 0;
}
//magma_ssyr2k time: 0.13833 sec.
//after magma_ssyr2k:
//c:
// 2047.3660,...
// 2044.8237, 2041.2444,...
// 2041.6855, 2038.5908, 2023.9705,...
// 2050.2649, 2057.5630, 2046.7908, 2059.3657,...
//.............................................
```

4.2.17 magma_strmm - triangular matrix-matrix multiplication

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

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

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 case, A^T (transposition) in MagmaTrans case or A^H (conjugate transposition) in MagmaConjTrans 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) or upper (MagmaUpper) mode. If the diagonal of the matrix A has non-unit elements, then the parameter MagmaNonUnit should be used (in the opposite case - MagmaUnit).

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include "magma_v2.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                                       // initialize Magma
  magma_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  real_Double_t  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_int_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 matrices 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);          // randomize a
  lapackf77_slarnv(&ione,ISEED,&mn,c);          // randomize c
// upper triangular part of a is the upper triangular part
// of some  matrix, the strictly lower
// triangular part of a is not referenced
// copy data from host to device
  magma_ssetmatrix( m, m, a, m, d_a,m,queue); // copy a -> d_a
  magma_ssetmatrix( m, n, c, m, d_c,m,queue); // 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
  gpu_time = magma_sync_wtime(NULL);

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

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  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,queue); // 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++)  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_queue_destroy(queue);                   // destroy queue
  magma_finalize();                            // finalize Magma
  return 0;
}
//magma_strmm time: 0.04829 sec.
//after magma_strmm:
//c:
```

```
// 2051.0024, 2038.8608, 2033.2482, 2042.2589,...
// 2040.4783, 2027.2789, 2025.2496, 2041.6721,...
// 2077.4158, 2052.2390, 2050.5039, 2074.0791,...
// 2028.7070, 2034.3572, 2003.8625, 2031.4501,...
//...............................................
```

4.2.18 magma_strmm - unified memory version

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                                // initialize Magma
  magma_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  real_Double_t  gpu_time;
  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
  float *c;                                       // c- mxn matrix
  float alpha = 1.0;                                    // alpha=1
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = { 0,1,2,3 };                      // seed
  cudaMallocManaged(&a,mm*sizeof(float)); // unified mem.for a
  cudaMallocManaged(&c,mn*sizeof(float)); // unified mem.for c
// generate random matrices a, c;
  lapackf77_slarnv(&ione,ISEED,&mm,a);           // randomize a
  lapackf77_slarnv(&ione,ISEED,&mn,c);           // randomize c
// upper triangular part of a is the upper triangular part
// of some  matrix, the strictly lower
// triangular part of a is not referenced

// triangular matrix-matrix multiplication  c=alpha*a*c
// c -mxn matrix,  a -mxm triangular matrix;  alpha - scalar
  gpu_time = magma_sync_wtime(NULL);

  magma_strmm(MagmaLeft,MagmaUpper,MagmaNoTrans,MagmaNonUnit,
                        m,n,alpha,a,m,c,m,queue);

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("magma_strmm time: %7.5f sec.\n",gpu_time);
  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(a);                                   // free  memory
  magma_free(c);                                   // free  memory
  magma_queue_destroy(queue);                     // destroy queue
  magma_finalize();                              // finalize Magma
  return 0;
}
//magma_strmm time: 0.12141 sec.
//after magma_strmm:
//c:
// 2051.0024, 2038.8608, 2033.2482, 2042.2589,...
// 2040.4783, 2027.2789, 2025.2496, 2041.6721,...
// 2077.4158, 2052.2390, 2050.5039, 2074.0791,...
// 2028.7070, 2034.3572, 2003.8625, 2031.4501,...
//...............................................
```

4.2.19 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_v2.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                                // initialize Magma
  magma_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  real_Double_t  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_int_t err;
// allocate matrices the host
  err = magma_smalloc_pinned( &a , mn );   // host memory for a
  err = magma_smalloc_pinned( &c , mn );   // host memory for c
// allocate matrices 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);          // randomize a
  lapackf77_slarnv(&ione,ISEED,&mn,c);          // randomize 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,queue);  // copy a -> d_a
  magma_ssetmatrix( m, n, c,m, d_c,m,queue);  // copy c -> d_c
// d_c=alpha*d_a+d_c
// d_a, d_c -mxn matrices;
// alpha - scalar
  gpu_time = magma_sync_wtime(NULL);

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

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  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,queue);  // 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_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_queue_destroy(queue);                  // destroy queue
  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: 0.00174 sec.
//after magmablas_sgeadd:
//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,...
//...............................................
```

4.2.20 magmablas_sgeadd - unified memory version

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                               // initialize Magma
  magma_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  real_Double_t  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
  float *c;                                      // c- mxn matrix
  float alpha = 2.0;                                   // alpha=2
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = { 0,1,2,3 };                     // seed
  cudaMallocManaged(&a,mn*sizeof(float)); // unified mem.for a
  cudaMallocManaged(&c,mn*sizeof(float)); // unified mem.for c
// generate random matrices a, c;
  lapackf77_slarnv(&ione,ISEED,&mn,a);           // randomize a
  lapackf77_slarnv(&ione,ISEED,&mn,c);           // randomize 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");
// c=alpha*a+c;   a, c -mxn matrices;   alpha - scalar
  gpu_time = magma_sync_wtime(NULL);

  magmablas_sgeadd(m,n,alpha,a,m,c,m,queue);

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("magmablas_sgeadd time: %7.5f sec.\n",gpu_time);
  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_free(a);                                // free  memory
  magma_free(c);                                // free  memory
  magma_queue_destroy(queue);                  // destroy queue
  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: 0.03860 sec.
//after magmablas_sgeadd:
//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_v2.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                              // initialize Magma
  real_Double_t  gpu_time;
  magma_int_t *piv, info;
  magma_int_t m = 8192;                        // a - mxm matrix
  magma_int_t n = 100;                         // 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 *b;                      // b- mxn matrix on the host
  float *c;                      // c- mxn matrix on the host
  magma_int_t ione = 1;       //random uniform distr. in (0,1)
  magma_int_t ISEED[4] = { 0,0,0,1 };                   // seed
  magma_int_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( &b , mn );  // host memory for b
  err = magma_smalloc_pinned( &c , mn );  // host memory for c
  piv=(magma_int_t*)malloc(m*sizeof(magma_int_t));
// generate random matrices a, b;
  lapackf77_slarnv(&ione,ISEED,&mm,a);          // randomize a
  lapackf77_slarnv(&ione,ISEED,&mn,b);          // randomize b
  printf("upper left corner of the expected solution:\n");
  magma_sprint( 4, 4, b, m );
// right hand side c=a*b
  blasf77_sgemm("N","N",&m,&n,&n,&alpha,a,&m,b,&m,&beta,c,&m);
// solve the linear system a*x=c
// c -mxn matrix,  a -mxm  matrix;
// c is overwritten by the solution
  gpu_time = magma_sync_wtime(NULL);

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

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("magma_sgesv time: %7.5f sec.\n",gpu_time);   // time
  printf("upper left corner of the solution:\n");
  magma_sprint( 4, 4, c, m );          // part of the  solution
  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:
//[
//    0.3924   0.5546   0.6481   0.5479
//    0.9790   0.7204   0.4220   0.4588
//    0.5246   0.0813   0.8202   0.6163
//    0.6624   0.8634   0.8748   0.0717
//];
//magma_sgesv time: 0.61733 sec.
//upper left corner of the solution:
//[
//    0.3927   0.5548   0.6484   0.5483
//    0.9788   0.7204   0.4217   0.4586
//    0.5242   0.0815   0.8199   0.6161
//    0.6626   0.8631   0.8749   0.0717
//];
```

4.3.2 magma_sgesv - unified memory version

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                            // initialize Magma
  magma_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  real_Double_t  gpu_time;
  magma_int_t *piv, info;
  magma_int_t m = 8192;                      // a - mxm matrix
  magma_int_t n = 100;                       // 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
  float *b;                                   // b- mxn matrix
  float *c;                                   // c- mxn matrix
  magma_int_t ione = 1;       //random uniform distr. in (0,1)
  magma_int_t ISEED[4] = { 0,0,0,1 };                   // seed
  const float alpha = 1.0;                           // alpha=1
  const float beta = 0.0;                             // beta=0
  cudaMallocManaged(&a,mm*sizeof(float)); // unified mem.for a
  cudaMallocManaged(&b,mn*sizeof(float)); // unified mem.for b
  cudaMallocManaged(&c,mn*sizeof(float)); // unified mem.for c
  cudaMallocManaged(&piv,m*sizeof(int));// unified mem.for piv
// generate random matrices a, b;
```

```
  lapackf77_slarnv(&ione,ISEED,&mm,a);          // randomize a
  lapackf77_slarnv(&ione,ISEED,&mn,b);          // randomize b
  printf("expected solution:\n");
  magma_sprint( 4, 4, b, m );
// right hand side c=a*b
  blasf77_sgemm("N","N",&m,&n,&n,&alpha,a,&m,b,&m,&beta,c,&m);
// solve the linear system a*x=c
// c -mxn matrix,  a -mxm  matrix;
// c is overwritten by the solution
  gpu_time = magma_sync_wtime(NULL);

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

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("magma_sgesv time: %7.5f sec.\n",gpu_time);   // time
  printf("solution:\n");
  magma_sprint( 4, 4, c, m );         // part of the  solution
  magma_free(a);                                // free memory
  magma_free(b);                                // free memory
  magma_free(c);                                // free memory
  magma_free(piv);                              // free memory
  magma_finalize();                           // finalize Magma
  return 0;
}
//expected solution:
//[
//   0.3924   0.5546   0.6481   0.5479
//   0.9790   0.7204   0.4220   0.4588
//   0.5246   0.0813   0.8202   0.6163
//   0.6624   0.8634   0.8748   0.0717
//];
//magma_sgesv time: 0.42720 sec.
//solution:
//[
//   0.3927   0.5548   0.6484   0.5483
//   0.9788   0.7204   0.4217   0.4586
//   0.5242   0.0815   0.8199   0.6161
//   0.6626   0.8631   0.8749   0.0717
//];
```

4.3.3 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_v2.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                              // initialize Magma
  real_Double_t  gpu_time;
  magma_int_t *piv, info;
  magma_int_t m = 8192;                        // a - mxm matrix
  magma_int_t n = 100;                         // c - mxn matrix
  magma_int_t mm=m*m;                                // size of a
  magma_int_t mn=m*n;                                // size of 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;       //random uniform distr. in (0,1)
  magma_int_t ISEED[4] = { 0,0,0,1 };                   // seed
  magma_int_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( &b , mn );  // host memory for b
  err = magma_dmalloc_pinned( &c , mn );  // host memory for c
  piv=(magma_int_t*)malloc(m*sizeof(magma_int_t));
// generate random matrices a, b;
  lapackf77_dlarnv(&ione,ISEED,&mm,a);          // randomize a
  lapackf77_dlarnv(&ione,ISEED,&mn,b);          // randomize b
  printf("expected solution:\n");
  magma_dprint( 4, 4, b, m );
// right hand side c=a*b
  blasf77_dgemm("N","N",&m,&n,&n,&alpha,a,&m,b,&m,&beta,c,&m);
// solve the linear system a*x=c
// c -mxn matrix,  a -mxm  matrix;
// c is overwritten by the solution
  gpu_time = magma_sync_wtime(NULL);

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

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("magma_dgesv time: %7.5f sec.\n",gpu_time);   // time
  printf("solution:\n");
  magma_dprint( 4, 4, c, m );          // part of the  solution
  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;
}
//expected solution:
```

```
//[
//   0.5440   0.5225   0.8499   0.4012
//   0.4878   0.9321   0.2277   0.7495
//   0.0124   0.7743   0.5884   0.3296
//   0.2166   0.6253   0.8843   0.3685
//];
//magma_dgesv time: 1.81342 sec.
//solution:
//[
//   0.5440   0.5225   0.8499   0.4012
//   0.4878   0.9321   0.2277   0.7495
//   0.0124   0.7743   0.5884   0.3296
//   0.2166   0.6253   0.8843   0.3685
//];
```

4.3.4 magma_dgesv - unified memory version

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                            // initialize Magma
  magma_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  real_Double_t  gpu_time;
  magma_int_t *piv, info;
  magma_int_t m = 8192;                      // a - mxm matrix
  magma_int_t n = 100;                       // c - mxn matrix
  magma_int_t mm=m*m;                              // size of a
  magma_int_t mn=m*n;                              // size of c
  double *a;                                   // a- mxm matrix
  double *b;                                   // b- mxn matrix
  double *c;                                   // c- mxn matrix
  magma_int_t ione = 1;        //random uniform distr. in (0,1)
  magma_int_t ISEED[4] = { 0,0,0,1 };                    // seed
  const double alpha = 1.0;                           // alpha=1
  const double beta = 0.0;                             // beta=0
  cudaMallocManaged(&a,mm*sizeof(double));// unified mem.for a
  cudaMallocManaged(&b,mn*sizeof(double));// unified mem.for b
  cudaMallocManaged(&c,mn*sizeof(double));// unified mem.for c
  cudaMallocManaged(&piv,m*sizeof(int));// unified mem.for piv
// generate random matrices a, b;
  lapackf77_dlarnv(&ione,ISEED,&mm,a);           // randomize a
  lapackf77_dlarnv(&ione,ISEED,&mn,b);           // randomize b
  printf("expected solution:\n");
  magma_dprint( 4, 4, b, m );
// right hand side c=a*b
```

```
  blasf77_dgemm("N","N",&m,&n,&n,&alpha,a,&m,b,&m,&beta,c,&m);
// solve the linear system a*x=c
// c -mxn matrix,  a -mxm  matrix;
// c is overwritten by the solution
  gpu_time = magma_sync_wtime(NULL);

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

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("magma_dgesv time: %7.5f sec.\n",gpu_time);   // time
  printf("solution:\n");
  magma_dprint( 4, 4, c, m );          // part of the  solution
  magma_free(a);                                    // free memory
  magma_free(b);                                    // free memory
  magma_free(c);                                    // free memory
  magma_free(piv);                                  // free memory
  magma_finalize();                               // finalize Magma
  return 0;
}
//expected solution:
//[
//   0.5440   0.5225   0.8499   0.4012
//   0.4878   0.9321   0.2277   0.7495
//   0.0124   0.7743   0.5884   0.3296
//   0.2166   0.6253   0.8843   0.3685
//];
//magma_dgesv time: 1.69905 sec.
//solution:
//[
//   0.5440   0.5225   0.8499   0.4012
//   0.4878   0.9321   0.2277   0.7495
//   0.0124   0.7743   0.5884   0.3296
//   0.2166   0.6253   0.8843   0.3685
//];
```

4.3.5 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_v2.h"
#include "magma_lapack.h"
```

```
int main( int argc, char** argv ){
  magma_init();                                 // initialize Magma
  magma_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  real_Double_t  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,d_a
  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_int_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                                  // for piv
  lapackf77_slarnv(&ione,ISEED,&mm,a);          // randomize a
  lapackf77_slarnv(&ione,ISEED,&mn,b);          // randomize b
  printf("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,queue); // copy a -> d_a
  magma_ssetmatrix( m, n, c, m, d_c,m,queue); // 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)
  gpu_time = magma_sync_wtime(NULL);

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

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("magma_sgesv_gpu time: %7.5f sec.\n",gpu_time);
  magma_sgetmatrix( m, n, d_c, m, c,m,queue);
  printf("solution:\n");
  magma_sprint( 4, 4, c, m );        // part of 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_queue_destroy(queue);                    // destroy queue
  magma_finalize();                             // finalize Magma
  return 0;
}
//expected solution:
//[
//   0.3924   0.5546   0.6481   0.5479
//   0.9790   0.7204   0.4220   0.4588
//   0.5246   0.0813   0.8202   0.6163
//   0.6624   0.8634   0.8748   0.0717
//];
//magma_sgesv_gpu time: 0.29100 sec.
//solution:
//[
//   0.3546   0.5629   0.6696   0.4666
//   1.0140   0.7044   0.4187   0.4630
//   0.5813   0.0568   0.8220   0.5983
//   0.6398   0.8704   0.8650   0.1062
//];
```

4.3.6 magma_sgesv_gpu - unified memory version

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                               // initialize Magma
  magma_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  real_Double_t  gpu_time;
  magma_int_t *piv, info;  // piv - array of indices of inter-
  magma_int_t m = 8192;        // changed rows; 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
  float *b;                                   // b- mxn matrix
  float *c;                                   // c- mxn matrix
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = { 0,0,0,1 };                      // seed
  const float alpha = 1.0;                              // alpha=1
  const float beta = 0.0;                                // beta=0
// allocate matrices
```

```
  cudaMallocManaged(&a,mm*sizeof(float)); // unified mem.for a
  cudaMallocManaged(&b,mn*sizeof(float)); // unified mem.for b
  cudaMallocManaged(&c,mn*sizeof(float)); // unified mem.for c
  cudaMallocManaged(&piv,m*sizeof(int));// unified mem.for piv
// generate  matrices
  lapackf77_slarnv(&ione,ISEED,&mm,a);          // randomize a
  lapackf77_slarnv(&ione,ISEED,&mn,b);          // randomize b
  printf("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, a -mxm matrix,
// c -mxn matrix, c is overwritten by the solution;
// LU decomposition with partial pivoting and row
// interchanges is used, row i is interchanged with row piv(i)
  gpu_time = magma_sync_wtime(NULL);

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

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("magma_sgesv_gpu time: %7.5f sec.\n",gpu_time);
  printf("solution:\n");
  magma_sprint( 4, 4, c, m );        // part of Magma solution
  magma_free(piv);                            // free  memory
  magma_free(a);                               // free memory
  magma_free(b);                               // free memory
  magma_free(c);                               // free memory
  magma_queue_destroy(queue);               // destroy queue
  magma_finalize();                        // finalize Magma
  return 0;
}
//expected solution:
//[
//   0.3924   0.5546   0.6481   0.5479
//   0.9790   0.7204   0.4220   0.4588
//   0.5246   0.0813   0.8202   0.6163
//   0.6624   0.8634   0.8748   0.0717
//];
//magma_sgesv_gpu time: 0.31976 sec.
//solution:
//[
//   0.3546   0.5629   0.6696   0.4666
//   1.0140   0.7044   0.4187   0.4630
//   0.5813   0.0568   0.8220   0.5983
//   0.6398   0.8704   0.8650   0.1062
//];
```

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

This function solves in double precision a general,f 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_v2.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                                  // initialize Magma
  magma_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  real_Double_t  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,d_a
  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_int_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);           // randomize a
  lapackf77_dlarnv(&ione,ISEED,&mn,b);           // randomize b
  printf("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,queue); // copy a -> d_a
  magma_dsetmatrix( m, n, c, m, d_c,m,queue); // 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)
  gpu_time = magma_sync_wtime(NULL);

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

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("magma_dgesv_gpu time: %7.5f sec.\n",gpu_time);
  magma_dgetmatrix( m, n, d_c, m, c,m,queue);
  printf("solution:\n");
  magma_dprint( 4, 4, c, m );         // part of 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_queue_destroy(queue);                    // destroy queue
  magma_finalize();                             // finalize Magma
  return 0;
}
//expected solution:
//[
//   0.5440   0.5225   0.8499   0.4012
//   0.4878   0.9321   0.2277   0.7495
//   0.0124   0.7743   0.5884   0.3296
//   0.2166   0.6253   0.8843   0.3685
//];
//magma_dgesv_gpu time: 1.47404 sec.
//solution:
//[
//   0.5440   0.5225   0.8499   0.4012
//   0.4878   0.9321   0.2277   0.7495
//   0.0124   0.7743   0.5884   0.3296
//   0.2166   0.6253   0.8843   0.3685
//];
```

4.3.8 magma_dgesv_gpu - unified memory version

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                                // initialize Magma
  magma_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  real_Double_t  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
  magma_int_t mn=m*n;                            // size of b,c
  double *a;                                  // a- mxm matrix
  double *b;                                  // b- mxn matrix
  double *c;                                  // c- mxn matrix
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = { 0,0,0,1 };                   // seed
  const double alpha = 1.0;                          // alpha=1
  const double beta = 0.0;                            // beta=0
// allocate matrices
  cudaMallocManaged(&a,mm*sizeof(double));// unified mem.for a
  cudaMallocManaged(&b,mn*sizeof(double));// unified mem.for b
  cudaMallocManaged(&c,mn*sizeof(double));// unified mem.for c
  cudaMallocManaged(&piv,m*sizeof(int));// unified mem.for piv
// generate  matrices
  lapackf77_dlarnv(&ione,ISEED,&mm,a);           // randomize a
  lapackf77_dlarnv(&ione,ISEED,&mn,b);           // randomize b
  printf("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, a -mxm matrix,
// c -mxn matrix, c is overwritten by the solution;
// LU decomposition with partial pivoting and row
// interchanges is used, row i is interchanged with row piv(i)
  gpu_time = magma_sync_wtime(NULL);

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

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("magma_dgesv_gpu time: %7.5f sec.\n",gpu_time);
  printf("solution:\n");
  magma_dprint( 4, 4, c, m );        // part of Magma solution
  magma_free(piv);                            // free  memory
  magma_free(a);                               // free memory
  magma_free(b);                               // free memory
  magma_free(c);                               // free memory
  magma_queue_destroy(queue);                 // destroy queue
  magma_finalize();                          // finalize Magma
  return 0;
}

//expected solution:
```

```
//[
//   0.5440   0.5225   0.8499   0.4012
//   0.4878   0.9321   0.2277   0.7495
//   0.0124   0.7743   0.5884   0.3296
//   0.2166   0.6253   0.8843   0.3685
//];
//magma_dgesv_gpu time: 1.55957 sec.
//solution:
//[
//   0.5440   0.5225   0.8499   0.4012
//   0.4878   0.9321   0.2277   0.7495
//   0.0124   0.7743   0.5884   0.3296
//   0.2166   0.6253   0.8843   0.3685
//];
```

4.3.9 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_v2.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                               // initialize Magma
  real_Double_t  gpu_time;
  magma_int_t *piv, info;  // piv - array of indices of inter-
  magma_int_t m = 8192,n=8192; // changed rows; a - mxn matrix
  magma_int_t nrhs = 100;   // b - nxnrhs, c - mxnrhs 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- mxn matrix on the host
  float *b;                      // b- nxnrhs matrix on the host
  float *c;                    // c- mxnrhs matrix on the host
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = {0,0,0,1};                       // seed
  magma_int_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);           // randomize a
  lapackf77_slaset(MagmaFullStr,&n,&nrhs,&alpha,&alpha,b,&n);
                                   // b - nxnrhs 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 -mxn matrix, c -mxnrhs 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)
  gpu_time = magma_sync_wtime(NULL);

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

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("magma_sgetrf + lapackf77_sgetrs time: %7.5f sec.\n",
                             gpu_time); // Magma/Lapack time
  printf("upper left corner of the 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: 0.77011 sec.
//upper left corner of the solution:
```

```
//[
//   0.9682   0.9682   0.9682   0.9682
//   1.0134   1.0134   1.0134   1.0134
//   1.0147   1.0147   1.0147   1.0147
//   1.0034   1.0034   1.0034   1.0034
//];
```

4.3.10 magma_sgetrf, lapackf77_sgetrs - unified memory version

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                                  // initialize Magma
  real_Double_t  gpu_time;
  magma_int_t *piv, info;  // piv - array of indices of inter-
  magma_int_t m = 8192,n=8192; // changed rows; a - mxn matrix
  magma_int_t nrhs = 100;   // b - nxnrhs, c - mxnrhs 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- mxn matrix
  float *b;                                     // b- nxnrhs matrix
  float *c;                                     // c- mxnrhs matrix
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = {0,0,0,1};                         // seed
  const float alpha = 1.0;                               // alpha=1
  const float beta = 0.0;                                 // beta=0
  cudaMallocManaged(&a,mn*sizeof(float)); // unified mem.for a
  cudaMallocManaged(&b,nnrhs*sizeof(float)); //unif. mem.for b
  cudaMallocManaged(&c,mnrhs*sizeof(float)); //unif. mem.for c
  cudaMallocManaged(&piv,m*sizeof(int));// unified mem.for piv
// generate  matrices
  lapackf77_slarnv(&ione,ISEED,&mn,a);          // randomize a
  lapackf77_slaset(MagmaFullStr,&n,&nrhs,&alpha,&alpha,b,&n);
                                  // b - nxnrhs 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);    // right hand side c=a*b
// solve the linear system a*x=c, a -mxn matrix, c -mxnrhs 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)
  gpu_time = magma_sync_wtime(NULL);
```

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

    gpu_time = magma_sync_wtime(NULL)-gpu_time;
    printf("magma_sgetrf + lapackf77_sgetrs time: %7.5f sec.\n",
                                 gpu_time); // Magma/Lapack time
    printf("upper left corner of the Magma solution:\n");
    magma_sprint( 4, 4, c, m );//part of the Magma/Lap. solution
    magma_free(a);                                // free memory
    magma_free(b);                                // free memory
    magma_free(c);                                // free memory
    magma_free(piv);                              // free 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: 0.80166 sec.
//upper left corner of the Magma solution:
//[
//    0.9682    0.9682    0.9682    0.9682
//    1.0134    1.0134    1.0134    1.0134
//    1.0147    1.0147    1.0147    1.0147
//    1.0034    1.0034    1.0034    1.0034
//];
```

4.3.11 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_v2.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                              // initialize Magma
  real_Double_t  gpu_time;
  magma_int_t *piv, info;  // piv - array of indices of inter-
  magma_int_t m = 8192,n=8192; // changed rows; a - mxn matrix
  magma_int_t nrhs = 100;   // b - nxnrhs, c - mxnrhs 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- mxn matrix on the host
  double *b;                   // b- nxnrhs matrix on the host
  double *c;                   // c- mxnrhs matrix on the host
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = {0,0,0,1};                      // seed
  magma_int_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);          // randomize a
  lapackf77_dlaset(MagmaFullStr,&n,&nrhs,&alpha,&alpha,b,&n);
                                 // b - nxnrhs 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 -mxn matrix, c -mxnrhs 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)
  gpu_time = magma_sync_wtime(NULL);

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

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("magma_dgetrf + lapackf77_dgetrs time: %7.5f sec.\n",
                              gpu_time); // Magma/Lapack time
  printf("upper left corner of the Magma solution:\n");
  magma_dprint( 4, 4, c, m );//part of the Magma/Lap. solution
  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: 1.89429 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
//];
```

4.3.12 magma_dgetrf, lapackf77_dgetrs - unified memory version

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                               // initialize Magma
  real_Double_t  gpu_time;
  magma_int_t *piv, info;  // piv - array of indices of inter-
  magma_int_t m = 8192,n=8192; // changed rows; a - mxn matrix
  magma_int_t nrhs = 100;   // b - nxnrhs, c - mxnrhs 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- mxn matrix
  double *b;                                  // b- nxnrhs matrix
  double *c;                                  // c- mxnrhs matrix
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = {0,0,0,1};                       // seed
  const double alpha = 1.0;                            // alpha=1
  const double beta = 0.0;                              // beta=0
  cudaMallocManaged(&a,mn*sizeof(double));// unified mem.for a
  cudaMallocManaged(&b,nnrhs*sizeof(double));// unif.mem.for b
  cudaMallocManaged(&c,mnrhs*sizeof(double));// unif.mem.for c
  cudaMallocManaged(&piv,m*sizeof(int));// unified mem.for piv
// generate  matrices
  lapackf77_dlarnv(&ione,ISEED,&mn,a);          // randomize a
  lapackf77_dlaset(MagmaFullStr,&n,&nrhs,&alpha,&alpha,b,&n);
                                      // b - nxnrhs 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
// solve the linear system a*x=c, a -mxn matrix, c -mxnrhs 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)
  gpu_time = magma_sync_wtime(NULL);

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

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("magma_dgetrf + lapackf77_dgetrs time: %7.5f sec.\n",
                                gpu_time); // Magma/Lapack time
  printf("upper left corner of the Magma solution:\n");
  magma_dprint( 4, 4, c, m );//part of the Magma/Lap. solution
  magma_free(a);                               // free memory
  magma_free(b);                               // free memory
  magma_free(c);                               // free memory
  magma_free(piv);                             // free 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: 2.03707 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
//];
```

4.3.13 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_v2.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                              // initialize Magma
  magma_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  real_Double_t  gpu_time;
  magma_int_t *piv, info;  // piv - array of indices of inter-
  magma_int_t m = 8192,n=8192; // changed rows; a - mxn matrix
  magma_int_t nrhs = 100;   // b - nxnrhs, c - mxnrhs 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- mxn matrix on the host
  float *b;                       // b- nxnrhs matrix on the host
  float *c;                       // c- mxnrhs matrix on the host
  float *d_a;                  // d_a- mxn matrix a on the device
  float *d_c;               // d_c- mxnrhs matrix c on the device
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = {0,0,0,1};                       // seed
  magma_int_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 , 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);          // randomize a
  lapackf77_slaset(MagmaFullStr,&n,&nrhs,&alpha,&alpha,b,&n);
                                 // b - nxnrhs matrix of ones
  printf("upper left corner of the expected solution:\n");
  magma_sprint( 4, 4, b, n ); // 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,queue);   // copy a -> d_a
  magma_ssetmatrix( m, nrhs,c,m,d_c,m,queue); // copy c -> d_c
// MAGMA
// solve the linear system d_a*x=d_c, d_a -mxn matrix,
// d_c -mxnrhs 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)
  gpu_time = magma_sync_wtime(NULL);

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

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  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,queue);
  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_queue_destroy(queue);                   // destroy queue
  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: 0.28847 sec.
//upper left corner of the Magma solution:
//[
//   1.0431   1.0431   1.0431   1.0431
//   1.0446   1.0446   1.0446   1.0446
//   1.1094   1.1094   1.1094   1.1094
//   0.9207   0.9207   0.9207   0.9207
//];
```

4.3.14 magma_sgetrf_gpu, magma_sgetrs_gpu - unified memory version

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                              // initialize Magma
  magma_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  real_Double_t  gpu_time;
  magma_int_t *piv, info;  // piv - array of indices of inter-
  magma_int_t m = 8192,n=8192; // changed rows; a - mxn matrix
  magma_int_t nrhs = 100;   // b - nxnrhs, c - mxnrhs 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- mxn matrix
  float *b;                                  // b- nxnrhs matrix
  float *c;                                  // c- mxnrhs matrix
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = { 0,0,0,1 };                    // seed
  const float alpha = 1.0;                            // alpha=1
  const float beta = 0.0;                              // beta=0
  cudaMallocManaged(&a,mn*sizeof(float)); // unified mem.for a
  cudaMallocManaged(&b,nnrhs*sizeof(float));// unif. mem.for b
  cudaMallocManaged(&c,mnrhs*sizeof(float));// unif. mem.for c
  cudaMallocManaged(&piv,m*sizeof(int));// unified mem.for piv
// generate  matrices a, b;
  lapackf77_slarnv(&ione,ISEED,&mn,a);          // randomize a
  lapackf77_slaset(MagmaFullStr,&n,&nrhs,&alpha,&alpha,b,&n);
                                     // b - nxnrhs 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);
// solve the linear system a*x=c, a -mxn matrix,
// c -mxnrhs matrix, c is overwritten by the solution;
// LU decomposition with partial pivoting and row interchanges
// is used, row i is interchanged with row piv(i)
  gpu_time = magma_sync_wtime(NULL);

  magma_sgetrf_gpu( m, n, a, m, piv, &info);
  magma_sgetrs_gpu(MagmaNoTrans,m,nrhs,a,m,piv,c,m,&info);

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("magma_sgetrf_gpu+magma_sgetrs_gpu time: %7.5f sec.\n",
                                     gpu_time);  // Magma time
  printf("upper left corner of the solution:\n");
  magma_sprint( 4, 4, c, m );    // part of the Magma solution
  magma_free(piv);                              // free memory
  magma_free(a);                               // free  memory
  magma_free(b);                               // free  memory
  magma_free(c);                               // free  memory
  magma_queue_destroy(queue);                 // destroy queue
  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: 0.31721 sec.
//upper left corner of the solution:
//[
//   1.0431   1.0431   1.0431   1.0431
//   1.0446   1.0446   1.0446   1.0446
//   1.1094   1.1094   1.1094   1.1094
//   0.9207   0.9207   0.9207   0.9207
//];
```

4.3.15 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_v2.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                            // initialize Magma
  magma_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  real_Double_t  gpu_time;
  magma_int_t *piv, info;  // piv - array of indices of inter-
  magma_int_t m = 8192,n=8192; // changed rows; a - mxn matrix
  magma_int_t nrhs = 100;   // b - nxnrhs, c - mxnrhs 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- mxn matrix on the host
  double *b;                      // b- nxnrhs matrix on the host
  double *c;                      // c- mxnrhs matrix on the host
  double *d_a;                  // d_a- mxn matrix a on the device
  double *d_c;                // d_c- mxnrhs matrix c on the device
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = { 0,0,0,1 };                      // seed
  magma_int_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 , 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);           // randomize a
  lapackf77_dlaset(MagmaFullStr,&n,&nrhs,&alpha,&alpha,b,&n);
                                  // b - nxnrhs 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,queue);// copy a -> d_a
  magma_dsetmatrix( m, nrhs, c,m,d_c,m,queue);// copy c -> d_c
// MAGMA
// solve the linear system d_a*x=d_c, d_a -mxn matrix,
// d_c -mxnrhs 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)
  gpu_time = magma_sync_wtime(NULL);

  magma_dgetrf_gpu( m, n, d_a, m, piv, &info);
  magma_dgetrs_gpu(MagmaNoTrans,m,nrhs,d_a,m,piv,d_c,m,&info);
  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  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,queue);
  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_queue_destroy(queue);              // destroy queue
  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: 1.47816 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
//];
```

4.3.16 magma_dgetrf_gpu, magma_dgetrs_gpu - unified memory version

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                          // initialize Magma
  magma_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  real_Double_t  gpu_time;
  magma_int_t *piv, info;  // piv - array of indices of inter-
  magma_int_t m = 8192,n=8192; // changed rows; a - mxn matrix
  magma_int_t nrhs = 100;   // b - nxnrhs, c - mxnrhs 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- mxn matrix
  double *b;                                  // b- nxnrhs matrix
  double *c;                                  // c- mxnrhs matrix
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = { 0,0,0,1 };                     // seed
  const double alpha = 1.0;                            // alpha=1
  const double beta = 0.0;                              // beta=0
  cudaMallocManaged(&a,mn*sizeof(double));// unified mem.for a
  cudaMallocManaged(&b,nnrhs*sizeof(double));// unif.mem.for b
  cudaMallocManaged(&c,mnrhs*sizeof(double));// unif.mem.for c
 cudaMallocManaged(&piv,m*sizeof(int)); // unified mem.for piv
// generate  matrices a, b;
  lapackf77_dlarnv(&ione,ISEED,&mn,a);          // randomize a
  lapackf77_dlaset(MagmaFullStr,&n,&nrhs,&alpha,&alpha,b,&n);
                                  // b - nxnrhs 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);
// solve the linear system a*x=c, a -mxn matrix,
// c -mxnrhs matrix, c is overwritten by the solution;
// LU decomposition with partial pivoting and row interchanges
// is used, row i is interchanged with row piv(i)
  gpu_time = magma_sync_wtime(NULL);

  magma_dgetrf_gpu( m, n, a, m, piv, &info);
  magma_dgetrs_gpu(MagmaNoTrans,m,nrhs,a,m,piv,c,m,&info);

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("magma_dgetrf_gpu+magma_dgetrs_gpu time: %7.5f sec.\n",
                                     gpu_time);  // Magma time
  printf("upper left corner of the solution:\n");
  magma_dprint( 4, 4, c, m );    // part of the Magma solution
  magma_free(piv);                               // free memory
  magma_free(a);                                // free  memory
  magma_free(b);                                // free  memory
  magma_free(c);                                // free  memory
  magma_queue_destroy(queue);                  // destroy queue
  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: 1.51280 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
//];
```

4.3.17 magma_sgetrf_mgpu - LU factorization in single precision on multiple GPUs

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_v2.h"
#include "magma_lapack.h"
#define min(a,b)  (((a)<(b))?(a):(b))
int main( int argc, char** argv)
{
  magma_init();                                // initialize Magma
  int num_gpus = 1;
  magma_setdevice(0);
  magma_queue_t queues[num_gpus];
  for( int dev = 0; dev < num_gpus; ++dev ) {
    magma_queue_create( dev, &queues[dev] );
  }
  magma_int_t err;
  real_Double_t  cpu_time,gpu_time;
  magma_int_t  m = 8192, n = 8192;        // a,r - mxn matrices
  magma_int_t  nrhs =100;    // b - nxnrhs, c - mxnrhs 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
  float *a, *r;              // a,r - mxn matrices on the host
  float *b, *c; // b - nxnrhs, c - mxnrhs matrices on the host
  magmaFloat_ptr d_la[num_gpus];
  float alpha=1.0, beta=0.0;                     // alpha=1,beta=0
  magma_int_t  n_local;
  magma_int_t ione = 1, info;
  magma_int_t i, min_mn=min(m,n), nb;
  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,n);  //optim.block size for sgetrf
// allocate memory on cpu
  ipiv=(magma_int_t*)malloc(min_mn*sizeof(magma_int_t));
                                         // host memory for ipiv
  err = magma_smalloc_cpu(&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;
    magma_setdevice(i);
    err = magma_smalloc(&d_la[i],m*ldn_local); //device memory
  }                                              // on i-th device
  magma_setdevice(0);
// generate matrices
  lapackf77_slarnv( &ione, ISEED, &n2, a );     // randomize a
  lapackf77_slaset(MagmaFullStr,&n,&nrhs,&alpha,&alpha,b,&n);
                                    // b - nxnrhs matrix of ones
  lapackf77_slacpy( MagmaFullStr,&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
  cpu_time=magma_wtime();
  lapackf77_sgetrf(&m, &n, a, &m, ipiv, &info);
  cpu_time=magma_wtime()-cpu_time;
  printf("lapackf77_sgetrf time: %7.5f sec.\n",cpu_time);
// copy the corresponding parts of the matrix r to num_gpus
  magma_ssetmatrix_1D_col_bcyclic( num_gpus, m, n, nb, r, m,
                                           d_la, m, queues );
// MAGMA
// LU decomposition on num_gpus devices with partial  pivoting
// and row interchanges, row i is interchanged with row ipiv(i)
```

```
  gpu_time = magma_sync_wtime(NULL);

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

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("magma_sgetrf_mgpu time: %7.5f sec.\n",gpu_time);
// copy the decomposition from num_gpus devices to r on the
// host
  magma_sgetmatrix_1D_col_bcyclic( num_gpus, m, n, nb, d_la,
                                          m, r, m, queues );
  magma_setdevice(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
  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
  }
   for( int dev = 0; dev < num_gpus; ++dev ) {
        magma_queue_destroy( queues[dev] );
    }
  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: 1.39675 sec.
//magma_sgetrf_mgpu time: 0.28165 sec.
//upper left corner of the solution
//  from sgetrf_mgpu+sgetrs:
//[
//   0.9682   0.9682   0.9682   0.9682
//   1.0134   1.0134   1.0134   1.0134
//   1.0147   1.0147   1.0147   1.0147
//   1.0034   1.0034   1.0034   1.0034
//];
```

4.3.18 magma_dgetrf_mgpu - LU factorization in double precision on multiple GPUs

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_v2.h"
#include "magma_lapack.h"
#define min(a,b)  (((a)<(b))?(a):(b))
int main( int argc, char** argv)
{
  magma_init();                              // initialize Magma
  int num_gpus = 1;
  magma_setdevice(0);
  magma_queue_t queues[num_gpus];
  for( int dev = 0; dev < num_gpus; ++dev ) {
    magma_queue_create( dev, &queues[dev] );
  }
  magma_int_t err;
  real_Double_t  cpu_time,gpu_time;
  magma_int_t  m = 8192, n = 8192;        // a,r - mxn matrices
  magma_int_t  nrhs =100;   // b - nxnrhs, c - mxnrhs 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
  double *a, *r;             // a,r - mxn matrices on the host
  double *b, *c;// b - nxnrhs, c - mxnrhs matrices on the host
  magmaDouble_ptr d_la[num_gpus];
  double alpha=1.0, beta=0.0;                  // alpha=1,beta=0
  magma_int_t  n_local;
  magma_int_t ione = 1, info;
  magma_int_t i, min_mn=min(m,n), nb;
  magma_int_t ldn_local;// mxldn_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,n);  //optim.block size 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;
    magma_setdevice(i);
    err = magma_dmalloc(&d_la[i],m*ldn_local); //device memory
  }                                             // on i-th device
  magma_setdevice(0);
// generate matrices
  lapackf77_dlarnv( &ione, ISEED, &n2, a );     // randomize a
  lapackf77_dlaset(MagmaFullStr,&n,&nrhs,&alpha,&alpha,b,&n);
                                    // b - nxnrhs matrix of ones
  lapackf77_dlacpy( MagmaFullStr,&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
  cpu_time=magma_wtime();
  lapackf77_dgetrf(&m, &n, a, &m, ipiv, &info);
  cpu_time=magma_wtime()-cpu_time;
  printf("lapackf77_dgetrf time: %7.5f sec.\n",cpu_time);
// copy the corresponding parts of the matrix r to num_gpus
  magma_dsetmatrix_1D_col_bcyclic( num_gpus, m, n, nb, r, m,
                                        d_la, m, queues );
// MAGMA
// LU decomposition on num_gpus devices with partial  pivoting
// and row interchanges, row i is interchanged with row ipiv(i)
  gpu_time = magma_sync_wtime(NULL);

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

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("magma_dgetrf_mgpu time: %7.5f sec.\n",gpu_time);
// copy the decomposition from num_gpus devices to r on the
// host
  magma_dgetmatrix_1D_col_bcyclic( num_gpus, m, n, nb, d_la,
                                              m, r, m, queues );
  magma_setdevice(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
  }
   for( int dev = 0; dev < num_gpus; ++dev ) {
        magma_queue_destroy( queues[dev] );
    }
  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: 2.82328 sec.
//magma_dgetrf_mgpu time: 1.41692 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
//];
```

4.3.19 magma_sgetri_gpu - inverse matrix in single precision, GPU interface

This function computes in single precision the inverse A^{-1} of an $m \times m$ 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_v2.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                              // initialize Magma
  magma_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  double   gpu_time;
  float    *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_int_t err;
  const float alpha = 1.0;                             // alpha=1
  const float beta = 0.0;                               // beta=0
  ldwork = m * magma_get_sgetri_nb( m ); // optimal block 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);          // randomize a
  magma_ssetmatrix( m, m, a,m, d_a,m,queue);  // copy a -> d_a
  magmablas_slacpy(MagmaFull,m,m,d_a,m,d_r,m,queue);//d_a->d_r
// find the inverse matrix: d_a*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
  gpu_time = magma_sync_wtime(NULL);

  magma_sgetrf_gpu( m, m, d_a, m, piv, &info);
  magma_sgetri_gpu(m,d_a,m,piv,dwork,ldwork,&info);
  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  magma_sgemm(MagmaNoTrans,MagmaNoTrans,m,m,m,alpha,d_a,m,
                 d_r,m,beta,d_c,m,queue); // multiply a^-1*a
  printf("magma_sgetrf_gpu + magma_sgetri_gpu time: %7.5f sec.\
                                                     \n",gpu_time);
  magma_sgetmatrix( m, m, d_c, m, a, m, queue); // 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
  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_queue_destroy(queue);                       // destroy queue
  magma_finalize();                                // finalize Magma
  return 0;
}
//magma_sgetrf_gpu + magma_sgetri_gpu time: 0.58294 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
//];
```

4.3.20 magma_sgetri_gpu - unified memory version

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                                   // initialize Magma
  magma_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  double  gpu_time;
  float   *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
  float *r;                                         // r- mxm matrix
  float *c;                                         // c- mxm matrix
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = { 0,0,0,1 };                         // seed
  const float alpha = 1.0;                                 // alpha=1
  const float beta = 0.0;                             // beta=0
  ldwork = m * magma_get_sgetri_nb( m ); // optimal block size
// allocate matrices
  cudaMallocManaged(&a,mm*sizeof(float)); // unified mem.for a
  cudaMallocManaged(&r,mm*sizeof(float)); // unified mem.for b
  cudaMallocManaged(&c,mm*sizeof(float)); // unified mem.for c
  cudaMallocManaged(&dwork,ldwork*sizeof(float));//m.for dwork
  cudaMallocManaged(&piv,m*sizeof(int));// unified mem.for piv
// generate random matrix a
  lapackf77_slarnv(&ione,ISEED,&mm,a);         // randomize a
  magmablas_slacpy(MagmaFull,m,m,a,m,r,m,queue);        //a->r
// find the inverse matrix: a*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);
// a -mxm  matrix; a is overwritten by the inverse
  gpu_time = magma_sync_wtime(NULL);

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

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  magma_sgemm(MagmaNoTrans,MagmaNoTrans,m,m,m,alpha,a,m,
                  r,m,beta,c,m,queue);    // multiply a^-1*a
  printf("magma_sgetrf_gpu + magma_sgetri_gpu time: %7.5f sec.\
                                             \n",gpu_time);
  magma_sgetmatrix( m, m, c, m, a, m, queue);      // copy c->a
  printf("upper left corner of a^-1*a:\n");
  magma_sprint( 4, 4, a, m );                 // part of a^-1*a
  magma_free(piv);                               // free memory
  magma_free(a);                                // free  memory
  magma_free(r);                                // free  memory
  magma_free(c);                                // free  memory
  magma_queue_destroy(queue);                  // destroy queue
  magma_finalize();                           // finalize Magma
  return 0;
}
//magma_sgetrf_gpu + magma_sgetri_gpu time: 0.53595 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
//];
```

4.3.21 magma_dgetri_gpu - inverse matrix in double precision, GPU interface

This function computes in double precision the inverse A^{-1} of an $m \times m$ 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_v2.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                              // initialize Magma
  magma_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  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_int_t err;
  const double alpha = 1.0;                             // alpha=1
  const double beta = 0.0;                               // beta=0
  ldwork = m * magma_get_dgetri_nb( m ); // optimal block 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 r
  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);          // randomize a
  magma_dsetmatrix( m, m, a,m, d_a,m,queue);  // copy a -> d_a
  magmablas_dlacpy(MagmaFull,m,m,d_a,m,d_r,m,queue);//d_a->d_r
// find the inverse matrix: d_a*X=I using the LU factorization
// with partial pivoting and row interchanges computed by
// magma_dgetrf_gpu; row i is interchanged with row piv(i);
// d_a -mxm  matrix; d_a is overwritten by the inverse
  gpu_time = magma_sync_wtime(NULL);

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

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  magma_dgemm(MagmaNoTrans,MagmaNoTrans,m,m,m,alpha,d_a,m,
               d_r,m,beta,d_c,m,queue);   // multiply a^-1*a
  printf("magma_dgetrf_gpu + magma_dgetri_gpu time: %7.5f sec.\
                                                   \n",gpu_time);
  magma_dgetmatrix( m, m, d_c, m, a, m, queue); // 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
  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_queue_destroy(queue);                      // destroy queue
  magma_finalize();                               // finalize Magma
  return 0;
}
//magma_dgetrf_gpu + magma_dgetri_gpu time: 4.79694 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
//];
```

4.3.22 magma_dgetri_gpu - unified memory version

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.h"
#include "magma_lapack.h"
int main( int argc, char** argv ){
  magma_init();                                  // initialize Magma
  magma_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  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
  double *r;                                       // r- mxm matrix
  double *c;                                       // c- mxm matrix
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = { 0,0,0,1 };                       // seed
  const double alpha = 1.0;                              // alpha=1
  const double beta = 0.0;                                // beta=0
  ldwork = m * magma_get_dgetri_nb( m ); // optimal block size
// allocate matrices
  cudaMallocManaged(&a,mm*sizeof(double));// unified mem.for a
  cudaMallocManaged(&r,mm*sizeof(double));// unified mem.for r
  cudaMallocManaged(&c,mm*sizeof(double));// unified mem.for c
  cudaMallocManaged(&dwork,ldwork*sizeof(double));//mem. dwork
  cudaMallocManaged(&piv,m*sizeof(int));// unified mem.for piv
// generate random matrix a
  lapackf77_dlarnv(&ione,ISEED,&mm,a);          // randomize a
  magmablas_dlacpy(MagmaFull,m,m,a,m,r,m,queue);        //a->r
// find the inverse matrix: a*X=I using the LU factorization
// with partial pivoting and row interchanges computed by
// magma_dgetrf_gpu; row i is interchanged with row piv(i);
// a -mxm  matrix; a is overwritten by the inverse
  gpu_time = magma_sync_wtime(NULL);

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

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  magma_dgemm(MagmaNoTrans,MagmaNoTrans,m,m,m,alpha,a,m,
                 r,m,beta,c,m,queue);     // multiply a^-1*a
  printf("magma_dgetrf_gpu + magma_dgetri_gpu time: %7.5f sec.\
                                             \n",gpu_time);
  magma_dgetmatrix( m, m, c, m, a, m, queue);     // copy c->a
  printf("upper left corner of a^-1*a:\n");
  magma_dprint( 4, 4, a, m );                   // part of a^-1*a
  magma_free(piv);                               // free memory
  magma_free(a);                                // free  memory
  magma_free(r);                                // free  memory
  magma_free(c);                                // free  memory
  magma_queue_destroy(queue);                  // destroy queue
  magma_finalize();                            // finalize Magma
  return 0;
}
//magma_dgetrf_gpu + magma_dgetri_gpu time: 4.77694 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

$$A = \begin{cases} U^T\,U & \text{in } \texttt{MagmaUpper} \text{ case,} \\ L\,L^T & \text{in } \texttt{MagmaLower} \text{ case} \end{cases}$$

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_v2.h"
#include "magma_lapack.h"
#include "magma_sutil.cpp"
int main( int argc, char** argv ){
  magma_init();                              // initialize Magma
  double   gpu_time;
  magma_int_t info;
  magma_int_t m = 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_int_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);           // randomize a
// b - mxn matrix of ones
  lapackf77_slaset(MagmaFullStr,&m,&n,&alpha,&alpha,b,&m);
// symmetrize a and increase  diagonal
  magma_smake_hpd( m, a, 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);
// 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
  gpu_time = magma_sync_wtime(NULL);

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

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  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 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: 0.41469 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_sposv - unified memory version

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.h"
#include "magma_lapack.h"
#include "magma_dutil.cpp"
int main( int argc, char** argv ){
  magma_init();                                // initialize Magma
  double   gpu_time;
  magma_int_t info;
  magma_int_t m = 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
  float *b;                                         // b- mxn matrix
  float *c;                                         // c- mxn matrix
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = { 0,0,0,1 };                       // seed
  const float alpha = 1.0;                               // alpha=1
  const float beta = 0.0;                                 // beta=0
// allocate matrices
  cudaMallocManaged(&a,mm*sizeof(float)); // unif.memory for a
  cudaMallocManaged(&b,mn*sizeof(float)); // unif.memory for b
  cudaMallocManaged(&c,mn*sizeof(float)); // unif.memory for c
// generate random matrices a, b;
  lapackf77_slarnv(&ione,ISEED,&mm,a);            // randomize a
  lapackf77_slaset(MagmaFullStr,&m,&n,&alpha,&alpha,b,&m);
// symmetrize a and increase  diagonal
  magma_smake_hpd( m, a, 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,&n,&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
  gpu_time = magma_sync_wtime(NULL);

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

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  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 Magma solution
  magma_free(a);                                  // free memory
  magma_free(b);                                  // free memory
  magma_free(c);                                  // free 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: 0.44253 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_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

$$A = \begin{cases} U^T\,U & \text{in } \texttt{MagmaUpper} \text{ case,} \\ L\,L^T & \text{in } \texttt{MagmaLower} \text{ case} \end{cases}$$

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_v2.h"
#include "magma_lapack.h"
#include "magma_dutil.cpp"
int main( int argc, char** argv ){
  magma_init();                              // initialize Magma
  double   gpu_time;
  magma_int_t info;
  magma_int_t m = 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_int_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 random matrices a, b;
```

```
  lapackf77_dlarnv(&ione,ISEED,&mm,a);          // randomize a
  lapackf77_dlaset(MagmaFullStr,&m,&n,&alpha,&alpha,b,&m);
// symmetrize a and increase  diagonal
  magma_dmake_hpd( m, a, 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,&n,&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
  gpu_time = magma_sync_wtime(NULL);

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

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  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 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: 1.39989 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 - unified memory version

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.h"
#include "magma_lapack.h"
#include "magma_dutil.cpp"
int main( int argc, char** argv ){
  magma_init();                              // initialize Magma
  double   gpu_time;
  magma_int_t info;
  magma_int_t m = 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
  double *b;                                    // b- mxn matrix
  double *c;                                    // c- mxn matrix
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = { 0,0,0,1 };                   // seed
  const double alpha = 1.0;                          // alpha=1
  const double beta = 0.0;                            // beta=0
// allocate matrices
  cudaMallocManaged(&a,mm*sizeof(double));// unif.memory for a
  cudaMallocManaged(&b,mn*sizeof(double));// unif.memory for b
  cudaMallocManaged(&c,mn*sizeof(double));// unif.memory for c
// generate random matrices a, b;
  lapackf77_dlarnv(&ione,ISEED,&mm,a);           // randomize a
  lapackf77_dlaset(MagmaFullStr,&m,&n,&alpha,&alpha,b,&m);
// symmetrize a and increase  diagonal
  magma_dmake_hpd( m, a, 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,&n,&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
  gpu_time = magma_sync_wtime(NULL);

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

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  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 Magma solution
  magma_free(a);                                // free memory
  magma_free(b);                                // free memory
  magma_free(c);                                // free 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:  1.39497 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
//];
```

4.4.5 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

$$A = \begin{cases} U^T\,U & \text{in } \texttt{MagmaUpper} \text{ case,} \\ L\,L^T & \text{in } \texttt{MagmaLower} \text{ case} \end{cases}$$

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/sposv_gpu.cpp for more details.

```
#include <stdio.h>
#include <cuda.h>
#include "magma_v2.h"
#include "magma_lapack.h"
#include "magma_sutil.cpp"
int main( int argc, char** argv ){
  magma_init();                                 // initialize Magma
  magma_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  double   gpu_time;
  magma_int_t info;
  magma_int_t m = 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_int_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
  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);           // randomize a
// b - mxn matrix of ones
  lapackf77_slaset(MagmaFullStr,&m,&n,&alpha,&alpha,b,&m);
// symmetrize a and increase  diagonal
  magma_smake_hpd( m, a, 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,queue);  // copy a -> d_a
  magma_ssetmatrix( m, n, c,m, d_c,m,queue);  // 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
  gpu_time = magma_sync_wtime(NULL);

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

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("magma_sposv_gpu time: %7.5f sec.\n",gpu_time);
  magma_sgetmatrix( m, n, d_c, m, c, m, queue );
  printf("upper left corner of the Magma solution:\n");
  magma_sprint( 4, 4, c, m );        // part of 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_queue_destroy(queue);                  // destroy queue
  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: 0.05821 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
//];
```

4.4.6 magma_sposv_gpu - unified memory version

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.h"
#include "magma_lapack.h"
#include "magma_dutil.cpp"
int main( int argc, char** argv ){
  magma_init();                               // initialize Magma
  magma_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  double   gpu_time;
  magma_int_t info;
  magma_int_t m = 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
  float *b;                                       // b- mxn matrix
  float *c;                                       // c- mxn matrix
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = { 0,0,0,1 };                        // seed
  const float alpha = 1.0;                                // alpha=1
  const float beta = 0.0;                                  // beta=0
// allocate matrices
  cudaMallocManaged(&a,mm*sizeof(float)); // unif.memory for a
  cudaMallocManaged(&b,mn*sizeof(float)); // unif.memory for b
  cudaMallocManaged(&c,mn*sizeof(float)); // unif.memory for c
// generate  matrices
  lapackf77_slarnv(&ione,ISEED,&mm,a);            // randomize a
// b - mxn matrix of ones
  lapackf77_slaset(MagmaFullStr,&m,&n,&alpha,&alpha,b,&m);
// symmetrize a and increase  diagonal
  magma_smake_hpd( m, a, 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);
// 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
```

```
  gpu_time = magma_sync_wtime(NULL);

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

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("magma_sposv_gpu time: %7.5f sec.\n",gpu_time);
  printf("upper left corner of the solution:\n");
  magma_sprint( 4, 4, c, m );         // part of Magma solution
  magma_free(a);                                // free  memory
  magma_free(b);                                // free  memory
  magma_free(c);                                // free  memory
  magma_queue_destroy(queue);                  // destroy queue
  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: 0.09663 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
//];
```

4.4.7 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

$$A = \begin{cases} U^T\,U & \text{in MagmaUpper case,} \\ L\,L^T & \text{in MagmaLower case} \end{cases}$$

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_v2.h"
#include "magma_lapack.h"
#include "magma_dutil.cpp"
int main( int argc, char** argv ){
  magma_init();                                  // initialize Magma
  magma_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  double   gpu_time;
  magma_int_t info;
  magma_int_t m = 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_int_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
  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);           // randomize a
// b - mxn matrix of ones
  lapackf77_dlaset(MagmaFullStr,&m,&n,&alpha,&alpha,b,&m);
// symmetrize a and increase  diagonal
  magma_dmake_hpd( m, a, 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,queue ); // copy a -> d_a
  magma_dsetmatrix( m, n, c,m, d_c,m,queue ); // 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
  gpu_time = magma_sync_wtime(NULL);

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

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("magma_dposv_gpu time: %7.5f sec.\n",gpu_time);
  magma_dgetmatrix( m, n, d_c, m, c, m, queue );
  printf("upper left corner of the solution:\n");
  magma_dprint( 4, 4, c, m );        // part of 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_queue_destroy(queue);                      // destroy queue
  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: 0.93042 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
//];
```

4.4.8 magma_dposv_gpu - unified memory version

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.h"
#include "magma_lapack.h"
#include "magma_dutil.cpp"
int main( int argc, char** argv ){
  magma_init();                                 // initialize Magma
  magma_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  double   gpu_time;
  magma_int_t info;
  magma_int_t m = 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
  double *b;                                       // b- mxn matrix
  double *c;                                       // c- mxn matrix
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = { 0,0,0,1 };                      // seed
  const double alpha = 1.0;                             // alpha=1
  const double beta = 0.0;                               // beta=0
// allocate matrices
  cudaMallocManaged(&a,mm*sizeof(double));// unif.memory for a
  cudaMallocManaged(&b,mn*sizeof(double));// unif.memory for b
  cudaMallocManaged(&c,mn*sizeof(double));// unif.memory for c
// generate  matrices
  lapackf77_dlarnv(&ione,ISEED,&mm,a);           // randomize a
// b - mxn matrix of ones
  lapackf77_dlaset(MagmaFullStr,&m,&n,&alpha,&alpha,b,&m);
// symmetrize a and increase  diagonal
  magma_dmake_hpd( m, a, 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);
// 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
  gpu_time = magma_sync_wtime(NULL);

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

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("magma_dposv_gpu time: %7.5f sec.\n",gpu_time);
  printf("upper left corner of the solution:\n");
  magma_dprint( 4, 4, c, m );        // part of Magma solution
  magma_free(a);                               // free  memory
  magma_free(b);                               // free  memory
  magma_free(c);                               // free  memory
  magma_queue_destroy(queue);                 // destroy queue
  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: 0.94532 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`, `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:

$$A = \begin{cases} U^T\,U & \text{in } \texttt{MagmaUpper} \text{ case,} \\ L\,L^T & \text{in } \texttt{MagmaLower} \text{ case,} \end{cases}$$

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_v2.h"
#include "magma_lapack.h"
#include "magma_sutil.cpp"
int main( int argc, char** argv ){
  magma_init();                                // initialize Magma
  double   gpu_time;
  magma_int_t  info;
  magma_int_t m = 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_int_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);            // randomize a
// b - mxn matrix of ones
  lapackf77_slaset(MagmaFullStr,&m,&n,&alpha,&alpha,b,&m);
// symmetrize a and increase  diagonal
  magma_smake_hpd( m, a, m );
  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
  gpu_time = magma_sync_wtime(NULL);

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

  gpu_time = magma_sync_wtime(NULL)-gpu_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: 0.49789 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
//];
```

4.4.10 magma_spotrf, lapackf77_spotrs - unified memory version

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.h"
#include "magma_lapack.h"
#include "magma_dutil.cpp"
```

```
int main( int argc, char** argv ){
  magma_init();                                  // initialize Magma
  double   gpu_time;
  magma_int_t info;
  magma_int_t m = 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
  float *b;                                         // b- mxn matrix
  float *c;                                         // c- mxn matrix
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = { 0,0,0,1 };                        // seed
//  magma_int_t err;
  const float alpha = 1.0;                                // alpha=1
  const float beta = 0.0;                                  // beta=0
// allocate matrices
  cudaMallocManaged(&a,mm*sizeof(float)); // unif.memory for a
  cudaMallocManaged(&b,mn*sizeof(float)); // unif.memory for b
  cudaMallocManaged(&c,mn*sizeof(float)); // unif.memory for c
// generate random matrices a, b;
  lapackf77_slarnv(&ione,ISEED,&mm,a);          // randomize a
// b - mxn matrix of ones
  lapackf77_slaset(MagmaFullStr,&m,&n,&alpha,&alpha,b,&m);
// symmetrize a and increase  diagonal
  magma_smake_hpd( m, a, m );
  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
  gpu_time = magma_sync_wtime(NULL);

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

  gpu_time = magma_sync_wtime(NULL)-gpu_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.
  magma_free(a);                                  // free  memory
  magma_free(b);                                  // free  memory
  magma_free(c);                                  // free  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: 0.48314 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
//];
```

4.4.11 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:

$$A = \begin{cases} U^T\,U & \text{in MagmaUpper case,} \\ L\,L^T & \text{in MagmaLower case,} \end{cases}$$

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_v2.h"
#include "magma_lapack.h"
#include "magma_dutil.cpp"
int main( int argc, char** argv ){
  magma_init();                               // initialize Magma
  double   gpu_time;
  magma_int_t info;
  magma_int_t m = 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_int_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 random matrices a, b;
  lapackf77_dlarnv(&ione,ISEED,&mm,a);          // randomize a
// b - mxn matrix of ones
  lapackf77_dlaset(MagmaFullStr,&m,&n,&alpha,&alpha,b,&m);
// symmetrize a and increase  diagonal
  magma_dmake_hpd( m, a, m );
  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
  gpu_time = magma_sync_wtime(NULL);

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

  gpu_time = magma_sync_wtime(NULL)-gpu_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: 1.40168  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
//];
```

4.4.12 magma_dpotrf, lapackf77_dpotrs - unified memory version

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.h"
#include "magma_lapack.h"
#include "magma_dutil.cpp"
int main( int argc, char** argv ){
  magma_init();                                 // initialize Magma
  double   gpu_time;
  magma_int_t info;
  magma_int_t m = 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
  double *b;                                       // b- mxn matrix
  double *c;                                       // c- mxn matrix
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = { 0,0,0,1 };                      // seed
//  magma_int_t err;
  const double alpha = 1.0;                             // alpha=1
  const double beta = 0.0;                               // beta=0
// allocate matrices
  cudaMallocManaged(&a,mm*sizeof(double));// unif.memory for a
  cudaMallocManaged(&b,mn*sizeof(double));// unif.memory for b
  cudaMallocManaged(&c,mn*sizeof(double));// unif.memory for c
// generate random matrices a, b;
  lapackf77_dlarnv(&ione,ISEED,&mm,a);           // randomize a
// b - mxn matrix of ones
  lapackf77_dlaset(MagmaFullStr,&m,&n,&alpha,&alpha,b,&m);
// symmetrize a and increase  diagonal
  magma_dmake_hpd( m, a, m );
  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
  gpu_time = magma_sync_wtime(NULL);
```

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

   gpu_time = magma_sync_wtime(NULL)-gpu_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.
   magma_free(a);                                 // free  memory
   magma_free(b);                                 // free  memory
   magma_free(c);                                 // free  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: 1.30345  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
//];
```

4.4.13 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:

$$A = \begin{cases} U^T\,U & \text{in MagmaUpper case,} \\ L\,L^T & \text{in MagmaLower case,} \end{cases}$$

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_v2.h"
#include "magma_lapack.h"
#include "magma_sutil.cpp"
int main( int argc, char** argv ){
  magma_init();                              // initialize Magma
  magma_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  double   gpu_time;
  magma_int_t info;
  magma_int_t m = 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_int_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
  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);           // randomize a
  lapackf77_slaset(MagmaFullStr,&m,&n,&alpha,&alpha,b,&m);
                                       // b - mxn matrix of ones
// symmetrize a and increase  diagonal
  magma_smake_hpd( m, a, m );
  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,queue);  // copy a -> d_a
  magma_ssetmatrix( m, n, c,m, d_c,m,queue);  // 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
  gpu_time = magma_sync_wtime(NULL);
  magma_spotrf_gpu(MagmaLower, m, d_a, m, &info);
  magma_spotrs_gpu(MagmaLower,m,n,d_a,m,d_c,m,&info);

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("magma_spotrf_gpu+magma_spotrs_gpu time: %7.5f sec.\n",
                                                  gpu_time);
  magma_sgetmatrix( m, n, d_c, m, c,m,queue); // copy d_c -> c
  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_queue_destroy(queue);                  // destroy queue
  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: 0.05582 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
//];
```

4.4.14 magma_spotrf_gpu, magma_spotrs_gpu - unified memory version

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.h"
#include "magma_lapack.h"
#include "magma_dutil.cpp"
int main( int argc, char** argv ){
  magma_init();                             // initialize Magma
  magma_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  double   gpu_time;
  magma_int_t info;
  magma_int_t m = 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
  float *b;                                         // b- mxn matrix
  float *c;                                         // c- mxn matrix
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = {0,0,0,1};                           // seed
  const float alpha = 1.0;                                 // alpha=1
  const float beta = 0.0;                                   // beta=0
// allocate matrices
  cudaMallocManaged(&a,mm*sizeof(float)); // unif.memory for a
  cudaMallocManaged(&b,mn*sizeof(float)); // unif.memory for b
  cudaMallocManaged(&c,mn*sizeof(float)); // unif.memory for c
// generate  matrices
  lapackf77_slarnv(&ione,ISEED,&mm,a);          // randomize a
  lapackf77_slaset(MagmaFullStr,&m,&n,&alpha,&alpha,b,&m);
                                         // b - mxn matrix of ones
// symmetrize a and increase  diagonal
  magma_smake_hpd( m, a, m );
  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
  gpu_time = magma_sync_wtime(NULL);

  magma_spotrf_gpu(MagmaLower, m, a, m, &info);
  magma_spotrs_gpu(MagmaLower,m,n,a,m,c,m,&info);

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("magma_spotrf_gpu+magma_spotrs_gpu time: %7.5f sec.\n",
                                                    gpu_time);
  printf("upper left corner of the solution:\n");
  magma_sprint( 4, 4, c, m );    // part of the Magma solution
  magma_free(a);                                 // free  memory
  magma_free(b);                                 // free  memory
  magma_free(c);                                 // free  memory
  magma_queue_destroy(queue);                   // destroy queue
  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: 0.09600 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
//];
```

4.4.15 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:

$$A = \begin{cases} U^T\,U & \text{in } \texttt{MagmaUpper} \text{ case,} \\ L\,L^T & \text{in } \texttt{MagmaLower} \text{ case,} \end{cases}$$

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/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_v2.h"
#include "magma_lapack.h"
#include "magma_dutil.cpp"
int main( int argc, char** argv ){
  magma_init();                                  // initialize Magma
  magma_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  double   gpu_time;
  magma_int_t info;
  magma_int_t m = 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_int_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);           // randomize a
  lapackf77_dlaset(MagmaFullStr,&m,&n,&alpha,&alpha,b,&m);
                                     // b - mxn matrix of ones
// symmetrize a and increase diagonal
  magma_dmake_hpd( m, a, m );
  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,queue);  // copy a -> d_a
  magma_dsetmatrix( m, n, c,m, d_c,m,queue);  // 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
  gpu_time = magma_sync_wtime(NULL);

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

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("magma_dpotrf_gpu+magma_dpotrs_gpu time:%7.5f sec.\n",
                                                      gpu_time);
  magma_dgetmatrix( m, n, d_c, m, c,m,queue); // 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_queue_destroy(queue);                  // destroy queue
  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: 0.93016 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
//];
```

4.4.16 magma_dpotrf_gpu, magma_dpotrs_gpu - unified memory version

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.h"
#include "magma_lapack.h"
#include "magma_dutil.cpp"
int main( int argc, char** argv ){
  magma_init();                               // initialize Magma
  magma_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  double   gpu_time;
  magma_int_t info;
  magma_int_t m = 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
  double *b;                                     // b- mxn matrix
  double *c;                                     // c- mxn matrix
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = {0,0,0,1};                       // seed
  const double alpha = 1.0;                            // alpha=1
  const double beta = 0.0;                              // beta=0
// allocate matrices
  cudaMallocManaged(&a,mm*sizeof(double));// unif.memory for a
  cudaMallocManaged(&b,mn*sizeof(double));// unif.memory for b
  cudaMallocManaged(&c,mn*sizeof(double));// unif.memory for c
// generate  matrices
  lapackf77_dlarnv(&ione,ISEED,&mm,a);           // randomize a
  lapackf77_dlaset(MagmaFullStr,&m,&n,&alpha,&alpha,b,&m);
                                         // b - mxn matrix of ones
// symmetrize a and increase  diagonal
  magma_dmake_hpd( m, a, m );
  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
  gpu_time = magma_sync_wtime(NULL);

  magma_dpotrf_gpu(MagmaLower, m, a, m, &info);
  magma_dpotrs_gpu(MagmaLower,m,n,a,m,c,m,&info);

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("magma_dpotrf_gpu+magma_dpotrs_gpu time: %7.5f sec.\n",
                                                   gpu_time);
  printf("upper left corner of the solution:\n");
  magma_dprint( 4, 4, c, m );    // part of the Magma solution

  magma_free(a);                                   // free  memory
  magma_free(b);                                   // free  memory
  magma_free(c);                                   // free  memory
  magma_queue_destroy(queue);                     // destroy queue
  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: 0.95875 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
//];
```

4.4.17 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:

$$A = \begin{cases} U^T\, U & \text{in } \texttt{MagmaUpper} \text{ case,} \\ L\, L^T & \text{in } \texttt{MagmaLower} \text{ case,} \end{cases}$$

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_v2.h"
#include "magma_lapack.h"
#include "magma_sutil.cpp"
int main( int argc, char** argv) {
  magma_init();                                // initialize Magma
  int num_gpus = 1;
  magma_setdevice(0);
  magma_queue_t queues[num_gpus];
  for( int dev = 0; dev < num_gpus; ++dev ) {
    magma_queue_create( dev, &queues[dev] );
  }
  double   cpu_time,gpu_time;
  magma_int_t err;
  magma_int_t m = 8192;                       // a,r - mxm matrices
  magma_int_t nrhs =100;                   // b,c - mxnrhs matrices
  magma_int_t mm=m*m;                                 // size of a,r
  magma_int_t mnrhs=m*nrhs;                           // size of b,c
  float *a, *r;              // a,r - mxn matrices on the host
  float *b, *c;            // b,c - mxnrhs matrices on the host
  magmaFloat_ptr d_la[num_gpus];
  float alpha=1.0, beta=0.0;
  magma_int_t  mb, nb;
  magma_int_t lda=m, ldda, n_local;
  magma_int_t i, info;
  magma_int_t ione = 1 ;
  magma_int_t ISEED[4] = {0,0,0,1};
  nb = magma_get_spotrf_nb(m);// optimal block size for spotrf
  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_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 local matrix on the 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 );     // randomize a
  lapackf77_slaset(MagmaFullStr,&m,&nrhs,&alpha,&alpha,b,&m);
                                 // b - mxnrhs matrix of ones
// Symmetrize a and increase diagonal
  magma_smake_hpd( m, a, m );
// copy a -> r
  lapackf77_slacpy( MagmaFullStr,&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 c=a*b
// MAGMA
// distribute the matrix a to num_gpus devices
// going through each block-row
  ldda = (1+m/(nb*num_gpus))*nb;
  magma_ssetmatrix_1D_row_bcyclic( num_gpus, m, m, nb, r, lda,
                                        d_la, ldda, queues );
  magma_setdevice(0);
  gpu_time = magma_sync_wtime(NULL);
// 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);

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("magma_spotrf_mgpu time: %7.5f sec.\n", gpu_time);
// gather the resulting matrix from num_gpus devices to r
  magma_sgetmatrix_1D_row_bcyclic( num_gpus, m, m, nb, d_la,
                                    ldda, r, lda, queues );
  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 solution \n\
  from spotrf_mgpu+spotrs:\n");
  magma_sprint( 4, 4, c, m);          // Magma/Lapack solution
```

```
// LAPACK version of spotrf for time comparison
   cpu_time=magma_wtime();
   lapackf77_spotrf("L", &m, a, &lda, &info);
   cpu_time=magma_wtime()-cpu_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
   }
    for( int dev = 0; dev < num_gpus; ++dev ) {
         magma_queue_destroy( queues[dev] );
     }
   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: 0.05060 sec.
//upper left corner of the 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: 0.70702 sec.
```

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

$$A = \begin{cases} U^T\,U & \text{in MagmaUpper case,} \\ L\,L^T & \text{in MagmaLower case,} \end{cases}$$

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_v2.h"
#include "magma_lapack.h"
#include "magma_dutil.cpp"
int main( int argc, char** argv) {
  magma_init();                              // initialize Magma
  int num_gpus = 1;
  magma_setdevice(0);
  magma_queue_t queues[num_gpus];
  for( int dev = 0; dev < num_gpus; ++dev ) {
    magma_queue_create( dev, &queues[dev] );
  }
  double   cpu_time,gpu_time;
  magma_int_t err;
  magma_int_t m = 8192;                    // a,r - m*m matrices
  magma_int_t nrhs =100;               // b,c - mxnrhs matrices
  magma_int_t mm=m*m;                                // size of a,r
  magma_int_t mnrhs=m*nrhs;                          // size of b,c
  double *a, *r;               // a,r - mxn matrices on the host
  double *b, *c;           // b,c - mxnrhs matrices on the host
  magmaDouble_ptr d_la[num_gpus];
  double alpha=1.0, beta=0.0;
  magma_int_t  mb, nb;
  magma_int_t lda=m, ldda, n_local;
  magma_int_t i, info;
  magma_int_t ione = 1 ;
  magma_int_t ISEED[4] = {0,0,0,1};
  nb = magma_get_dpotrf_nb(m);// optimal block size 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 );     // randomize a
  lapackf77_dlaset(MagmaFullStr,&m,&nrhs,&alpha,&alpha,b,&m);
                                // b - mxnrhs matrix of ones
// Symmetrize a and increase  diagonal
  magma_dmake_hpd( m, a, m );
// copy a -> r
  lapackf77_dlacpy( MagmaFullStr,&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;
  magma_dsetmatrix_1D_row_bcyclic( num_gpus, m, m, nb, r, lda,
                                        d_la, ldda, queues );
  magma_setdevice(0);
  gpu_time = magma_sync_wtime(NULL);
// 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);

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("magma_dpotrf_mgpu time: %7.5f sec.\n", gpu_time);
// gather the resulting matrix from num_gpus devices to r
  magma_dgetmatrix_1D_row_bcyclic( num_gpus, m, m, nb, d_la,
                                      ldda, r, lda, queues );
  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
  cpu_time=magma_wtime();
  lapackf77_dpotrf("L", &m, a, &lda, &info);
  cpu_time=magma_wtime()-cpu_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
  }
   for( int dev = 0; dev < num_gpus; ++dev ) {
        magma_queue_destroy( queues[dev] );
    }
  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: 0.79751 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: 1.72130 sec.
```

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

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

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

It uses the Cholesky decomposition:

$$A = \begin{cases} U^T\,U & \text{in MagmaUpper case,} \\ L\,L^T & \text{in MagmaLower case,} \end{cases}$$

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_v2.h"
#include "magma_lapack.h"
#include "magma_sutil.cpp"
int main( int argc, char** argv ){
  magma_init();                                  // initialize Magma
  double   gpu_time ;
  magma_int_t  info;
  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_int_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);           // randomize a
// symmetrize a and increase  diagonal
  magma_smake_hpd( m, a, m );
  lapackf77_slacpy(MagmaFullStr,&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
  gpu_time = magma_sync_wtime(NULL);

  magma_spotrf( MagmaLower, m, a, m, &info);
  magma_spotri( MagmaLower, m, a, m, &info);
                                            //a overwritten by a^-1
  gpu_time = magma_sync_wtime(NULL)-gpu_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: 0.58457 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
//];
```

4.4.20 magma_spotri - unified memory version

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.h"
#include "magma_lapack.h"
#include "magma_dutil.cpp"
int main( int argc, char** argv ){
  magma_init();                              // initialize Magma
  double   gpu_time ;
  magma_int_t  info;
  magma_int_t m = 8192;                        // a - mxm matrix
  magma_int_t mm=m*m;                         // size of a, r, c
  float *a;                                    // a- mxm matrix
  float *r;                                    // r- mxm matrix
  float *c;                                    // c- mxm matrix
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = {0,0,0,1};                     // seed
  const float alpha = 1.0;                           // alpha=1
  const float beta = 0.0;                             // beta=0
// allocate matrices
  cudaMallocManaged(&a,mm*sizeof(float)); // unified mem.for a
  cudaMallocManaged(&r,mm*sizeof(float)); // unified mem.for r
  cudaMallocManaged(&c,mm*sizeof(float)); // unified mem.for c
// generate random matrix a
  lapackf77_slarnv(&ione,ISEED,&mm,a);          // randomize a
// symmetrize a and increase  diagonal
  magma_smake_hpd( m, a, m );
  lapackf77_slacpy(MagmaFullStr,&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
  gpu_time = magma_sync_wtime(NULL);

  magma_spotrf( MagmaLower, m, a, m, &info);
  magma_spotri( MagmaLower, m, a, m, &info);
                                     // a overwritten by a^-1
  gpu_time = magma_sync_wtime(NULL)-gpu_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
  magma_free(a);                                 // free memory
  magma_free(r);                                 // free memory
  magma_free(c);                                 // free memory
  magma_finalize();                            // finalize Magma
  return 0;
}
//magma_spotrf + magma_spotri time: 0.57705 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
//];
```

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

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

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

It uses the Cholesky decomposition:

$$A = \begin{cases} U^T\,U & \text{in MagmaUpper case,} \\ L\,L^T & \text{in MagmaLower case,} \end{cases}$$

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_v2.h"
#include "magma_lapack.h"
#include "magma_dutil.cpp"
int main( int argc, char** argv ){
  magma_init();                                  // initialize Magma
  double   gpu_time ;
  magma_int_t  info;
  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_int_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);           // randomize a
// symmetrize a and increase  diagonal
  magma_dmake_hpd( m, a, m );
  lapackf77_dlacpy(MagmaFullStr,&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_dpotrf;  a is overwritten by the inverse
  gpu_time = magma_sync_wtime(NULL);

  magma_dpotrf( MagmaLower, m, a, m, &info);
  magma_dpotri( MagmaLower, m, a, m, &info);
                                     // a overwritten by a^-1
  gpu_time = magma_sync_wtime(NULL)-gpu_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: 3.06706 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
//];
```

4.4.22 magma_dpotri - unified memory version

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.h"
#include "magma_lapack.h"
#include "magma_dutil.cpp"
int main( int argc, char** argv ){
  magma_init();                               // initialize Magma
  double   gpu_time ;
  magma_int_t  info;
  magma_int_t m = 8192;                         // a - mxm matrix
  magma_int_t mm=m*m;                          // size of a, r, c
  double *a;                                     // a- mxm matrix
  double *r;                                     // r- mxm matrix
  double *c;                                     // c- mxm matrix
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = {0,0,0,1};                        // seed
  const double alpha = 1.0;                              // alpha=1
  const double beta = 0.0;                                // beta=0
// allocate matrices
  cudaMallocManaged(&a,mm*sizeof(double));// unified mem.for a
  cudaMallocManaged(&r,mm*sizeof(double));// unified mem.for r
  cudaMallocManaged(&c,mm*sizeof(double));// unified mem.for c
// generate random matrix a
  lapackf77_dlarnv(&ione,ISEED,&mm,a);          // randomize a
// symmetrize a and increase  diagonal
  magma_dmake_hpd( m, a, m );
  lapackf77_dlacpy(MagmaFullStr,&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_dpotrf;  a is overwritten by the inverse
  gpu_time = magma_sync_wtime(NULL);

  magma_dpotrf( MagmaLower, m, a, m, &info);
  magma_dpotri( MagmaLower, m, a, m, &info);
                                     // a overwritten by a^-1
  gpu_time = magma_sync_wtime(NULL)-gpu_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
  magma_free(a);                                   // free memory
  magma_free(r);                                   // free memory
  magma_free(c);                                   // free memory
  magma_finalize();                              // finalize Magma
  return 0;
}
//magma_dpotrf + magma_dpotri time: 3.06806 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
//];
```

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

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

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

It uses the Cholesky decomposition:

$$A = \begin{cases} U^T\,U & \text{in } \texttt{MagmaUpper} \text{ case,} \\ L\,L^T & \text{in } \texttt{MagmaLower} \text{ case,} \end{cases}$$

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_v2.h"
#include "magma_lapack.h"
#include "magma_dutil.cpp"
int main( int argc, char** argv ){
  magma_init();                                 // initialize Magma
  magma_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  double   gpu_time ;
  magma_int_t  info;
  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_int_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);           // randomize a
// symmetrize a and increase  diagonal
  magma_smake_hpd( m, a, m );
  magma_ssetmatrix( m, m, a,m,d_a,m,queue );  // copy a -> d_a
  magmablas_slacpy(MagmaFull,m,m,d_a,m,d_r,m,queue);//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 decompos.
// obtained by magma_spotrf_gpu;
// d_a is overwritten by the inverse
  gpu_time = magma_sync_wtime(NULL);

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

```
                                         //d_a overwritten by d_a^-1
  gpu_time = magma_sync_wtime(NULL)-gpu_time;
// compute d_a^-1*d_a
  magma_ssymm(MagmaLeft,MagmaLower,m,m,alpha,d_a,m,d_r,m,beta,
                                                  d_c,m,queue);
  printf("magma_spotrf_gpu + magma_spotri_gpu time: %7.5f sec.\
                                                     \n",gpu_time);
  magma_sgetmatrix( m, m, d_c, m, a, m,queue);  // 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_queue_destroy(queue);                          // destroy queue
  magma_finalize();                                  // finalize Magma
  return 0;
}
//magma_spotrf_gpu + magma_spotri_gpu time: 0.16664 sec.
//upper left corner of a^-1*a:
//[
//    1.0000    0.0000    0.        0.
//    0.0000    1.0000    0.0000    0.0000
//    0.0000    0.0000    1.0000   -0.0000
//    0.0000   -0.0000   -0.0000    1.0000
//];
```

4.4.24 magma_spotri_gpu - unified memory version

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.h"
#include "magma_lapack.h"
#include "magma_dutil.cpp"
int main( int argc, char** argv ){
  magma_init();                                     // initialize Magma
  magma_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  double   gpu_time ;
  magma_int_t  info;
  magma_int_t m = 8192;                               // a - mxm matrix
  magma_int_t mm=m*m;                               // size of a, r, c
  float *a;                                            // a- mxm matrix
  float *r;                                            // r- mxm matrix
  float *c;                                            // c- mxm matrix
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = {0,0,0,1};                            // seed
  const float alpha = 1.0;                                  // alpha=1
  const float beta = 0.0;                                    // beta=0
// allocate matrices
  cudaMallocManaged(&a,mm*sizeof(float)); // unified mem.for a
  cudaMallocManaged(&r,mm*sizeof(float)); // unified mem.for r
  cudaMallocManaged(&c,mm*sizeof(float)); // unified mem.for c
// generate random matrix a
  lapackf77_slarnv(&ione,ISEED,&mm,a);           // randomize a
// symmetrize a and increase  diagonal
  magma_smake_hpd( m, a, m );
  magmablas_slacpy(MagmaFull,m,m,a,m,r,m,queue);         //a->r
// find the inverse matrix (a)^-1: a*X=I for mxm symmetric
// positive definite matrix a using the Cholesky factoriza-
// tion obtained by magma_spotrf_gpu;
// a is overwritten by the inverse
  gpu_time = magma_sync_wtime(NULL);

  magma_spotrf_gpu( MagmaLower, m, a, m, &info);
  magma_spotri_gpu( MagmaLower, m, a, m, &info);
                                             //inv overwrites a

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  magma_ssymm(MagmaLeft,MagmaLower,m,m,alpha,a,m,r,m,beta,
                          c,m,queue);               // c=a^-1*a
  printf("magma_spotrf_gpu + magma_spotri_gpu time: %7.5f sec.\
                                                  \n",gpu_time);
  magma_sgetmatrix( m, m, c, m, a, m,queue);        // copy c->a
  printf("upper left corner of a^-1*a:\n");
  magma_sprint( 4, 4, a, m );                    // part of a^-1*a
  magma_free(a);                                   // free  memory
  magma_free(r);                                   // free  memory
  magma_free(c);                                   // free  memory
  magma_queue_destroy(queue);                     // destroy queue
  magma_finalize();                              // finalize Magma
  return 0;
}
//magma_spotrf_gpu + magma_spotri_gpu time: 0.15814 sec.
//upper left corner of a^-1*a:
//[
//   1.0000   0.0000   0.        0.
//   0.0000   1.0000   0.0000   0.0000
//   0.0000   0.0000   1.0000  -0.0000
//   0.0000  -0.0000  -0.0000   1.0000
//];
```

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

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

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

It uses the Cholesky decomposition:

$$A = \begin{cases} U^T\,U & \text{in } \texttt{MagmaUpper} \text{ case,} \\ L\,L^T & \text{in } \texttt{MagmaLower} \text{ case,} \end{cases}$$

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_v2.h"
#include "magma_lapack.h"
#include "magma_dutil.cpp"
int main( int argc, char** argv ){
  magma_init();                                  // initialize Magma
  magma_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  double   gpu_time ;
  magma_int_t  info;
  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_int_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);           // randomize a
// symmetrize a and increase  diagonal
  magma_dmake_hpd( m, a, m );
  magma_dsetmatrix( m, m, a, m,d_a,m,queue ); // copy a -> d_a

  magmablas_dlacpy(MagmaFull,m,m,d_a,m,d_r,m,queue);//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
  gpu_time = magma_sync_wtime(NULL);

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

```
                                              //inv overwrites d_a

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  magma_dsymm(MagmaLeft,MagmaLower,m,m,alpha,d_a,m,d_r,m,beta,
                        d_c,m,queue);      // d_c=d_a^-1*d_a
  printf("magma_dpotrf_gpu + magma_dpotri_gpu time: %7.5f sec.\
                                             \n",gpu_time);
  magma_dgetmatrix( m, m, d_c, m, a, m,queue);  // 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_queue_destroy(queue);                    // destroy queue
  magma_finalize();                             // finalize Magma
  return 0;
}
//magma_dpotrf_gpu + magma_dpotri_gpu time: 2.51915 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
//];
```

4.4.26 magma_dpotri_gpu - unified memory version

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.h"
#include "magma_lapack.h"
#include "magma_dutil.cpp"
int main( int argc, char** argv ){
  magma_init();                               // initialize Magma
  magma_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  double   gpu_time ;
  magma_int_t  info;
  magma_int_t m = 8192;                         // a - mxm matrix
  magma_int_t mm=m*m;                        // size of a, r, c
  double *a;                                    // a- mxm matrix
  double *r;                                    // r- mxm matrix
  double *c;                                    // c- mxm matrix
  magma_int_t ione = 1;
  magma_int_t ISEED[4] = {0,0,0,1};                      // seed
  const double alpha = 1.0;                           // alpha=1
  const double beta = 0.0;                              // beta=0
// allocate matrices
  cudaMallocManaged(&a,mm*sizeof(double));// unified mem.for a
  cudaMallocManaged(&r,mm*sizeof(double));// unified mem.for r
  cudaMallocManaged(&c,mm*sizeof(double));// unified mem.for c
// generate random matrix a
  lapackf77_dlarnv(&ione,ISEED,&mm,a);           // randomize a
// symmetrize a and increase  diagonal
  magma_dmake_hpd( m, a, m );
  magmablas_dlacpy(MagmaFull,m,m,a,m,r,m,queue);         //a->r
// find the inverse matrix (a)^-1: a*X=I for mxm symmetric
// positive definite matrix a using the Cholesky factoriza-
// tion obtained by magma_dpotrf_gpu;
// a is overwritten by the inverse
  gpu_time = magma_sync_wtime(NULL);

  magma_dpotrf_gpu( MagmaLower, m, a, m, &info);
  magma_dpotri_gpu( MagmaLower, m, a, m, &info);
                                              //inv overwrites a
  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  magma_dsymm(MagmaLeft,MagmaLower,m,m,alpha,a,m,r,m,beta,
                      c,m,queue);                   // c=a^-1*a
  printf("magma_dpotrf_gpu + magma_dpotri_gpu time: %7.5f sec.\
                                                 \n",gpu_time);
  magma_dgetmatrix( m, m, c, m, a, m,queue);       // copy c->a
  printf("upper left corner of a^-1*a:\n");
  magma_dprint( 4, 4, a, m );                    // part of a^-1*a
  magma_free(a);                                   // free  memory
  magma_free(r);                                   // free  memory
  magma_free(c);                                   // free  memory
  magma_queue_destroy(queue);                     // destroy queue
  magma_finalize();                              // finalize Magma
  return 0;
}
//magma_dpotrf_gpu + magma_dpotri_gpu time: 2.53001 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. See magma-X.Y.Z/src/sgels_gpu.cpp for more details.

```
#include <stdio.h>
#include <cuda.h>
#include "magma_v2.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_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  double gpu_time, cpu_time;
  magma_int_t m = 8192, n = 8192;              // a - mxn matrix
  magma_int_t nrhs = 100;      // b - nxnrhs, c - mxnrhs matrix
  float *a;                         // a - mxn matrix on the host
  float *b, *c;   // b - nxnrhs, c - mxnrhs  matrix on the host
  float *d_a, *d_c;    // d_a - mxn matrix, d_c - mxnrhs 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  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,n); //optim.block size 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 );    // randomize  a
  lapackf77_slaset(MagmaFullStr,&n,&nrhs,&alpha,&alpha,b,&m);
                                // b - mxnrhs 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,queue);// copy a -> d_a
  magma_ssetmatrix( m, nrhs, c,m,d_c,lddb,queue); //  c -> d_c
  gpu_time = magma_sync_wtime(NULL);
// 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);

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("MAGMA time: %7.3f sec.\n",gpu_time);   // Magma time
 // Get the solution in b
  magma_sgetmatrix( n, nrhs, d_c, lddb, b,n,queue);// d_c -> b
  printf("upper left corner of of the magma_sgels sol.:\n");
  magma_sprint( 4, 4, b, n ); // part of the Magma QR solution
// LAPACK  version of sgels
  cpu_time=magma_wtime();
  lapackf77_sgels( MagmaNoTransStr, &m, &n, &nrhs,
                   a, &m, c, &m, hwork, &lhwork, &info);
  cpu_time=magma_wtime()-cpu_time;
  printf("LAPACK time: %7.3f sec.\n",cpu_time); // Lapack time
  printf("upper left corner of the lapackf77_sgels sol.:\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_queue_destroy(queue);                       // destroy queue
  magma_finalize( );                               // finalize Magma
  return EXIT_SUCCESS;
}
//MAGMA time:   0.358 sec.
//upper left corner of of the magma_sgels solution:
//[
//   0.9811   0.9811   0.9811   0.9811
//   1.0186   1.0186   1.0186   1.0186
//   1.0216   1.0216   1.0216   1.0216
//   0.9952   0.9952   0.9952   0.9952
//];
//LAPACK time:  11.352 sec.
//upper left corner of the lapackf77_sgels solution:
//[
//   0.9963   0.9963   0.9963   0.9963
//   0.9969   0.9969   0.9969   0.9969
//   0.9925   0.9925   0.9925   0.9925
//   1.0070   1.0070   1.0070   1.0070
//];
```

4.5.2 magma_sgels_gpu - unified memory version

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.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_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  double gpu_time, cpu_time;
  magma_int_t m = 8192, n = 8192;                  // a - mxn matrix
  magma_int_t nrhs = 100;      // b - nxnrhs, c - mxnrhs matrix
  float *a;                                        // a - mxn matrix
  float *b, *c;                 // b - nxnrhs, c - mxnrhs  matrix
  float *a1, *c1;        // a1 - mxn matrix, c1 - mxnrhs matrix
  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 a and c
  float *tau, *hwork, tmp[1]; // used in workspace preparation
  magma_int_t lworkgpu, lhwork;              // workspace sizes
  magma_int_t  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,n);//optim. block size for sgeqrf
  lworkgpu = (m-n + nb)*(nrhs+2*nb);
// prepare unified memory
  cudaMallocManaged(&tau,min_mn*sizeof(float));//u.mem.for tau
  cudaMallocManaged(&a,mn*sizeof(float)); // unified mem.for a
  cudaMallocManaged(&b,nnrhs*sizeof(float)); //unif. mem.for b
  cudaMallocManaged(&c,mnrhs*sizeof(float)); //unif. mem.for c
  cudaMallocManaged(&a1,mn*sizeof(float));//unified mem.for a1
  cudaMallocManaged(&c1,mnrhs*sizeof(float));//unif.mem for c1
// Get size for workspace
  lhwork = -1;
  lapackf77_sgeqrf(&m, &n, a, &m, tau, tmp, &lhwork, &info);
  l1 = (magma_int_t)MAGMA_D_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_D_REAL( tmp[0] );
  lhwork = max( max( l1, l2 ), lworkgpu );
  cudaMallocManaged(&hwork,lhwork*sizeof(float));//mem.f.hwork
  lapackf77_slarnv( &ione, ISEED, &mn, a );    // randomize  a
  lapackf77_slaset(MagmaFullStr,&n,&nrhs,&alpha,&alpha,b,&m);
                                   // b - mxnrhs 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,a1,ldda,queue);  // copy a -> a1
  magma_ssetmatrix( m, nrhs, c,m,c1,lddb,queue);   //  c -> c1
  gpu_time = magma_sync_wtime(NULL);
// solve the least squares problem  min ||a1*x-c1||
// using the QR decomposition,
// the solution overwrites c

  magma_sgels_gpu(MagmaNoTrans, m, n, nrhs, a1, ldda, c1, lddb,
                        hwork, lworkgpu, &info);

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("MAGMA time: %7.3f sec.\n",gpu_time);   // Magma time
  printf("upper left corner of of the magma_sgels sol.:\n");
  magma_sprint( 4, 4, c1, n );// part of the Magma QR solution
```

```
// LAPACK  version of sgels
  cpu_time=magma_wtime();
  lapackf77_sgels( MagmaNoTransStr, &m, &n, &nrhs,
                   a, &m, c, &m, hwork, &lhwork, &info);
  cpu_time=magma_wtime()-cpu_time;
  printf("LAPACK time: %7.3f sec.\n",cpu_time); // Lapack time
  printf("upper left corner of the lapackf77_sgels sol.:\n");
  magma_sprint( 4, 4, c, m );// part of the Lapack QR solution
  magma_free(tau);                              // free memory
  magma_free(a);                                // free memory
  magma_free(b);                                // free memory
  magma_free(c);                                // free memory
  magma_free(hwork);                            // free memory
  magma_free(a1);                               // free memory
  magma_free(c1);                               // free memory
  magma_queue_destroy(queue);                  // destroy queue
  magma_finalize( );                          // finalize Magma
  return EXIT_SUCCESS;
}

//MAGMA time:   0.358 sec.
//upper left corner of of the magma_sgels sol.:
//[
//   0.9899   0.9899   0.9899   0.9899
//   1.0087   1.0087   1.0087   1.0087
//   1.0115   1.0115   1.0115   1.0115
//   0.9993   0.9993   0.9993   0.9993
//];
//LAPACK time:  12.776 sec.
//upper left corner of the lapackf77_sgels sol.:
//[
//   0.9960   0.9960   0.9960   0.9960
//   0.9966   0.9966   0.9966   0.9966
//   0.9952   0.9952   0.9952   0.9952
//   1.0066   1.0066   1.0066   1.0066
//];
```

4.5.3 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. See magma-X.Y.Z/src/dgels_gpu.cpp for more details.

```
#include <stdio.h>
#include <cuda.h>
#include "magma_v2.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_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  double gpu_time, cpu_time;
  magma_int_t m = 8192, n = 8192;              // a - mxn matrix
  magma_int_t nrhs = 100;      // b - nxnrhs, c - mxnrhs matrix
  double *a;                       // a - mxn matrix on the host
  double *b, *c; // b - nxnrhs, c - mxnrhs  matrix on the host
  double *d_a, *d_c;  // d_a - mxn matrix, d_c - mxnrhs 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  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,n);//optim. block size 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 );    // randomize  a
  lapackf77_dlaset(MagmaFullStr,&n,&nrhs,&alpha,&alpha,b,&m);
                                // b - mxnrhs 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,queue);// copy a -> d_a
  magma_dsetmatrix( m, nrhs, c,m,d_c,lddb,queue); //  c -> d_c
  gpu_time = magma_sync_wtime(NULL);
// 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);

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("MAGMA time: %7.3f sec.\n",gpu_time);   // Magma time
 // Get the solution in b
  magma_dgetmatrix( n, nrhs, d_c, lddb,b,n,queue); // d_c -> b
  printf("upper left corner of of the magma_dgels sol.:\n");
  magma_dprint( 4, 4, b, n ); // part of the Magma QR solution
// LAPACK  version of dgels
  cpu_time=magma_wtime();
  lapackf77_dgels( MagmaNoTransStr, &m, &n, &nrhs,
                       a, &m, c, &m, hwork, &lhwork, &info);
  cpu_time=magma_wtime()-cpu_time;
  printf("LAPACK time: %7.3f sec.\n",cpu_time); // Lapack time
  printf("upper left corner of the lapackf77_dgels sol.:\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_queue_destroy(queue);                   // destroy queue
  magma_finalize( );                           // finalize Magma
  return EXIT_SUCCESS;
}
//MAGMA time:   3.157 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:  18.927 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
//];
```

4.5.4 magma_dgels_gpu - unified memory version

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.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_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  double gpu_time, cpu_time;
  magma_int_t m = 8192, n = 8192;               // a - mxn matrix
  magma_int_t nrhs = 100;      // b - nxnrhs, c - mxnrhs matrix
  double *a;                                    // a - mxn matrix
  double *b, *c;              // b - nxnrhs, c - mxnrhs  matrix
  double *a1, *c1;      // a1 - mxn matrix, c1 - mxnrhs matrix
  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 a and c
  double *tau, *hwork, tmp[1];// used in workspace preparation
  magma_int_t lworkgpu, lhwork;              // workspace sizes
  magma_int_t  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,n);//optim. block size for dgeqrf
  lworkgpu = (m-n + nb)*(nrhs+2*nb);
// prepare unified memory
  cudaMallocManaged(&tau,min_mn*sizeof(double)); //mem.for tau
  cudaMallocManaged(&a,mn*sizeof(double));// unified mem.for a
  cudaMallocManaged(&b,nnrhs*sizeof(double));// unif.mem.for b
  cudaMallocManaged(&c,mnrhs*sizeof(double));// unif.mem.for c
  cudaMallocManaged(&a1,mn*sizeof(double));// unif. mem.for a1
  cudaMallocManaged(&c1,mnrhs*sizeof(double));//uni.mem.for c1
// 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 );
  cudaMallocManaged(&hwork,lhwork*sizeof(double));//mem.-hwork
  lapackf77_dlarnv( &ione, ISEED, &mn, a );    // randomize  a
  lapackf77_dlaset(MagmaFullStr,&n,&nrhs,&alpha,&alpha,b,&m);
                                   // b - mxnrhs 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,a1,ldda,queue);  // copy a -> a1
  magma_dsetmatrix( m, nrhs, c,m,c1,lddb,queue);   //  c -> c1
  gpu_time = magma_sync_wtime(NULL);
// solve the least squares problem  min ||a1*x-c1||
// using the QR decomposition,
// the solution overwrites c

  magma_dgels_gpu(MagmaNoTrans, m, n, nrhs, a1, ldda, c1, lddb,
                  hwork, lworkgpu, &info);

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("MAGMA time: %7.3f sec.\n",gpu_time);   // Magma time
  printf("upper left corner of of the magma_dgels sol.:\n");
  magma_dprint( 4, 4, c1, n );// part of the Magma QR solution
// LAPACK  version of dgels
  cpu_time=magma_wtime();
  lapackf77_dgels( MagmaNoTransStr, &m, &n, &nrhs,
                   a, &m, c, &m, hwork, &lhwork, &info);
  cpu_time=magma_wtime()-cpu_time;
  printf("LAPACK time: %7.3f sec.\n",cpu_time); // Lapack time
  printf("upper left corner of the lapackf77_dgels sol.:\n");
  magma_dprint( 4, 4, c, m );// part of the Lapack QR solution
  magma_free(tau);                                // free memory
  magma_free(a);                                  // free memory
  magma_free(b);                                  // free memory
  magma_free(c);                                  // free memory
  magma_free(hwork);                              // free memory
  magma_free(a1);                                 // free memory
  magma_free(c1);                                 // free memory
  magma_queue_destroy(queue);                   // destroy queue
  magma_finalize( );                           // finalize Magma
  return EXIT_SUCCESS;
}
```

```
//MAGMA time:   3.168 sec.
//upper left corner of of the magma_dgels sol.:
//[
//    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:  19.405 sec.
//upper left corner of the lapackf77_dgels sol.:
//[
//    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
//];
```

4.5.5 magma_sgels3_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. See magma-X.Y.Z/src/sgels3_gpu.cpp for more details.

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime_api.h>
#include "magma_v2.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_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  double gpu_perf, cpu_perf;
  float  matnorm, work[1];
  float  c_one = MAGMA_S_ONE;
  float  c_neg_one = MAGMA_S_NEG_ONE;
  magma_int_t m = 8192, n = 8192, n2;
  magma_int_t nrhs = 4;
  float *a, *a2;                          // a, a2 - mxn matrices
                                                     // on the host
  float *b, *x, *r, *tau, *hwork, tmp[1];  // b, x, r - mxnrhs
                                     // matrices on the host
  float *d_a, *d_b;   // d_a - mxn matrix, d_b - mxnrhs matrix
                                            // on the device
  magma_int_t lda, ldb, ldda, lddb, lworkgpu, lhwork;
  magma_int_t i, info, min_mn, nb, l1, l2;
  magma_int_t *piv,ione      = 1;
  magma_int_t ISEED[4] = {0,0,0,1};
  ldda = ((m+31)/32)*32;
  lddb = ldda;
  n2 = m * n;
  min_mn = min(m, n);
  nb = magma_get_sgeqrf_nb(m,n);
  lda = ldb = m;
  lworkgpu = (m-n + nb)*(nrhs+2*nb);
  magma_smalloc_cpu(&tau,min_mn);       // host memory for tau
  magma_smalloc_cpu(&a,lda*n);            // host memory for a
  magma_smalloc_cpu(&a2,lda*n);          // host memory for a2
  magma_smalloc_cpu(&b,ldb*nrhs);         // host memory for b
  magma_smalloc_cpu(&x,ldb*nrhs);         // host memory for x
  magma_smalloc_cpu(&r,ldb*nrhs);         // host memory for r
  magma_smalloc(&d_a,ldda*n);          // device memory for d_a
  magma_smalloc(&d_b,lddb*nrhs);       // device memory for d_b
  piv=(magma_int_t*)malloc(n*sizeof(magma_int_t));// host mem.
// Get size for host workspace                         // for piv
  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, &lda, tau,
                    x, &ldb, 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 hwork
// randomize the matrices a, b
  lapackf77_slarnv( &ione, ISEED, &n2, a );
  n2 = m*nrhs;
  lapackf77_slarnv( &ione, ISEED, &n2, b );
// make copies of a and b: a-> a2, b -> r
  lapackf77_slacpy(MagmaFullStr,&m,&nrhs,b,&ldb,r,&ldb);
  lapackf77_slacpy(MagmaFullStr,&m,&m,a,&lda,a2,&lda);
// MAGMA
  magma_ssetmatrix( m,n,a,lda,d_a,ldda,queue);// copy a -> d_a
  magma_ssetmatrix( m,nrhs,b,ldb,d_b,lddb,queue); //  b -> d_b
  gpu_perf = magma_sync_wtime(NULL);
// solve the least squares problem  min ||d_a*x-d_b||
// using the QR decomposition,
// the solution overwrites d_b

  magma_sgels3_gpu( MagmaNoTrans, m, n, nrhs, d_a, ldda, d_b,
                    lddb, hwork, lworkgpu, &info);
  gpu_perf = magma_sync_wtime(NULL)-gpu_perf;
  printf("MAGMA time: %7.3f sec.\n",gpu_perf);   // Magma time
 // Get the solution in x
  magma_sgetmatrix( n, nrhs, d_b,lddb,x,ldb,queue);// d_b -> x
  printf("upper left corner of of the Magma solution:\n");
  magma_sprint( 4, 4, x, m );  // small part of Magma solution
// LAPACK  version of sgels
  cpu_perf=magma_wtime();
  lapackf77_sgels( MagmaNoTransStr, &m, &n, &nrhs,
                     a, &lda, b, &ldb, hwork, &lhwork, &info);
  cpu_perf=magma_wtime()-cpu_perf;
  printf("LAPACK time: %7.3f sec.\n",cpu_perf); // Lapack time
  printf("upper left corner of of the Lapack solution:\n");
  magma_sprint( 4, 4, b, m ); // small part of Lapack solution
  magma_sgesv(n,nrhs,a2,n,piv,r,n,&info);
  printf("upper left corner of of the Lapack sgesv solution\n"
         "for comparison:\n");
  magma_sprint( 4, 4, r, m ); // small part of Lapack solution
                                   // using LU decomposition
// Free memory
  free(tau);                                 // free host memory
  free(a);                                   // free host memory
  free(b);                                   // free host memory
  free(x);                                   // free host memory
  free(r);                                   // free host memory
  free(hwork);                               // free host memory
  magma_free(d_a);                         // free device memory
  magma_free(d_b);                         // free device memory
  magma_queue_destroy(queue);                   // destroy queue
  magma_finalize( );                           // finalize Magma
  return EXIT_SUCCESS;
}
//MAGMA time:   0.361 sec.
//upper left corner of of the Magma solution:
//[
//   1.4699 -10.2185  -5.7395  -5.9746
//  -2.2209   2.5485  -0.9190   3.5244
//  -2.9939   1.9242  -2.9884   4.5815
//   1.8249   3.5963   3.4809  -1.2047
//];
//LAPACK time:  11.019 sec.
//upper left corner of of the Lapack solution:
//[
//   1.4686 -10.2189  -5.7409  -5.9751
//  -2.2217   2.5500  -0.9187   3.5261
//  -2.9954   1.9264  -2.9890   4.5840
//   1.8256   3.5976   3.4825  -1.2042
//];
//upper left corner of of the Lapack sgesv solution
//for comparison:
```

```
//[
//    1.4768 -10.2685  -5.7679  -5.9999
//   -2.2205   2.5265  -0.9392   3.5168
//   -2.9938   1.8580  -3.0401   4.5591
//    1.8242   3.6547   3.5300  -1.1885
//];
```

4.5.6 magma_sgels3_gpu - unified memory version

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.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_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  double gpu_perf, cpu_perf;
  magma_int_t m = 8192, n = 8192, n2;
  magma_int_t nrhs = 4;
  float *a, *a2;                          // a, a2 -mxn matrices
// a used in Lapack sgels,   a2 -copy of a used in Magma sgesv
  float *b, *x, *r, *tau, *hwork, tmp[1];  // b, x, r - mxnrhs
// matr.: b used in Lapack sgels,   r copy used in Magma sgesv
  float *a1, *b1;       // a1 - mxn matrix, b1 - mxnrhs matrix
                         // copies of a, b used in Magma sgels
  magma_int_t lda, ldb, ldda, lddb, lworkgpu, lhwork;
  magma_int_t  info, min_mn, nb, l1, l2;
  magma_int_t *piv,ione   = 1;
  magma_int_t ISEED[4] = {0,0,0,1};
  ldda    = ((m+31)/32)*32;
  lddb    = ldda;
  n2      = m * n;
  min_mn = min(m, n);
  nb = magma_get_sgeqrf_nb(m,n);
  lda = ldb = m;
  lworkgpu = (m-n + nb)*(nrhs+2*nb);
// prepare unified memory
  cudaMallocManaged(&tau,min_mn*sizeof(float));  //mem.for tau
  cudaMallocManaged(&a,lda*n*sizeof(float)); // unif.mem.for a
  cudaMallocManaged(&a2,lda*n*sizeof(float));//unif.mem.for a2
  cudaMallocManaged(&b,ldb*nrhs*sizeof(float));//uni.mem.for b
  cudaMallocManaged(&x,ldb*nrhs*sizeof(float));//uni.mem.for x
  cudaMallocManaged(&r,ldb*nrhs*sizeof(float));//uni.mem.for r
  cudaMallocManaged(&a1,ldda*n*sizeof(float));//uni.mem.for a1
  cudaMallocManaged(&b1,lddb*nrhs*sizeof(float)); //mem.for b1
  cudaMallocManaged(&piv,n*sizeof(magma_int_t)); //mem.for piv
// Get size for  workspace
  lhwork = -1;
  lapackf77_sgeqrf(&m, &n, a, &m, tau, tmp, &lhwork, &info);
  l1 = (magma_int_t)MAGMA_D_REAL( tmp[0] );
  lhwork = -1;
  lapackf77_sormqr( MagmaLeftStr, MagmaTransStr,
                    &m, &nrhs, &min_mn, a, &lda, tau,
                    x, &ldb, tmp, &lhwork, &info);
  l2 = (magma_int_t)MAGMA_S_REAL( tmp[0] );
  lhwork = max( max( l1, l2 ), lworkgpu );
// magma_sgels3 needs this workspace
  cudaMallocManaged(&hwork,lhwork*sizeof(float));//mem.f.hwork
// randomize the matrices  a, b
  lapackf77_slarnv( &ione, ISEED, &n2, a );        // random a
  n2 = m*nrhs;                                   // size of b, x, r
  lapackf77_slarnv( &ione, ISEED, &n2, b );        // random b
// make copies of a and b: a-> a2, b -> r (they are overwrit.)
  lapackf77_slacpy(MagmaFullStr,&m,&nrhs,b,&ldb,r,&ldb);
  lapackf77_slacpy(MagmaFullStr,&m,&m,a,&lda,a2,&lda);
// copies of a,b for MAGMA
  magma_ssetmatrix(m,n,a,lda,a1,ldda,queue);   // copy a -> a1
  magma_ssetmatrix( m,nrhs,b,ldb,b1,lddb,queue);    // b -> b1
  gpu_perf = magma_sync_wtime(NULL);
// solve the least squares problem  min ||a1*x-b1||
// using the QR decomposition,
// the solution overwrites b1
// MAGMA version

  magma_sgels3_gpu( MagmaNoTrans, m, n, nrhs, a1, ldda, b1,
                    lddb, hwork, lworkgpu, &info);

  gpu_perf = magma_sync_wtime(NULL)-gpu_perf;
  printf("MAGMA time: %7.3f sec.\n",gpu_perf);   // Magma time
  printf("upper left corner of of the Magma  solution:\n");
  magma_sprint( 4, 4, b1, m ); // small part of Magma solution
// LAPACK version of sgels
  cpu_perf=magma_wtime();
  lapackf77_sgels( MagmaNoTransStr, &m, &n, &nrhs,
                   a, &lda, b, &ldb, hwork, &lhwork, &info);
  cpu_perf=magma_wtime()-cpu_perf;
  printf("LAPACK time: %7.3f sec.\n",cpu_perf); // Lapack time
  printf("upper left corner of of the Lapack solution:\n");
  magma_sprint( 4, 4, b, m ); // small part of Lapack solution
// MAGMA sgesv for comparison
  magma_sgesv(n,nrhs,a2,n,piv,r,n,&info);
  printf("upper left corner of of the Lapack sgesv solution\n"
         "for comparison:\n");
  magma_sprint( 4, 4, r, m );  // small part of dgesv solution
                                    // using LU decomposition
// Free unified memory
```

```
  magma_free(tau);                                        // free memory
  magma_free(a);                                          // free memory
  magma_free(a2);                                         // free memory
  magma_free(b);                                          // free memory
  magma_free(x);                                          // free memory
  magma_free(r);                                          // free memory
  magma_free(hwork);                                      // free memory
  magma_free(a1);                                         // free memory
  magma_free(b1);                                         // free memory
  magma_queue_destroy(queue);                            // destroy queue
  magma_finalize( );                                    // finalize Magma
  return EXIT_SUCCESS;
}

//MAGMA time:   0.357 sec.
//upper left corner of of the Magma  solution:
//[
//   1.4699 -10.2185  -5.7395  -5.9746
//  -2.2209   2.5485  -0.9190   3.5244
//  -2.9939   1.9242  -2.9884   4.5815
//   1.8249   3.5963   3.4809  -1.2047
//];
//LAPACK time:  12.676 sec.
//upper left corner of of the Lapack solution:
//[
//   1.4686 -10.2189  -5.7409  -5.9751
//  -2.2217   2.5500  -0.9187   3.5261
//  -2.9954   1.9264  -2.9890   4.5840
//   1.8256   3.5976   3.4825  -1.2042
//];
//upper left corner of of the Lapack sgesv solution
//for comparison:
//[
//   1.4768 -10.2685  -5.7679  -5.9999
//  -2.2205   2.5265  -0.9392   3.5168
//  -2.9938   1.8580  -3.0401   4.5591
//   1.8242   3.6547   3.5300  -1.1885
//];
```

4.5.7 magma_dgels3_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. See magma-X.Y.Z/src/dgels3_gpu.cpp for more details.

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime_api.h>
#include "magma_v2.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_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  double gpu_perf, cpu_perf;
  double  matnorm, work[1];
  double  c_one = MAGMA_D_ONE;
  double  c_neg_one = MAGMA_D_NEG_ONE;
  magma_int_t m = 8192, n = 8192, n2;
  magma_int_t nrhs = 4;
  double *a, *a2;                         // a, a2 - mxn matrices
                                                   // on the host
  double *b, *x, *r, *tau, *hwork, tmp[1]; // b, x, r - mxnrhs
                                          // matrices on the host
  double *d_a, *d_b;  // d_a - mxn matrix, d_b - mxnrhs matrix
                                                 // on the device
  magma_int_t lda, ldb, ldda, lddb, lworkgpu, lhwork;
  magma_int_t i, info, min_mn, nb, l1, l2;
  magma_int_t *piv,ione   = 1;
  magma_int_t ISEED[4] = {0,0,0,1};
  ldda   = ((m+31)/32)*32;
  lddb   = ldda;
  n2     = m * n;
  min_mn = min(m, n);
  nb = magma_get_dgeqrf_nb(m,n);
  lda = ldb = m;
  lworkgpu = (m-n + nb)*(nrhs+2*nb);
  magma_dmalloc_cpu(&tau,min_mn);       // host memory for tau
  magma_dmalloc_cpu(&a,lda*n);            // host memory for a
  magma_dmalloc_cpu(&a2,lda*n);          // host memory for a2
  magma_dmalloc_cpu(&b,ldb*nrhs);         // host memory for b
  magma_dmalloc_cpu(&x,ldb*nrhs);         // host memory for x
  magma_dmalloc_cpu(&r,ldb*nrhs);         // host memory for r
  magma_dmalloc(&d_a,ldda*n);         // device memory for d_a
  magma_dmalloc(&d_b,lddb*nrhs);      // device memory for d_b
  piv=(magma_int_t*)malloc(n*sizeof(magma_int_t));// host mem.
// Get size for host workspace                       // for piv
  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, &lda, tau,
                          x, &ldb, tmp, &lhwork, &info);
  l2 = (magma_int_t)MAGMA_S_REAL( tmp[0] );
  lhwork = max( max( l1, l2 ), lworkgpu );
  magma_dmalloc_cpu(&hwork,lhwork);   // host memory for hwork
// randomize the matrices  a, b
  lapackf77_dlarnv( &ione, ISEED, &n2, a );
  n2 = m*nrhs;                              // size of b, x, r
  lapackf77_dlarnv( &ione, ISEED, &n2, b );
// make copies of a and b: a-> a2, b -> r
  lapackf77_dlacpy(MagmaFullStr,&m,&nrhs,b,&ldb,r,&ldb);
  lapackf77_dlacpy(MagmaFullStr,&m,&m,a,&lda,a2,&lda);
// MAGMA
  magma_dsetmatrix(m,n,a,lda,d_a,ldda,queue);// copy a -> d_a
  magma_dsetmatrix( m,nrhs,b,ldb,d_b,lddb,queue); // b -> d_b
  gpu_perf = magma_sync_wtime(NULL);
// solve the least squares problem  min ||d_a*x-d_b||
// using the QR decomposition,
// the solution overwrites d_b

  magma_dgels3_gpu( MagmaNoTrans, m, n, nrhs, d_a, ldda, d_b,
                          lddb, hwork, lworkgpu, &info);

  gpu_perf = magma_sync_wtime(NULL)-gpu_perf;
  printf("MAGMA time: %7.3f sec.\n",gpu_perf);   // Magma time
// copy the solution to x
  magma_dgetmatrix(n, nrhs, d_b, lddb,x,ldb,queue);// d_b -> x
  printf("upper left corner of of the Magma  solution:\n");
  magma_dprint( 4, 4, x, m );  // small part of Magma solution
// LAPACK version of dgels
  cpu_perf=magma_wtime();
  lapackf77_dgels( MagmaNoTransStr, &m, &n, &nrhs,
                         a, &lda, b, &ldb, hwork, &lhwork, &info);
  cpu_perf=magma_wtime()-cpu_perf;
  printf("LAPACK time: %7.3f sec.\n",cpu_perf); // Lapack time
  printf("upper left corner of of the Lapack solution:\n");
  magma_dprint( 4, 4, b, m ); // small part of Lapack solution
  magma_dgesv(n,nrhs,a2,n,piv,r,n,&info);
  printf("upper left corner of of the Lapack dgesv solution\n"
          "for comparison:\n");
  magma_dprint( 4, 4, r, m ); // small part of Lapack solution
                                   // using LU decomposition
// Free memory
  free(tau);                                 // free host memory
  free(a);                                   // free host memory
  free(b);                                   // free host memory
  free(x);                                   // free host memory
  free(r);                                   // free host memory
  free(hwork);                               // free host memory
  magma_free(d_a);                         // free device memory
  magma_free(d_b);                         // free device memory
  magma_queue_destroy(queue);                   // destroy queue
  magma_finalize( );                           // finalize Magma
  return EXIT_SUCCESS;
}
//MAGMA time:   3.032 sec.
//upper left corner of of the Magma  solution:
//[
//   -2.9416   0.1624   0.2631  -2.0923
//   -0.0242   0.5965  -0.4656  -0.3765
//    0.6595   0.5525   0.5783  -0.1609
//   -0.5521  -1.2515   0.0901  -0.2223
//];
//LAPACK time:  18.957 sec.
//upper left corner of of the Lapack solution:
//[
//   -2.9416   0.1624   0.2631  -2.0923
//   -0.0242   0.5965  -0.4656  -0.3765
//    0.6595   0.5525   0.5783  -0.1609
//   -0.5521  -1.2515   0.0901  -0.2223
//];
//upper left corner of of the Lapack dgesv solution
//for comparison:
//[
//   -2.9416   0.1624   0.2631  -2.0923
//   -0.0242   0.5965  -0.4656  -0.3765
//    0.6595   0.5525   0.5783  -0.1609
//   -0.5521  -1.2515   0.0901  -0.2223
//];
```

4.5.8 magma_dgels3_gpu - unified memory version

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.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_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  double gpu_perf, cpu_perf;
  magma_int_t m = 8192, n = 8192, n2;
  magma_int_t nrhs = 4;
  double *a, *a2;                          // a, a2 -mxn matrices
// a used in Lapack dgels,   a2 -copy of a used in Magma dgesv
  double *b, *x, *r, *tau, *hwork, tmp[1]; // b, x, r - mxnrhs
// matr.: b used in Lapack dgels,   r copy used in Magma dgesv
  double *a1, *b1;      // a1 - mxn matrix, b1 - mxnrhs matrix
```

```
                        // copies of a, b used in Magma dgels
  magma_int_t lda, ldb, ldda, lddb, lworkgpu, lhwork;
  magma_int_t  info, min_mn, nb, l1, l2;
  magma_int_t *piv,ione  = 1;
  magma_int_t ISEED[4] = {0,0,0,1};
  ldda    = ((m+31)/32)*32;
  lddb    = ldda;
  n2      = m * n;
  min_mn = min(m, n);
  nb = magma_get_dgeqrf_nb(m,n);
  lda = ldb = m;
  lworkgpu = (m-n + nb)*(nrhs+2*nb);
// prepare unified memory
  cudaMallocManaged(&tau,min_mn*sizeof(double)); //mem.for tau
  cudaMallocManaged(&a,lda*n*sizeof(double));// unif.mem.for a
  cudaMallocManaged(&a2,lda*n*sizeof(double)); //un.mem.for a2
  cudaMallocManaged(&b,ldb*nrhs*sizeof(double)); //u.mem.for b
  cudaMallocManaged(&x,ldb*nrhs*sizeof(double)); //u.mem.for x
  cudaMallocManaged(&r,ldb*nrhs*sizeof(double)); //u.mem.for r
  cudaMallocManaged(&a1,ldda*n*sizeof(double));//un.mem.for a1
  cudaMallocManaged(&b1,lddb*nrhs*sizeof(double));//mem.for b1
  cudaMallocManaged(&piv,n*sizeof(magma_int_t)); //mem.for piv
// 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, &lda, tau,
                    x, &ldb, tmp, &lhwork, &info);
  l2 = (magma_int_t)MAGMA_S_REAL( tmp[0] );
  lhwork = max( max( l1, l2 ), lworkgpu );
// magma_dgels3_gpu needs this workspace
  cudaMallocManaged(&hwork,lhwork*sizeof(double));//mem.-hwork
// randomize the matrices  a, b
  lapackf77_dlarnv( &ione, ISEED, &n2, a );        // random a
  n2 = m*nrhs;                                  // size of b, x, r
  lapackf77_dlarnv( &ione, ISEED, &n2, b );        // random b
// make copies of a and b: a-> a2, b -> r (they are overwrit.)
  lapackf77_dlacpy(MagmaFullStr,&m,&nrhs,b,&ldb,r,&ldb);
  lapackf77_dlacpy(MagmaFullStr,&m,&m,a,&lda,a2,&lda);
// copies of a,b for MAGMA
  magma_dsetmatrix(m,n,a,lda,a1,ldda,queue);   // copy a -> a1
  magma_dsetmatrix( m,nrhs,b,ldb,b1,lddb,queue);     // b -> b1
  gpu_perf = magma_sync_wtime(NULL);
// solve the least squares problem  min ||a1*x-b1||
// using the QR decomposition,
// the solution overwrites b1
// MAGMA version

  magma_dgels3_gpu( MagmaNoTrans, m, n, nrhs, a1, ldda, b1,
                    lddb, hwork, lworkgpu, &info);
  gpu_perf = magma_sync_wtime(NULL)-gpu_perf;
  printf("MAGMA time: %7.3f sec.\n",gpu_perf);   // Magma time
  printf("upper left corner of of the Magma  solution:\n");
  magma_dprint( 4, 4, b1, m ); // small part of Magma solution
// LAPACK version of dgels
  cpu_perf=magma_wtime();
  lapackf77_dgels( MagmaNoTransStr, &m, &n, &nrhs,
                      a, &lda, b, &ldb, hwork, &lhwork, &info);
  cpu_perf=magma_wtime()-cpu_perf;
  printf("LAPACK time: %7.3f sec.\n",cpu_perf); // Lapack time
  printf("upper left corner of of the Lapack solution:\n");
  magma_dprint( 4, 4, b, m ); // small part of Lapack solution
// MAGMA dgesv for comparison
  magma_dgesv(n,nrhs,a2,n,piv,r,n,&info);
  printf("upper left corner of of the Lapack dgesv solution\n"
          "for comparison:\n");
  magma_dprint( 4, 4, r, m );  // small part of dgesv solution
                                   // using LU decomposition
// Free unified memory
  magma_free(tau);                                  // free memory
  magma_free(a);                                    // free memory
  magma_free(a2);                                   // free memory
  magma_free(b);                                    // free memory
  magma_free(x);                                    // free memory
  magma_free(r);                                    // free memory
  magma_free(hwork);                                // free memory
  magma_free(a1);                                   // free memory
  magma_free(b1);                                   // free memory
  magma_queue_destroy(queue);                     // destroy queue
  magma_finalize( );                             // finalize Magma
  return EXIT_SUCCESS;
}
//MAGMA time:   3.047 sec.
//upper left corner of of the Magma  solution:
//[
//   -2.9416   0.1624   0.2631  -2.0923
//   -0.0242   0.5965  -0.4656  -0.3765
//    0.6595   0.5525   0.5783  -0.1609
//   -0.5521  -1.2515   0.0901  -0.2223
//];
//LAPACK time:  21.545 sec.
//upper left corner of of the Lapack solution:
//[
//   -2.9416   0.1624   0.2631  -2.0923
//   -0.0242   0.5965  -0.4656  -0.3765
//    0.6595   0.5525   0.5783  -0.1609
//   -0.5521  -1.2515   0.0901  -0.2223
//];
//upper left corner of of the Lapack dgesv solution
//for comparison:
```

```
//[
//   -2.9416    0.1624    0.2631   -2.0923
//   -0.0242    0.5965   -0.4656   -0.3765
//    0.6595    0.5525    0.5783   -0.1609
//   -0.5521   -1.2515    0.0901   -0.2223
//];
```

4.5.9 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 $H(1)\ldots H(\min(m,n))$, 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.cpp for more details.

```
#include <stdio.h>
#include <cuda.h>
#include "magma_v2.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
  double 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 *hwork, tmp[1];      // hwork - workspace; tmp -used in
  magma_int_t 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,n); //optim.block size for sgetrf
  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);   // host memory for hwork
// Randomize the matrix
  lapackf77_slarnv( &ione, ISEED, &n2, a );     // randomize a
  lapackf77_slacpy(MagmaFullStr,&m,&n,a,&m,r,&m);       // a->r
// MAGMA
  gpu_time = magma_sync_wtime(NULL);
// 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);

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("MAGMA time: %7.3f sec.\n",gpu_time);  // print Magma
// LAPACK                                                  time
  cpu_time=magma_wtime();
  lapackf77_sgeqrf(&m,&n,r,&m,tau,hwork,&lhwork,&info);
  cpu_time=magma_wtime()-cpu_time;
  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 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:   0.310 sec.
//LAPACK time:   1.397 sec.
//difference:  1.860795e-06
```

4.5.10 magma_sgeqrf - unified memory version

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.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
  double gpu_time, cpu_time;
  magma_int_t m = 3072, n = 4096, n2=m*n;
  float *a, *r;                              // a,r - mxn matrices
  float *tau;    // scalars defining the elementary reflectors
  float *hwork, tmp[1];     // hwork - workspace; tmp -used in
  magma_int_t  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
  cudaMallocManaged(&tau,min_mn*sizeof(float));  //mem.for tau
  cudaMallocManaged(&a,n2*sizeof(float)); // unified.mem.for b
  cudaMallocManaged(&r,n2*sizeof(float)); // unified.mem.for r
// Get size for workspace
  nb = magma_get_sgeqrf_nb(m,n); //optim.block size for sgetrf
  lhwork = -1;
  lapackf77_sgeqrf(&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));
  cudaMallocManaged(&hwork,lhwork*sizeof(float));//mem.f.hwork
// Randomize the matrix
  lapackf77_slarnv( &ione, ISEED, &n2, a );     // randomize a
  lapackf77_slacpy(MagmaFullStr,&m,&n,a,&m,r,&m);       // a->r
// MAGMA
  gpu_time = magma_sync_wtime(NULL);
// compute a QR factorization of a real mxn matrix a
// a=Q*R

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

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("MAGMA time: %7.3f sec.\n",gpu_time);  // print Magma
// LAPACK                                                  time
  cpu_time=magma_wtime();
  lapackf77_sgeqrf(&m,&n,r,&m,tau,hwork,&lhwork,&info);
  cpu_time=magma_wtime()-cpu_time;
  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 memory
  magma_free(tau);                                // free memory
  magma_free(hwork);                              // free memory
  magma_free(a);                                  // free memory
  magma_free(r);                                  // free memory
  magma_finalize( );                            // finalize Magma
  return EXIT_SUCCESS;
}
//MAGMA time:   0.321 sec.
```

```
//LAPACK time:   0.775 sec.
//difference:  2.625919e-06
```

4.5.11 `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 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.cpp` for more details.

```
#include <stdio.h>
#include <cuda.h>
#include "magma_v2.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
  double gpu_time, cpu_time;
  magma_int_t m = 3072, n = 4096, 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,n); //optim.block size for dgetrf
  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);   // host memory for hwork
// Randomize the matrix
  lapackf77_dlarnv( &ione, ISEED, &n2, a );     // randomize a
  lapackf77_dlacpy(MagmaFullStr,&m,&n,a,&m,r,&m);      // a->r
// MAGMA
  gpu_time = magma_sync_wtime(NULL);
// compute a QR factorization of a real mxn matrix a
// a=Q*R

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

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("MAGMA time: %7.3f sec.\n",gpu_time);  // print Magma
// LAPACK                                                 time
  cpu_time=magma_wtime();
  lapackf77_dgeqrf(&m,&n,r,&m,tau,hwork,&lhwork,&info);
  cpu_time=magma_wtime()-cpu_time;
  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 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:   0.561 sec.
//LAPACK time:   1.521 sec.
//difference:  4.705079e-15
```

4.5.12 magma_dgeqrf - unified memory version

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.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
  double gpu_time, cpu_time;
  magma_int_t m = 3072, n = 4096, n2=m*n;// a,r - mxn matrices
  double *a, *r;                                // on the host
  double *tau;   // scalars defining the elementary reflectors
  double *hwork, tmp[1];    // hwork - workspace; tmp -used in
  magma_int_t  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
  cudaMallocManaged(&tau,min_mn*sizeof(double));//u.mem.for tau
  cudaMallocManaged(&a,n2*sizeof(double)); // unified mem.for a
  cudaMallocManaged(&r,n2*sizeof(double)); // unified mem.for r
// Get size for workspace
  nb = magma_get_dgeqrf_nb(m,n); // optim.block size for dgetrf
  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));
  cudaMallocManaged(&hwork,lhwork*sizeof(double));//mem.f.hwork
// Randomize the matrix
  lapackf77_dlarnv( &ione, ISEED, &n2, a );     // randomize a
  lapackf77_dlacpy(MagmaFullStr,&m,&n,a,&m,r,&m);       // a->r
// MAGMA
  gpu_time = magma_sync_wtime(NULL);
// compute a QR factorization of a real mxn matrix a
// a=Q*R

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

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("MAGMA time: %7.3f sec.\n",gpu_time);  // print Magma
// LAPACK                                                  time
  cpu_time=magma_wtime();
  lapackf77_dgeqrf(&m,&n,r,&m,tau,hwork,&lhwork,&info);
  cpu_time=magma_wtime()-cpu_time;
  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 memory
  magma_free(tau);                              // free memory
  magma_free(hwork);                            // free memory
  magma_free(a);                                // free memory
  magma_free(r);                                // free memory
  magma_finalize( );                          // finalize Magma
  return EXIT_SUCCESS;
}
//MAGMA time:   0.614 sec.
//LAPACK time:   1.521 sec.
//difference:  4.705079e-15
```

4.5.13 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_v2.h"
#include "magma_lapack.h"
#define min(a,b)  (((a)<(b))?(a):(b))
int main( int argc, char** argv)
{
  magma_init();                              // initialize Magma
  magma_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  double 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
  magma_int_t  info, min_mn;             // in worksp.size comp.
  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);    // host memory for hwork
```

```
                                        // Lapack needs this array
// Randomize the matrix
  lapackf77_slarnv( &ione, ISEED, &n2, a );      // randomize a
// MAGMA
  magma_ssetmatrix( m, n, a,m,d_a,ldda,queue);// copy a -> d_a
  gpu_time = magma_sync_wtime(NULL);
// 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);

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("MAGMA time: %7.3f sec.\n",gpu_time);   // Magma time
// LAPACK
  cpu_time=magma_wtime();
  lapackf77_sgeqrf(&m,&n,a,&m,tau,hwork,&lhwork,&info);
  cpu_time=magma_wtime()-cpu_time;
  printf("LAPACK time: %7.3f sec.\n",cpu_time); // Lapack time
// difference
  magma_sgetmatrix( m, n,d_a,ldda,r,m,queue); // 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_queue_destroy(queue);                     // destroy queue
  magma_finalize( );                             // finalize Magma
  return EXIT_SUCCESS;
}
//MAGMA time:   0.335 sec.
//LAPACK time:  10.037 sec.
//difference:  2.670853e-06
```

4.5.14 magma_sgeqrf_gpu - unified memory version

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.h"
#include "magma_lapack.h"
#define min(a,b)  (((a)<(b))?(a):(b))
int main( int argc, char** argv)
```

```
{
  magma_init();                                // initialize Magma
  magma_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  double gpu_time, cpu_time;
  magma_int_t m = 8192, n = 8192, n2=m*n, ldda;
  float *a, *r;                                // a, r - mxn matrices
  float *a1;       // a1 mxn matrix,  used by Magma sgeqrf2_gpu
  float *tau;    // scalars defining the elementary reflectors
  float *hwork, tmp[1];      // hwork - workspace; tmp -used in
  magma_int_t  info, min_mn;              // comp. workspace size
  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_D_NEG_ONE;
  float matnorm, work[1];    // used in difference computations
// prepare unified memory
  cudaMallocManaged(&tau,min_mn*sizeof(float));//u.mem.for tau
  cudaMallocManaged(&a,n2*sizeof(float));  //unified mem.for a
  cudaMallocManaged(&r,n2*sizeof(float));  //unified mem.for r
  cudaMallocManaged(&a1,ldda*n*sizeof(float));//uni.mem.for a1
// Get size for workspace
  lhwork = -1;
  lapackf77_sgeqrf(&m,&n,a,&m,tau,tmp,&lhwork,&info);
  lhwork = (magma_int_t)MAGMA_D_REAL( tmp[0] );
  cudaMallocManaged(&hwork,lhwork*sizeof(float));//mem.f.hwork
// Randomize the matrix
  lapackf77_slarnv( &ione, ISEED, &n2, a );      // randomize a
// MAGMA
  magma_ssetmatrix( m, n, a,m,a1,ldda,queue);  // copy a -> a1
  gpu_time = magma_sync_wtime(NULL);
// compute a QR factorization of a real mxn matrix a1
// a1=Q*R, Q - orthogonal, R - upper triangular

  magma_sgeqrf2_gpu( m, n, a1, ldda, tau, &info);

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("MAGMA time: %7.3f sec.\n",gpu_time);   // Magma time
// LAPACK
  cpu_time=magma_wtime();
  lapackf77_sgeqrf(&m,&n,a,&m,tau,hwork,&lhwork,&info);
  cpu_time=magma_wtime()-cpu_time;
  printf("LAPACK time: %7.3f sec.\n",cpu_time); // Lapack time
// difference
  magma_sgetmatrix( m, n, a1,ldda,r,m,queue);  // copy a1 -> 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
```

```
  magma_free(tau);                                    // free memory
  magma_free(hwork);                                  // free memory
  magma_free(a);                                      // free memory
  magma_free(r);                                      // free memory
  magma_free(a1);                                     // free memory
  magma_queue_destroy(queue);                       // destroy queue
  magma_finalize( );                                // finalize Magma
  return EXIT_SUCCESS;
}
//MAGMA time:   0.341 sec.
//LAPACK time:  11.828 sec.
//difference:  2.670853e-06
```

4.5.15 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 $H(1)\ldots H(\min(m,n))$, 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_gpu.cpp for more details.

```
#include <stdio.h>
#include <cuda.h>
#include "magma_v2.h"
#include "magma_lapack.h"
#define min(a,b)  (((a)<(b))?(a):(b))
int main( int argc, char** argv)
{
  magma_init();                                   // initialize Magma
  magma_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  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  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);   // host memory for hwork
                         // Lapack version needs  this array
// Randomize the matrix
  lapackf77_dlarnv( &ione, ISEED, &n2, a );     // randomize a
// MAGMA
  magma_dsetmatrix( m, n, a,m,d_a,ldda,queue);// copy a -> d_a
  gpu_time = magma_sync_wtime(NULL);
// 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);

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("MAGMA time: %7.3f sec.\n",gpu_time);   // Magma time
// LAPACK
  cpu_time=magma_wtime();
  lapackf77_dgeqrf(&m,&n,a,&m,tau,hwork,&lhwork,&info);
  cpu_time=magma_wtime()-cpu_time;
  printf("LAPACK time: %7.3f sec.\n",cpu_time); // Lapack time
// difference
  magma_dgetmatrix( m, n, d_a,ldda,r,m,queue);// 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_queue_destroy(queue);                   // destroy queue
  magma_finalize( );                           // finalize Magma
  return EXIT_SUCCESS;
}
//MAGMA time:   2.955 sec.
//LAPACK time:  16.932 sec.
//difference:  4.933266e-15
```

4.5.16 magma_dgeqrf_gpu - unified memory version

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.h"
#include "magma_lapack.h"
#define min(a,b)  (((a)<(b))?(a):(b))
int main( int argc, char** argv)
{
  magma_init();                             // initialize Magma
  magma_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  double gpu_time, cpu_time;
  magma_int_t m = 8192, n = 8192, n2=m*n, ldda;
  double *a, *r;                          // a, r - mxn matrices
  double *a1;     // a1 mxn matrix,  used by Magma dgeqrf2_gpu
  double *tau;   // scalars defining the elementary reflectors
  double *hwork, tmp[1];    // hwork - workspace; tmp -used in
  magma_int_t  info, min_mn;             // comp. workspace size
  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
// prepare unified memory
  cudaMallocManaged(&tau,min_mn*sizeof(double)); //mem.for tau
  cudaMallocManaged(&a,n2*sizeof(double)); //unified.mem.for a
  cudaMallocManaged(&r,n2*sizeof(double)); //unified.mem.for r
  cudaMallocManaged(&a1,ldda*n*sizeof(double));//un.mem.for a1
// 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] );
  cudaMallocManaged(&hwork,lhwork*sizeof(double)); //mem-hwork
// Randomize the matrix
  lapackf77_dlarnv( &ione, ISEED, &n2, a );     // randomize a
// MAGMA
  magma_dsetmatrix( m, n, a,m,a1,ldda,queue);  // copy a -> a1
  gpu_time = magma_sync_wtime(NULL);
// compute a QR factorization of a real mxn matrix a1
// a1=Q*R, Q - orthogonal, R - upper triangular

  magma_dgeqrf2_gpu( m, n, a1, ldda, tau, &info);

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("MAGMA time: %7.3f sec.\n",gpu_time);   // Magma time
// LAPACK
  cpu_time=magma_wtime();
  lapackf77_dgeqrf(&m,&n,a,&m,tau,hwork,&lhwork,&info);
  cpu_time=magma_wtime()-cpu_time;
  printf("LAPACK time: %7.3f sec.\n",cpu_time); // Lapack time
// difference
  magma_dgetmatrix( m, n, a1,ldda,r,m,queue);  // copy a1 -> 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
  magma_free(tau);                                // free memory
  magma_free(hwork);                              // free memory
  magma_free(a);                                  // free memory
  magma_free(r);                                  // free memory
  magma_free(a1);                                 // free memory
  magma_queue_destroy(queue);                    // destroy queue
  magma_finalize( );                             // finalize Magma
  return EXIT_SUCCESS;
}
//MAGMA time:   3.014 sec.
//LAPACK time:  17.932 sec.
//difference:  4.933266e-15
```

4.5.17 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_v2.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
  int num_gpus = 1;                          // for num_gpus GPUs
  magma_setdevice(0);
  magma_queue_t queues[num_gpus];
  for( int dev = 0; dev < num_gpus; ++dev ) {
      magma_queue_create( dev, &queues[dev] );
  }
  double cpu_time, gpu_time;
  magma_int_t m = 8192, n = m, n2=m*n;
  float *a, *r ;              // a, r - mxn matrices on the host
  magmaFloat_ptr  d_la[num_gpus];        // pointers to memeory
                                         // 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, info, min_mn= min(m, n);

  magma_int_t ione = 1, lhwork;      // lhwork - workspace size
  float c_neg_one = MAGMA_D_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,n);//optim.block size
  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;
    magma_setdevice(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]);
  }
  magma_setdevice(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); // host memory for h_work
                           //Lapack sgeqrf needs this  array
// Randomize the matrix a and copy a -> r
  lapackf77_slarnv(&ione,ISEED,&n2,a);
  lapackf77_slacpy(MagmaFullStr,&m,&n,a,&m,r,&m);      // a->r
// LAPACK
  cpu_time=magma_wtime();
// QR decomposition on the host
```

```
  lapackf77_sgeqrf(&m,&n,a,&m,tau,h_work,&lhwork,&info);
  cpu_time=magma_wtime()-cpu_time;
  printf("Lapack dgeqrf time: %7.5f sec.\n",cpu_time);
// MAGMA                                    // print Lapack time
  magma_ssetmatrix_1D_col_bcyclic(num_gpus, m, n,nb, r,m,d_la,
           ldda, queues);  // distribute r -> num_gpus devices
  gpu_time = magma_sync_wtime(NULL);
// QR decomposition on num_gpus devices

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

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("Magma dgeqrf_mgpu time: %7.5f sec.\n",gpu_time);
                                              // print Magma time
  magma_sgetmatrix_1D_col_bcyclic(num_gpus,m, n,nb, d_la,ldda,
             r, m, queues);    // 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
  }
  for( int dev = 0; dev < num_gpus; ++dev ) {
    magma_queue_destroy( queues[dev] );
  }
  magma_finalize( );                            // finalize Magma
  return EXIT_SUCCESS;
}
//Number of GPUs to be used = 1
//device  0 n_local=8192
//Lapack dgeqrf time: 10.11191 sec.
//Magma dgeqrf_mgpu time: 0.33583 sec.
//difference: 2.670853e-06
```

4.5.18 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_v2.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
  int num_gpus = 1;                         // for num_gpus GPUs
  magma_setdevice(0);
  magma_queue_t queues[num_gpus];
  for( int dev = 0; dev < num_gpus; ++dev ) {
      magma_queue_create( dev, &queues[dev] );
  }
  double cpu_time, gpu_time;
  magma_int_t m = 8192, n = m, n2=m*n;
  double *a, *r ;              // a, r - mxn matrices on the host
  magmaDouble_ptr  d_la[num_gpus];       // pointers to memeory
                                         // 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, info, min_mn= min(m, n);

  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_dgeqrf_nb(m,n);//optim.block size
  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;
    magma_setdevice(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]);
  }
  magma_setdevice(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); // host memory for h_work
                                 //Lapack sgeqrf needs this array
// Randomize the matrix a and copy a -> r
  lapackf77_dlarnv(&ione,ISEED,&n2,a);
  lapackf77_dlacpy(MagmaFullStr,&m,&n,a,&m,r,&m);// a->r
// LAPACK
  cpu_time=magma_wtime();
// QR decomposition on the host
  lapackf77_dgeqrf(&m,&n,a,&m,tau,h_work,&lhwork,&info);
  cpu_time=magma_wtime()-cpu_time;
  printf("Lapack dgeqrf time: %7.5f sec.\n",cpu_time);
// MAGMA                                 // print Lapack time
  magma_dsetmatrix_1D_col_bcyclic(num_gpus, m, n,nb, r,m,d_la,
           ldda, queues);  // distribute r -> num_gpus devices
  gpu_time = magma_sync_wtime(NULL);
// QR decomposition on num_gpus devices

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

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("Magma dgeqrf_mgpu time: %7.5f sec.\n",gpu_time);
                                          // print Magma time
  magma_dgetmatrix_1D_col_bcyclic(num_gpus,m, n,nb, d_la,ldda,
             r, m, queues);    // 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
  }
  for( int dev = 0; dev < num_gpus; ++dev ) {
    magma_queue_destroy( queues[dev] );
  }
  magma_finalize( );                        // finalize Magma
  return EXIT_SUCCESS;
}
//Number of GPUs to be used = 1
//device  0 n_local=8192
//Lapack dgeqrf time: 16.91422 sec.
//Magma dgeqrf_mgpu time: 2.99641 sec.
//difference: 4.933266e-15
```

4.5.19 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_v2.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
  double  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  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,n); //optim.block size 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);//host memory for h_work
// Randomize the matrix a and copy a -> r
  lapackf77_slarnv( &ione, ISEED, &n2, a );
  lapackf77_slacpy(MagmaFullStr,&m,&n,a,&m,r,&m );
// MAGMA
  gpu_time = magma_sync_wtime(NULL);
// LQ factorization for a real matrix r=L*Q using Magma
// L - lower triangular, Q - orthogonal

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

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("MAGMA time: %7.3f sec.\n",gpu_time);  // print Magma
// LAPACK                                                   time
  cpu_time=magma_wtime();
// LQ factorization for a real matrix a=L*Q on the host
  lapackf77_sgelqf(&m,&n,a,&m,tau,h_work,&lwork,&info);
  cpu_time=magma_wtime()-cpu_time;
  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:   0.318 sec.
//LAPACK time:   2.394 sec.
//difference: 1.846170e-06
```

4.5.20 magma_sgelqf - unified memory version

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.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
  double  gpu_time, cpu_time;
  magma_int_t m = 4096, n = 4096, n2=m*n;
  float *a, *r;                            // a, r - mxn matrices
  float *tau;    // scalars defining the elementary reflectors
  float *h_work, tmp[1];   // h_work - workspace; tmp -used in
                                             // workspace query
  magma_int_t  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_D_NEG_ONE;
  min_mn = min(m, n);
  nb = magma_get_sgeqrf_nb(m,n);//optim. block size for sgeqrf
// prepare unified memory
  cudaMallocManaged(&tau,min_mn*sizeof(float));//u.mem.for tau
  cudaMallocManaged(&a,n2*sizeof(float)); //unif. memory for a
  cudaMallocManaged(&r,n2*sizeof(float)); //unif. memory for r
// Get size for workspace
  lwork = -1;
  lapackf77_sgelqf(&m, &n, a, &m, tau, tmp, &lwork, &info);
  lwork = (magma_int_t)MAGMA_D_REAL( tmp[0] );
  lwork = max( lwork, m*nb );
  cudaMallocManaged(&h_work,lwork*sizeof(float)); //mem.h_work
// Randomize the matrix a and copy a -> r
  lapackf77_slarnv( &ione, ISEED, &n2, a );
  lapackf77_slacpy(MagmaFullStr,&m,&n,a,&m,r,&m );
// MAGMA
  gpu_time = magma_sync_wtime(NULL);
// LQ factorization for a real matrix r=L*Q using Magma
// L - lower triangular, Q - orthogonal

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

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("MAGMA time: %7.3f sec.\n",gpu_time);  // print Magma
// LAPACK                                                   time
  cpu_time=magma_wtime();
// LQ factorization for a real matrix a=L*Q using Lapack
  lapackf77_sgelqf(&m,&n,a,&m,tau,h_work,&lwork,&info);
  cpu_time=magma_wtime()-cpu_time;
  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
  magma_free(tau);                                // free memory
  magma_free(a);                                  // free memory
  magma_free(r);                                         // free memory
  magma_free(h_work);                                    // free memory
  magma_finalize( );                                   // finalize Magma
  magma_finalize( );                                   // finalize Magma
  return EXIT_SUCCESS;
}
//MAGMA time:   0.342 sec.
//LAPACK time:   1.818 sec.
//difference: 1.846170e-06
```

4.5.21 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 $H(1)\dots H(\min(m,n))$, 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 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_v2.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
  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  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,n);//optim. block size 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);//host memory for h_work
// Randomize the matrix a and copy a -> r
  lapackf77_dlarnv( &ione, ISEED, &n2, a );
  lapackf77_dlacpy(MagmaFullStr,&m,&n,a,&m,r,&m );
// MAGMA
  gpu_time = magma_sync_wtime(NULL);
// LQ factorization for a real matrix r=L*Q using Magma
// L - lower triangular, Q - orthogonal

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

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("MAGMA time: %7.3f sec.\n",gpu_time);  // print Magma
// LAPACK                                                   time
  cpu_time=magma_wtime();
// LQ factorization for a real matrix a=L*Q on the host
  lapackf77_dgelqf(&m,&n,a,&m,tau,h_work,&lwork,&info);
  cpu_time=magma_wtime()-cpu_time;
  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
  magma_finalize( );                        // finalize Magma
  return EXIT_SUCCESS;
}
//MAGMA time:   0.715 sec.
//LAPACK time:   3.240 sec.
//difference: 3.434041e-15
```

4.5.22 magma_dgelqf - unified memory version

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.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
  double  gpu_time, cpu_time;
  magma_int_t m = 4096, n = 4096, n2=m*n;
  double *a, *r;                          // a, r - mxn matrices
  double *tau;   // scalars defining the elementary reflectors
  double *h_work, tmp[1];  // h_work - workspace; tmp -used in
                                              // workspace query
  magma_int_t  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,n);//optim. block size for dgeqrf
// prepare unified memory
  cudaMallocManaged(&tau,min_mn*sizeof(double)); //mem.for tau
  cudaMallocManaged(&a,n2*sizeof(double));//unif. memory for a
  cudaMallocManaged(&r,n2*sizeof(double));//unif. memory for r
// Get size for  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 );
  cudaMallocManaged(&h_work,lwork*sizeof(double));//mem.h_work
// Randomize the matrix a and copy a -> r
  lapackf77_dlarnv( &ione, ISEED, &n2, a );
  lapackf77_dlacpy(MagmaFullStr,&m,&n,a,&m,r,&m );
// MAGMA
  gpu_time = magma_sync_wtime(NULL);
// LQ factorization for a real matrix r=L*Q using Magma
// L - lower triangular, Q - orthogonal

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

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("MAGMA time: %7.3f sec.\n",gpu_time);  // print Magma
// LAPACK                                                  time
  cpu_time=magma_wtime();
// LQ factorization for a real matrix a=L*Q using Lapack
  lapackf77_dgelqf(&m,&n,a,&m,tau,h_work,&lwork,&info);
  cpu_time=magma_wtime()-cpu_time;
  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
  magma_free(tau);                             // free memory
  magma_free(a);                                       // free memory
  magma_free(r);                                       // free memory
  magma_free(h_work);                                  // free memory
  magma_finalize( );                                // finalize Magma
  magma_finalize( );                                // finalize Magma
  return EXIT_SUCCESS;
}
//MAGMA time:   0.728 sec.
//LAPACK time:   2.827 sec.
//difference: 3.434041e-15
```

4.5.23 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_v2.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_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  double  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  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,n); //optim.block size 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,m*n);            // 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);//host memory for h_work
// Randomize the matrix a and copy a -> r
  lapackf77_slarnv( &ione, ISEED, &n2, a );
  lapackf77_slacpy(MagmaFullStr,&m,&n,a,&m,r,&m );
// MAGMA
  magma_ssetmatrix( m, n, r, m, d_a,m,queue);// copy r -> d_a
  gpu_time = magma_sync_wtime(NULL);
// 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);

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("MAGMA time: %7.3f sec.\n",gpu_time);  // print Magma
// LAPACK                                                   time
  cpu_time=magma_wtime();
// LQ factorization for a real matrix a=L*Q on the host
  lapackf77_sgelqf(&m,&n,a,&m,tau,h_work,&lwork,&info);
  cpu_time=magma_wtime()-cpu_time;
  printf("LAPACK time: %7.3f sec.\n",cpu_time);// print Lapack
// difference                                               time
  magma_sgetmatrix( m, n, d_a, m, r, m,queue);
  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_queue_destroy(queue);                 // destroy queue
  magma_finalize( );                         // finalize Magma
  return EXIT_SUCCESS;
}
//MAGMA time:   0.067 sec.
//LAPACK time:   2.364 sec.
//difference: 1.846170e-06
```

4.5.24 magma_sgelqf_gpu - unified memory version

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.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_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  double  gpu_time, cpu_time;
  magma_int_t m = 4096, n = 4096, n2=m*n;
  float *a, *r;                           // a, r - mxn matrices
  float *h_work, tmp[1];   // h_work - workspace; tmp -used in
                                             // workspace query
  float *tau;    // scalars defining the elementary reflectors
  float *a1;       // a1 - mxn matrix used by Magma sgelqf_gpu
  magma_int_t  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,n); //optim.block size for sgeqrf
  cudaMallocManaged(&tau,min_mn*sizeof(float));//u.mem.for tau
  cudaMallocManaged(&a,n2*sizeof(float)); //unif. memory for a
  cudaMallocManaged(&r,n2*sizeof(float)); //unif. memory for r
  cudaMallocManaged(&a1,n2*sizeof(float)); //unif. mem. for a1
// Get size for  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 );
  cudaMallocManaged(&h_work,lwork*sizeof(float)); //mem.h_work
// Randomize the matrix a and copy a -> r
  lapackf77_slarnv( &ione, ISEED, &n2, a );
  lapackf77_slacpy(MagmaFullStr,&m,&n,a,&m,r,&m );      // a->r
// MAGMA
  magma_ssetmatrix( m, n, r, m, a1,m,queue);      // copy r->a1
  gpu_time = magma_sync_wtime(NULL);
// LQ factorization for a real matrix a1=L*Q using Magma
// L - lower triangular, Q - orthogonal

  magma_sgelqf_gpu(m,n,a1,m,tau,h_work,lwork,&info);

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("MAGMA time: %7.3f sec.\n",gpu_time);  // print Magma
// LAPACK                                                  time
```

```
  cpu_time=magma_wtime();
// LQ factorization for a real matrix a=L*Q using Lapack
  lapackf77_sgelqf(&m,&n,a,&m,tau,h_work,&lwork,&info);
  cpu_time=magma_wtime()-cpu_time;
  printf("LAPACK time: %7.3f sec.\n",cpu_time);// print Lapack
// difference                                            time
  matnorm = lapackf77_slange("f", &m, &n, a, &m, work);// norm
  blasf77_saxpy(&n2, &mzone, a, &ione, a1, &ione);   // a - a1
  printf("difference: %e\n",              // ||a-a1||_F/||a||_F
  lapackf77_slange("f", &m, &n, a1, &m, work) / matnorm);
// Free emory
  magma_free(tau);                                // free memory
  magma_free(a);                                  // free memory
  magma_free(r);                                  // free memory
  magma_free(h_work);                             // free memory
  magma_free(a1);                                 // free memory
  magma_queue_destroy(queue);                    // destroy queue
  magma_finalize( );                            // finalize Magma
  return EXIT_SUCCESS;
}

//MAGMA time:   0.068 sec.
//LAPACK time:   1.810 sec.
//difference: 1.846170e-06
```

4.5.25 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_v2.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_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  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  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,n); //optim.block size 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,m*n);             // 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);  // host mem.for h_work
// Randomize the matrix a and copy a -> r
  lapackf77_dlarnv( &ione, ISEED, &n2, a );
  lapackf77_dlacpy(MagmaFullStr,&m,&n,a,&m,r,&m );
// MAGMA
  magma_dsetmatrix( m, n, r, m, d_a,m,queue); // copy r -> d_a
  gpu_time = magma_sync_wtime(NULL);
// 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);

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("MAGMA time: %7.3f sec.\n",gpu_time);  // print Magma
// LAPACK                                                  time
  cpu_time=magma_wtime();
// LQ factorization for a real matrix a=L*Q on the host
  lapackf77_dgelqf(&m,&n,a,&m,tau,h_work,&lwork,&info);
  cpu_time=magma_wtime()-cpu_time;
  printf("LAPACK time: %7.3f sec.\n",cpu_time);// print Lapack
// difference                                              time
  magma_dgetmatrix( m, n, d_a, m, r, m, queue);
  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_queue_destroy(queue);                        // destroy queue
  magma_finalize( );                               // finalize Magma
  return EXIT_SUCCESS;
}
//MAGMA time:   0.466 sec.
//LAPACK time:   3.197 sec.
//difference: 3.434041e-15
```

4.5.26 magma_dgelqf_gpu - unified memory version

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.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_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  double  gpu_time, cpu_time;
  magma_int_t m = 4096, n = 4096, n2=m*n;
  double *a, *r;                               // a, r - mxn matrices
  double *h_work, tmp[1];  // h_work - workspace; tmp -used in
                                                 // workspace query
  double *tau;   // scalars defining the elementary reflectors
  double *a1;      // a1 - mxn matrix used by Magma dgelqf_gpu
  magma_int_t  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,n); //optim.block size for dgeqrf
  cudaMallocManaged(&tau,min_mn*sizeof(double)); //mem.for tau
  cudaMallocManaged(&a,n2*sizeof(double));//unif. memory for a
  cudaMallocManaged(&r,n2*sizeof(double));//unif. memory for r
  cudaMallocManaged(&a1,n2*sizeof(double));//un. memory for a1
// Get size for  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 );
  cudaMallocManaged(&h_work,lwork*sizeof(double)); //unif.mem
// Randomize the matrix a and copy a -> r        // for h_work
  lapackf77_dlarnv( &ione, ISEED, &n2, a );
  lapackf77_dlacpy(MagmaFullStr,&m,&n,a,&m,r,&m );      // a->r
// MAGMA
  magma_dsetmatrix( m, n, r, m, a1,m,queue);      // copy r->a1
  gpu_time = magma_sync_wtime(NULL);
// LQ factorization for a real matrix a1=L*Q using Magma
// L - lower triangular, Q - orthogonal

  magma_dgelqf_gpu(m,n,a1,m,tau,h_work,lwork,&info);

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("MAGMA time: %7.3f sec.\n",gpu_time);  // print Magma
// LAPACK                                                   time
  cpu_time=magma_wtime();
// LQ factorization for a real matrix a=L*Q using Lapack
  lapackf77_dgelqf(&m,&n,a,&m,tau,h_work,&lwork,&info);
  cpu_time=magma_wtime()-cpu_time;
  printf("LAPACK time: %7.3f sec.\n",cpu_time);// print Lapack
// difference                                               time
  matnorm = lapackf77_dlange("f", &m, &n, a, &m, work);// norm
  blasf77_daxpy(&n2, &mzone, a, &ione, a1, &ione);    // a - a1
  printf("difference: %e\n",               // ||a-a1||_F/||a||_F
  lapackf77_dlange("f", &m, &n, a1, &m, work) / matnorm);
// Free emory
  magma_free(tau);                                // free memory
  magma_free(a);                                  // free memory
  magma_free(r);                                  // free memory
  magma_free(h_work);                             // free memory
  magma_free(a1);                                 // free memory
  magma_queue_destroy(queue);                    // destroy queue
  magma_finalize( );                            // finalize Magma
  return EXIT_SUCCESS;
}
//MAGMA time:   0.472 sec.
//LAPACK time:   2.832 sec.
//difference: 3.434041e-15
```

4.5.27 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 *jpvt*. On exit if *jpvt*$(j) = k$, 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_v2.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
  double  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;                                     // workspace
  float *tau;    // scalars defining the elementary reflectors
  magma_int_t *jpvt;                       // pivoting information
  magma_int_t  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
  nb = magma_get_sgeqp3_nb(m,n);          // optimal block size
  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
// Randomize the matrix a and copy a -> r
  lapackf77_slarnv(&ione,ISEED,&n2,a);
  lapackf77_slacpy(MagmaFullStr,&m,&n,a,&m,r,&m);      // a->r
// LAPACK
  for (j = 0; j < n; j++)
    jpvt[j] = 0;
  cpu_time=magma_wtime();
// QR decomposition with column pivoting, Lapack version
  lapackf77_sgeqp3(&m,&n,r,&m,jpvt,tau,h_work,&lwork,&info);
  cpu_time=magma_wtime()-cpu_time;
  printf("LAPACK time: %7.3f sec.\n",cpu_time); // Lapack time
// MAGMA
  lapackf77_slacpy(MagmaFullStr,&m,&n,a,&m,r,&m);      // a->r
  for (j = 0; j < n; j++)
    jpvt[j] = 0 ;
  gpu_time = magma_sync_wtime(NULL);
// QR decomposition with column pivoting, Magma version

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

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("MAGMA time: %7.3f sec.\n",gpu_time);    // Magma time
// 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:   6.402 sec.
//MAGMA time:   1.568 sec.
```

4.5.28 magma_sgeqp3 - unified memory version

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.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
  double  gpu_time, cpu_time;
  magma_int_t m = 4096, n = 4096, n2=m*n;
  float *a, *r;                             // a, r - mxn matrices
  float *h_work;                                       // workspace
  float *tau;    // scalars defining the elementary reflectors
  magma_int_t *jpvt;                      // pivoting information
  magma_int_t  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
  nb = magma_get_sgeqp3_nb(m,n);          // optimal block size
  cudaMallocManaged(&jpvt,n*sizeof(magma_int_t));// m.for jpvt
  cudaMallocManaged(&tau,min_mn*sizeof(float));//u.mem.for tau
  cudaMallocManaged(&a,n2*sizeof(float)); //unif. memory for a
  cudaMallocManaged(&r,n2*sizeof(float)); //unif. memory for r
  lwork = 2*n + ( n+1 )*nb;
  lwork = max(lwork, m * n + n);

  cudaMallocManaged(&h_work,lwork*sizeof(float)); //mem.h_work
// Randomize the matrix a and copy a -> r
```

```
  lapackf77_slarnv(&ione,ISEED,&n2,a);
  lapackf77_slacpy(MagmaFullStr,&m,&n,a,&m,r,&m);      // a->r
// LAPACK
  for (j = 0; j < n; j++)
    jpvt[j] = 0;
  cpu_time=magma_wtime();
// QR decomposition with column pivoting, Lapack version
  lapackf77_sgeqp3(&m,&n,r,&m,jpvt,tau,h_work,&lwork,&info);
  cpu_time=magma_wtime()-cpu_time;
  printf("LAPACK time: %7.3f sec.\n",cpu_time); // Lapack time
// MAGMA
  lapackf77_slacpy(MagmaFullStr,&m,&n,a,&m,r,&m);      // a->r
  for (j = 0; j < n; j++)
    jpvt[j] = 0 ;
  gpu_time = magma_sync_wtime(NULL);
// QR decomposition with column pivoting, Magma version

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

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("MAGMA time: %7.3f sec.\n",gpu_time);   // Magma time
// Free memory
  magma_free(jpvt);                                // free memory
  magma_free(tau);                                 // free memory
  magma_free(a);                                   // free memory
  magma_free(r);                                   // free memory
  magma_free(h_work);                              // free memory
  magma_finalize( );                             // finalize Magma
  return EXIT_SUCCESS;
}
```

4.5.29 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 *jpvt*. On exit if $jpvt(j) = k$, 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_v2.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
  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;                                     // workspace
  double *tau;   // scalars defining the elementary reflectors
  magma_int_t *jpvt;                        // pivoting information
  magma_int_t  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
  nb = magma_get_dgeqp3_nb(m,n);          // optimal block size
  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
// Randomize the matrix a and copy a -> r
  lapackf77_dlarnv(&ione,ISEED,&n2,a);
  lapackf77_dlacpy(MagmaFullStr,&m,&n,a,&m,r,&m);      // a->r
// LAPACK
  for (j = 0; j < n; j++)
    jpvt[j] = 0;
  cpu_time=magma_wtime();
// QR decomposition with column pivoting, Lapack version
  lapackf77_dgeqp3(&m,&n,r,&m,jpvt,tau,h_work,&lwork,&info);
  cpu_time=magma_wtime()-cpu_time;
  printf("LAPACK time: %7.3f sec.\n",cpu_time); // Lapack time
// MAGMA
  lapackf77_dlacpy(MagmaFullStr,&m,&n,a,&m,r,&m);      // a->r
  for (j = 0; j < n; j++)
    jpvt[j] = 0 ;
  gpu_time = magma_sync_wtime(NULL);
// QR decomposition with column pivoting, Magma version

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

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("MAGMA time: %7.3f sec.\n",gpu_time);   // Magma time
// 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:  14.135 sec.
//MAGMA time:   2.173 sec.
```

4.5.30 magma_dgeqp3 - unified memory version

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.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
  double  gpu_time, cpu_time;
  magma_int_t m = 4096, n = 4096, n2=m*n;
  double *a, *r;                              // a, r - mxn matrices
  double *h_work;                                        // workspace
  double *tau;   // scalars defining the elementary reflectors
  magma_int_t *jpvt;                         // pivoting information
  magma_int_t  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
  nb = magma_get_dgeqp3_nb(m,n);          // optimal block size
  cudaMallocManaged(&jpvt,n*sizeof(magma_int_t));// m.for jpvt
  cudaMallocManaged(&tau,min_mn*sizeof(double)); //mem.for tau
  cudaMallocManaged(&a,n2*sizeof(double)); //unif.memory for a
  cudaMallocManaged(&r,n2*sizeof(double)); //unif.memory for r
  lwork = 2*n + ( n+1 )*nb;
  lwork = max(lwork, m * n + n);

  cudaMallocManaged(&h_work,lwork*sizeof(double));//mem.h_work
// Randomize the matrix a and copy a -> r
  lapackf77_dlarnv(&ione,ISEED,&n2,a);
  lapackf77_dlacpy(MagmaFullStr,&m,&n,a,&m,r,&m);       // a->r
// LAPACK
  for (j = 0; j < n; j++)
    jpvt[j] = 0;
  cpu_time=magma_wtime();
// QR decomposition with column pivoting, Lapack version
  lapackf77_dgeqp3(&m,&n,r,&m,jpvt,tau,h_work,&lwork,&info);
  cpu_time=magma_wtime()-cpu_time;
  printf("LAPACK time: %7.3f sec.\n",cpu_time); // Lapack time
// MAGMA
  lapackf77_dlacpy(MagmaFullStr,&m,&n,a,&m,r,&m);      // a->r
  for (j = 0; j < n; j++)
    jpvt[j] = 0 ;
  gpu_time = magma_sync_wtime(NULL);
// QR decomposition with column pivoting, Magma version

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

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("MAGMA time: %7.3f sec.\n",gpu_time);   // Magma time
// Free memory
  magma_free(jpvt);                                 // free memory
  magma_free(tau);                                  // free memory
  magma_free(a);                                    // free memory
  magma_free(r);                                    // free memory
  magma_free(h_work);                               // free memory
  magma_finalize( );                              // finalize Magma
  return EXIT_SUCCESS;
}
//LAPACK time:  14.135 sec.
//MAGMA time:    2.253 sec.
```

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` or `MagmaVec` 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_v2.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, error;     // wi1,wi2 - imaginary parts of
                                                // eigenvalues
  magma_int_t ione = 1, i, j, info, nb;
  float mione = -1.0f, *h_work;            // h_work - workspace
  magma_int_t incr = 1, inci = 1, lwork;// lwork -worksp. size
  nb = magma_get_sgehrd_nb(n);// optimal block size 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,
// Magma version, left eigenvectors not computed,
// right eigenvectors are computed

  magma_sgeev(MagmaNoVec,MagmaVec,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_sgeev - unified memory version, small matrix

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.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
  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 = -1.0 , error, *h_work;    // h_work - workspace
  magma_int_t incr = 1, inci = 1, lwork;// lwork -worksp. size
  nb = magma_get_sgehrd_nb(n);// optimal block size for sgehrd
  float work[1];             // used in difference computations
  lwork = n*(2+nb);
  lwork = max(lwork,n*(5+2*n));
  cudaMallocManaged(&wr1,n*sizeof(float));//unified memory for
  cudaMallocManaged(&wr2,n*sizeof(float)); //real parts of eig
  cudaMallocManaged(&wi1,n*sizeof(float));//unified memory for
  cudaMallocManaged(&wi2,n*sizeof(float)); //imag.parts of eig
  cudaMallocManaged(&a,n2*sizeof(float)); //unif. memory for a
  cudaMallocManaged(&r,n2*sizeof(float)); //unif. memory for r
  cudaMallocManaged(&VL,n2*sizeof(float));//u.mem.for left and
  cudaMallocManaged(&VR,n2*sizeof(float));  //right eigenvect.
  cudaMallocManaged(&h_work,lwork*sizeof(float));
// 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,
// Magma version, left eigenvectors not computed,
// right eigenvectors are computed

  magma_sgeev(MagmaNoVec,MagmaVec,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);
  magma_free(wr1);                                // free memory
  magma_free(wr2);                                // free memory
  magma_free(wi1);                                  // free memory
  magma_free(wi2);                                  // free memory
  magma_free(a);                                    // free memory
  magma_free(r);                                    // free memory
  magma_free(VL);                                   // free memory
  magma_free(VR);                                   // free memory
  magma_free(h_work);                               // free 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_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` or `MagmaVec` 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_v2.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 block size 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,
// Magma version, left eigenvectors not computed,
// right eigenvectors are computed

  magma_dgeev(MagmaNoVec,MagmaVec,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
```

4.6.4 magma_dgeev - unified memory version, small matrix

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

```
#include "magma_v2.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
  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 block size for dgehrd
  double work[1];            // used in difference computations
  lwork = n*(2+nb);
  lwork = max(lwork,n*(5+2*n));
  cudaMallocManaged(&wr1,n*sizeof(double)); //unified mem. for
  cudaMallocManaged(&wr2,n*sizeof(double));//real parts of eig
  cudaMallocManaged(&wi1,n*sizeof(double)); //unified mem. for
  cudaMallocManaged(&wi2,n*sizeof(double));//imag.parts of eig
  cudaMallocManaged(&a,n2*sizeof(double));//unif. memory for a
  cudaMallocManaged(&r,n2*sizeof(double));//unif. memory for r
  cudaMallocManaged(&VL,n2*sizeof(double)); //mem.for left and
  cudaMallocManaged(&VR,n2*sizeof(double)); //right eigenvect.
  cudaMallocManaged(&h_work,lwork*sizeof(double));
// 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,
// Magma version, left eigenvectors not computed,
// right eigenvectors are computed

  magma_dgeev(MagmaNoVec,MagmaVec,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);
  magma_free(wr1);                                  // free memory
  magma_free(wr2);                                  // free memory
  magma_free(wi1);                                  // free memory
  magma_free(wi2);                                  // free memory
  magma_free(a);                                    // free memory
  magma_free(r);                                    // free memory
  magma_free(VL);                                   // free memory
  magma_free(VR);                                   // free memory
  magma_free(h_work);                               // free 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
```

4.6.5 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` or `MagmaVec` 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_v2.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=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 eigenvals
  float  gpu_time, cpu_time, *h_work;     // h_work - workspace
  magma_int_t ione=1,info,nb,lwork;     // lwork - worksp. size
  magma_int_t ISEED[4] = {0,0,0,1};                     // seed
  nb = magma_get_sgehrd_nb(n);// optimal block size 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
// Randomize the matrix a and copy a -> r
  lapackf77_slarnv(&ione,ISEED,&n2,a);
  lapackf77_slacpy(MagmaFullStr,&n,&n,a,&n,r,&n);
// MAGMA
  gpu_time = magma_sync_wtime(NULL);
// compute the eigenvalues of a general, real  nxn matrix,
// Magma version, left and right eigenvectors not computed

  magma_sgeev(MagmaNoVec,MagmaNoVec,n,r,n,wr1,wi1,VL,n,VR,n,
                           h_work,lwork,&info);

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("sgeev gpu time: %7.5f sec.\n",gpu_time);    // Magma
// LAPACK                                               // time
  cpu_time=magma_wtime();
// 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);
  cpu_time=magma_wtime()-cpu_time;
  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: 46.21376 sec.
//sgeev cpu time: 157.79790 sec.
```

4.6.6 magma_sgeev - unified memory version, big matrix

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.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=8192, n2=n*n;
  float *a, *r;                          // a, r - nxn matrices
  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
  double  gpu_time, cpu_time;
  float *h_work;                        // h_work - workspace
  magma_int_t ione=1,info,nb,lwork;     // lwork - worksp. size
  magma_int_t ISEED[4] = {0,0,0,1};                     // seed
  nb = magma_get_sgehrd_nb(n);// optimal block size for sgehrd
  lwork = n*(2+nb);
  lwork = max(lwork, n*(5+2*n));
  cudaMallocManaged(&wr1,n*sizeof(float));//unified memory for
  cudaMallocManaged(&wr2,n*sizeof(float)); //real parts of eig
  cudaMallocManaged(&wi1,n*sizeof(float));//unified memory for
  cudaMallocManaged(&wi2,n*sizeof(float)); //imag.parts of eig
  cudaMallocManaged(&a,n2*sizeof(float)); //unif. memory for a
  cudaMallocManaged(&r,n2*sizeof(float)); //unif. memory for r
  cudaMallocManaged(&VL,n2*sizeof(float));//u.mem.for left and
  cudaMallocManaged(&VR,n2*sizeof(float));  //right eigenvect.
  cudaMallocManaged(&h_work,lwork*sizeof(float));//m.f. h_work
// Randomize the matrix a and copy a -> r
  lapackf77_slarnv(&ione,ISEED,&n2,a);
  lapackf77_slacpy(MagmaFullStr,&n,&n,a,&n,r,&n);
// MAGMA
  gpu_time = magma_sync_wtime(NULL);
// compute the eigenvalues of a general, real  nxn matrix,
// Magma version, left and right eigenvectors not computed

  magma_sgeev(MagmaNoVec,MagmaNoVec,n,r,n,wr1,wi1,VL,n,VR,n,
                      h_work,lwork,&info);

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("sgeev gpu time: %7.5f sec.\n",gpu_time);    // Magma
// LAPACK                                               // time
  cpu_time=magma_wtime();
// 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);
  cpu_time=magma_wtime()-cpu_time;
  printf("sgeev cpu time: %7.5f sec.\n",cpu_time);   // Lapack
  magma_free(wr1);                                      // time
  magma_free(wr2);                               // free memory
  magma_free(wi1);                               // free memory
  magma_free(wi2);                               // free memory
  magma_free(a);                                 // free memory
  magma_free(r);                                 // free memory
  magma_free(VL);                                // free memory
  magma_free(VR);                                // free memory
  magma_free(h_work);                            // free memory
  magma_finalize( );                           // finalize Magma
  return EXIT_SUCCESS;
}
//sgeev gpu time: 40.60117 sec.
//sgeev cpu time: 108.51452 sec.
```

4.6.7 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 or MagmaVec 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_v2.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=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,info,nb,lwork;  // lwork - workspace size
  magma_int_t ISEED[4] = {0,0,0,1};                     // seed
  nb = magma_get_dgehrd_nb(n);// optimal block size 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
// Randomize the matrix a and copy a -> r
  lapackf77_dlarnv(&ione,ISEED,&n2,a);
  lapackf77_dlacpy(MagmaFullStr,&n,&n,a,&n,r,&n);
// MAGMA
  gpu_time = magma_sync_wtime(NULL);
// compute the eigenvalues of a general, real  nxn matrix,
// Magma version, left and right eigenvectors not computed

  magma_dgeev(MagmaNoVec,MagmaNoVec,n,r,n,wr1,wi1,VL,n,VR,n,
                      h_work,lwork,&info);

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("dgeev gpu time: %7.5f sec.\n",gpu_time);    // Magma
// LAPACK                                                // time
  cpu_time=magma_wtime();
// 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);
  cpu_time=magma_wtime()-cpu_time;
  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:  95.42350 sec.
//dgeev cpu time: 211.23290 sec.
```

4.6.8 magma_dgeev - unified memory version, big matrix

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.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=8192, n2=n*n;
  double *a, *r;                            // a, r - nxn matrices
  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,info,nb,lwork;    // lwork - worksp. size
  magma_int_t ISEED[4] = {0,0,0,1};                     // seed
  nb = magma_get_dgehrd_nb(n);// optimal block size for dgehrd
  lwork = n*(2+nb);
  lwork = max(lwork, n*(5+2*n));
  cudaMallocManaged(&wr1,n*sizeof(double)); //unified mem. for
  cudaMallocManaged(&wr2,n*sizeof(double));//real parts of eig
  cudaMallocManaged(&wi1,n*sizeof(double)); //unified mem. for
  cudaMallocManaged(&wi2,n*sizeof(double));//imag.parts of eig
  cudaMallocManaged(&a,n2*sizeof(double));//unif. memory for a
  cudaMallocManaged(&r,n2*sizeof(double));//unif. memory for r
  cudaMallocManaged(&VL,n2*sizeof(double)); //mem.for left and
  cudaMallocManaged(&VR,n2*sizeof(double)); //right eigenvect.
  cudaMallocManaged(&h_work,lwork*sizeof(double));
// Randomize the matrix a and copy a -> r
  lapackf77_dlarnv(&ione,ISEED,&n2,a);
  lapackf77_dlacpy(MagmaFullStr,&n,&n,a,&n,r,&n);
// MAGMA
```

```
  gpu_time = magma_sync_wtime(NULL);
// compute the eigenvalues of a general, real  nxn matrix,
// Magma version, left and right eigenvectors not computed

  magma_dgeev(MagmaNoVec,MagmaNoVec,n,r,n,wr1,wi1,VL,n,VR,n,
                          h_work,lwork,&info);

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("dgeev gpu time: %7.5f sec.\n",gpu_time);    // Magma
// LAPACK                                              // time
  cpu_time=magma_wtime();
// 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);
  cpu_time=magma_wtime()-cpu_time;
  printf("dgeev cpu time: %7.5f sec.\n",cpu_time);   // Lapack
  magma_free(wr1);                                     // time
  magma_free(wr2);                               // free memory
  magma_free(wi1);                               // free memory
  magma_free(wi2);                               // free memory
  magma_free(a);                                 // free memory
  magma_free(r);                                 // free memory
  magma_free(VL);                                // free memory
  magma_free(VR);                                // free memory
  magma_free(h_work);                            // free memory
  magma_finalize( );                           // finalize Magma
  return EXIT_SUCCESS;
}
//dgeev gpu time: 62.50911 sec.
//dgeev cpu time: 185.40615 sec.
```

4.6.9 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 $H(ilo)\dots H(ihi)$, 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_v2.h"
#include "magma_lapack.h"
int main( int argc, char** argv)
{
  magma_init();                              // initialize Magma
  double  gpu_time, cpu_time;
  magma_int_t n=2048, 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  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
// Randomize the matrix a and copy a -> r, a -> r1
  lapackf77_slarnv( &ione, ISEED, &n2, a );
  lapackf77_slacpy(MagmaFullStr,&n,&n,a,&n,r,&n);
  lapackf77_slacpy(MagmaFullStr,&n,&n,a,&n,r1,&n);
// MAGMA
  gpu_time = magma_sync_wtime(NULL);
// 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);

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  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
  cpu_time=magma_wtime();
// 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);
  cpu_time=magma_wtime()-cpu_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 device memory
  magma_finalize( );                            // finalize Magma
  return EXIT_SUCCESS;
}
//Magma time:   0.365 sec.
//upper left corner of the Hessenberg form:
//[
//    0.1206  -19.4276 -11.6704    0.5872   -0.0777
//  -26.2667 765.4211 444.0294   -6.4941    0.5035
//    0.      444.5269 258.5998   -4.0942    0.2565
//    0.        0.     -15.2374    0.3507    0.0222
//    0.        0.        0.     -13.0577   -0.1760
//];
//Lapack time:   0.916 sec.
//max difference: 1.018047e-03
```

4.6.10 magma_sgehrd - unified memory version

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.h"
#include "magma_lapack.h"
int main( int argc, char** argv)
{
  magma_init();                               // initialize Magma
  double  gpu_time, cpu_time;
  magma_int_t n=2048, n2=n*n;
  float *a, *r, *r1;                    // a,r,r1 - nxn matrices
  float *tau;    // scalars defining the elementary reflectors
  float *h_work;                                  // workspace
  magma_int_t  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;
  cudaMallocManaged(&tau,n*sizeof(float)); //unif. mem.for tau
  cudaMallocManaged(&a,n2*sizeof(float)); //unif. memory for a
  cudaMallocManaged(&r,n2*sizeof(float)); //unif. memory for r
  cudaMallocManaged(&r1,n2*sizeof(float));//unif.memory for r1
  cudaMallocManaged(&h_work,lwork*sizeof(float)); //m.f.h_work
  cudaMallocManaged(&dT,nb*n*sizeof(float));//unif. mem.for dT
// Randomize the matrix a and copy a -> r, a -> r1
  lapackf77_slarnv( &ione, ISEED, &n2, a );
  lapackf77_slacpy(MagmaFullStr,&n,&n,a,&n,r,&n);
  lapackf77_slacpy(MagmaFullStr,&n,&n,a,&n,r1,&n);
// MAGMA
  gpu_time = magma_sync_wtime(NULL);
// 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);

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  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
  cpu_time=magma_wtime();
// 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);
  cpu_time=magma_wtime()-cpu_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));
  magma_free(a);                                    // free memory
  magma_free(tau);                                  // free memory
  magma_free(h_work);                               // free memory
  magma_free(r);                                    // free memory
  magma_free(r1);                                   // free memory
  magma_free(dT);                                   // free memory
  magma_finalize( );                              // finalize Magma
  return EXIT_SUCCESS;
}
//Magma time:   0.403 sec.
//upper left corner of the Hessenberg form:
//[
//   0.1206 -19.4276 -11.6704   0.5872  -0.0777
// -26.2667 765.4211 444.0294  -6.4941   0.5035
//   0.     444.5269 258.5998  -4.0942   0.2565
//   0.       0.     -15.2374   0.3507   0.0222
//   0.       0.       0.     -13.0577  -0.1760
//];
//Lapack time:   0.644 sec.
//max difference: 1.018047e-03
```

4.6.11 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_v2.h"
#include "magma_lapack.h"
int main( int argc, char** argv)
{
  magma_init();                                 // initialize Magma
  double  gpu_time, cpu_time;
  magma_int_t n=2048, 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  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
// Randomize the matrix a and copy a -> r, a -> r1
  lapackf77_dlarnv( &ione, ISEED, &n2, a );
  lapackf77_dlacpy(MagmaFullStr,&n,&n,a,&n,r,&n);
  lapackf77_dlacpy(MagmaFullStr,&n,&n,a,&n,r1,&n);
// MAGMA
  gpu_time = magma_sync_wtime(NULL);
// 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);

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  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
  cpu_time=magma_wtime();
// 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);
  cpu_time=magma_wtime()-cpu_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 device memory
  magma_finalize( );                              // finalize Magma
  return EXIT_SUCCESS;
}
//Magma time:   0.525 sec.
//upper left corner of the Hessenberg form:
//[
//    0.1206 -19.4276 -11.6704   0.5872  -0.0777
//  -26.2667 765.4211 444.0295  -6.4941   0.5035
//    0.     444.5269 258.5999  -4.0943   0.2565
//    0.       0.     -15.2374   0.3507   0.0222
//    0.       0.       0.     -13.0577  -0.1760
//];
//Lapack time:   2.067 sec.
//max difference: 1.444213e-12
```

4.6.12 magma_dgehrd - unified memory version

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.h"
#include "magma_lapack.h"
int main( int argc, char** argv)
{
  magma_init();                                 // initialize Magma
  double  gpu_time, cpu_time;
  magma_int_t n=2048, n2=n*n;
  double *a, *r, *r1;                     // a,r,r1 - nxn matrices
  double *tau;   // scalars defining the elementary reflectors
  double *h_work;                                     // workspace
  magma_int_t  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;
  cudaMallocManaged(&tau,n*sizeof(double));//unif. mem.for tau
  cudaMallocManaged(&a,n2*sizeof(double));//unif. memory for a
  cudaMallocManaged(&r,n2*sizeof(double));//unif. memory for r
  cudaMallocManaged(&r1,n2*sizeof(double));//unif. mem. for r1
  cudaMallocManaged(&h_work,lwork*sizeof(double));//m.f.h_work
  cudaMallocManaged(&dT,nb*n*sizeof(double));//unif.mem.for dT
// Randomize the matrix a and copy a -> r, a -> r1
  lapackf77_dlarnv( &ione, ISEED, &n2, a );
  lapackf77_dlacpy(MagmaFullStr,&n,&n,a,&n,r,&n);
  lapackf77_dlacpy(MagmaFullStr,&n,&n,a,&n,r1,&n);
// MAGMA
  gpu_time = magma_sync_wtime(NULL);
// 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);

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  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
  cpu_time=magma_wtime();
// 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);
  cpu_time=magma_wtime()-cpu_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));
  magma_free(a);                                   // free memory
  magma_free(tau);                                 // free memory
  magma_free(h_work);                              // free memory
  magma_free(r);                                   // free memory
  magma_free(r1);                                  // free memory
  magma_free(dT);                                  // free memory
  magma_finalize( );                             // finalize Magma
  return EXIT_SUCCESS;
}
//Magma time:   0.572 sec.
```

```
//upper left corner of the Hessenberg form:
//[
//    0.1206  -19.4276  -11.6704    0.5872   -0.0777
//  -26.2667  765.4211  444.0295   -6.4941    0.5035
//    0.      444.5269  258.5999   -4.0943    0.2565
//    0.        0.      -15.2374    0.3507    0.0222
//    0.        0.        0.      -13.0577   -0.1760
//];
//Lapack time:   1.753 sec.
//max difference: 1.444213e-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` or `MagmaNoVec` 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`) or upper (`MagmaUpper`) 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_v2.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;
  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(MagmaVec,MagmaLower,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
// 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_ssyevd - unified memory version

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.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
  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 = -1.0;
  magma_int_t incr = 1;
  cudaMallocManaged(&w1,n*sizeof(float));//unified memory for
  cudaMallocManaged(&w2,n*sizeof(float));       //eigenvalues
  cudaMallocManaged(&a,n2*sizeof(float));//unif. memory for a
  cudaMallocManaged(&r,n2*sizeof(float));//unif. memory for r
// Query for workspace sizes
  float aux_work[1];
  magma_int_t aux_iwork[1];
  magma_ssyevd(MagmaVec,MagmaLower,n,r,n,w1,aux_work,-1,
                             aux_iwork,-1,&info );
  lwork  = (magma_int_t) aux_work[0];
  liwork = aux_iwork[0];      // unified memory for workspace:
  cudaMallocManaged(&iwork,liwork*sizeof(magma_int_t));
  cudaMallocManaged(&h_work,lwork*sizeof(float));
// 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);
  magma_free(w1);                                // free memory
  magma_free(w2);                                // free memory
  magma_free(a);                                 // free memory
  magma_free(r);                                 // free memory
  magma_free(h_work);                            // free 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
```

4.7.3 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 or MagmaNoVec 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) or upper (MagmaUpper) 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_v2.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;
  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(MagmaVec,MagmaLower,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
// 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,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_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
//];
//difference in eigenvalues: 0.000000e+00
```

4.7.4 magma_dsyevd - unified memory version

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.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
  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;
  cudaMallocManaged(&w1,n*sizeof(double));//unified memory for
  cudaMallocManaged(&w2,n*sizeof(double));        //eigenvalues
  cudaMallocManaged(&a,n2*sizeof(double));//unif. memory for a
  cudaMallocManaged(&r,n2*sizeof(double));//unif. memory for r
// Query for workspace sizes
  double aux_work[1];
  magma_int_t aux_iwork[1];
  magma_dsyevd(MagmaVec,MagmaLower,n,r,n,w1,aux_work,-1,
                             aux_iwork,-1,&info );
  lwork  = (magma_int_t) aux_work[0];
  liwork = aux_iwork[0];      // unified memory for workspace:
  cudaMallocManaged(&iwork,liwork*sizeof(magma_int_t));
  cudaMallocManaged(&h_work,lwork*sizeof(double));
// 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,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_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);
  magma_free(w1);                                  // free memory
  magma_free(w2);                                  // free memory
  magma_free(a);                                   // free memory
  magma_free(r);                                   // free memory
  magma_free(h_work);                              // free 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
```

4.7.5 `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` or `MagmaNoVec` 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`) or upper (`MagmaUpper`) 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
  double  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, info;
  float mione = -1.0f;
  magma_int_t incr = 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(MagmaVec,MagmaLower,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
// Randomize the matrix a and copy a -> r
  lapackf77_slarnv(&ione,ISEED,&n2,a);
  lapackf77_slacpy(MagmaFullStr,&n,&n,a,&n,r,&n);
  gpu_time = magma_sync_wtime(NULL);
// 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);

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("ssyevd gpu time: %7.5f sec.\n",gpu_time);   // Magma
// Lapack version                                      // time
  cpu_time=magma_wtime();
  lapackf77_ssyevd("V","L",&n,a,&n,w2,h_work,&lwork,iwork,
                                              &liwork,&info);
  cpu_time=magma_wtime()-cpu_time;
  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: 5.58410 sec.
//ssyevd cpu time: 49.01886 sec.
//difference in eigenvalues: 9.765625e-04
```

4.7.6 magma_ssyevd - unified memory version, big matrix

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.h"
#include "magma_lapack.h"
int main( int argc, char** argv) {
  magma_init();                               // initialize Magma
  double  gpu_time, cpu_time;
  magma_int_t n=8192, n2=n*n;
  float  *a, *r;                           // a, r - nxn matrices
  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, info;
  float mione = -1.0f;
  magma_int_t incr = 1;
  magma_int_t ISEED[4] = {0,0,0,1};                      // seed
  cudaMallocManaged(&w1,n*sizeof(float)); //unified memory for
  cudaMallocManaged(&w2,n*sizeof(float));        //eigenvalues
  cudaMallocManaged(&a,n2*sizeof(float)); //unif. memory for a
  cudaMallocManaged(&r,n2*sizeof(float)); //unif. memory for r
// Query for workspace sizes
  float aux_work[1];
  magma_int_t aux_iwork[1];
  magma_ssyevd(MagmaVec,MagmaLower,n,r,n,w1,aux_work,-1,
                             aux_iwork,-1,&info );
  lwork  = (magma_int_t) aux_work[0];
  liwork = aux_iwork[0];        //unified memory for workspace:
  cudaMallocManaged(&iwork,liwork*sizeof(magma_int_t));
  cudaMallocManaged(&h_work,lwork*sizeof(float));
// Randomize the matrix a and copy a -> r
  lapackf77_slarnv(&ione,ISEED,&n2,a);
  lapackf77_slacpy(MagmaFullStr,&n,&n,a,&n,r,&n);
  gpu_time = magma_sync_wtime(NULL);
// 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);

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("ssyevd gpu time: %7.5f sec.\n",gpu_time);   // Magma
// Lapack version                                      // time
  cpu_time=magma_wtime();
  lapackf77_ssyevd("V","L",&n,a,&n,w2,h_work,&lwork,iwork,
                                            &liwork,&info);
  cpu_time=magma_wtime()-cpu_time;
  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);
  magma_free(w1);                                // free memory
  magma_free(w2);                                // free memory
  magma_free(a);                                 // free memory
  magma_free(r);                                 // free memory
  magma_free(h_work);                            // free memory
  magma_finalize();                           // finalize Magma
  return EXIT_SUCCESS;
}
//ssyevd gpu time: 5.77196 sec.
//ssyevd cpu time: 51.33320 sec.
//difference in eigenvalues: 9.765625e-04
```

4.7.7 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 or MagmaNoVec 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) or upper (MagmaUpper) 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_v2.h"
#include "magma_lapack.h"
int main( int argc, char** argv) {
  magma_init();                               // initialize Magma
  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, info;
  double mione = -1.0;
  magma_int_t incr = 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(MagmaVec,MagmaLower,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
// Randomize the matrix a and copy a -> r
  lapackf77_dlarnv(&ione,ISEED,&n2,a);
  lapackf77_dlacpy(MagmaFullStr,&n,&n,a,&n,r,&n);
  gpu_time = magma_sync_wtime(NULL);
// 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);

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("dsyevd gpu time: %7.5f sec.\n",gpu_time);   // Magma
// Lapack version                                       // time
  cpu_time=magma_wtime();
  lapackf77_dsyevd("V","L",&n,a,&n,w2,h_work,&lwork,iwork,
                                              &liwork,&info);
  cpu_time=magma_wtime()-cpu_time;
  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: 17.29120 sec.
//dsyevd cpu time: 91.15194 sec.
//difference in eigenvalues: 1.364242e-11
```

4.7.8 magma_dsyevd - unified memory version, big matrix

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.h"
#include "magma_lapack.h"
int main( int argc, char** argv) {
  magma_init();                              // initialize Magma
  double  gpu_time, cpu_time;
  magma_int_t n=8192, n2=n*n;
  double  *a, *r;                         // a, r - nxn matrices
  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, info;
  double mione = -1.0;
  magma_int_t incr = 1;
  magma_int_t ISEED[4] = {0,0,0,1};                        // seed
  cudaMallocManaged(&w1,n*sizeof(double));//unified memory for
  cudaMallocManaged(&w2,n*sizeof(double));       //eigenvalues
  cudaMallocManaged(&a,n2*sizeof(double));//unif. memory for a
  cudaMallocManaged(&r,n2*sizeof(double));//unif. memory for r
// Query for workspace sizes
  double aux_work[1];
  magma_int_t aux_iwork[1];
  magma_dsyevd(MagmaVec,MagmaLower,n,r,n,w1,aux_work,-1,
                             aux_iwork,-1,&info );
  lwork  = (magma_int_t) aux_work[0];
  liwork = aux_iwork[0];      // unified memory for workspace:
  cudaMallocManaged(&iwork,liwork*sizeof(magma_int_t));
  cudaMallocManaged(&h_work,lwork*sizeof(double));
// Randomize the matrix a and copy a -> r
  lapackf77_dlarnv(&ione,ISEED,&n2,a);
  lapackf77_dlacpy(MagmaFullStr,&n,&n,a,&n,r,&n);
  gpu_time = magma_sync_wtime(NULL);
// 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);

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("dsyevd gpu time: %7.5f sec.\n",gpu_time);   // Magma
// Lapack version                                      // time
  cpu_time=magma_wtime();
  lapackf77_dsyevd("V","L",&n,a,&n,w2,h_work,&lwork,iwork,
                                              &liwork,&info);
  cpu_time=magma_wtime()-cpu_time;
  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);
  magma_free(w1);                                // free memory
  magma_free(w2);                                // free memory
  magma_free(a);                                 // free memory
  magma_free(r);                                 // free memory
  magma_free(h_work);                            // free memory
  magma_finalize();                            // finalize Magma
  return EXIT_SUCCESS;
}
//dsyevd gpu time: 17.29073 sec.
//dsyevd cpu time: 96.53918 sec.
//difference in eigenvalues: 1.364242e-11
```

4.7.9 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` or `MagmaNoVec` 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`) or upper (`MagmaUpper`) 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_v2.h"
#include "magma_lapack.h"
int main( int argc, char** argv) {
  magma_init();                               // initialize Magma
  magma_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  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;
  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(MagmaVec,MagmaLower,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,queue);// 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,queue);// 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_queue_destroy(queue);                       // destroy queue
  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
```

4.7.10 magma_ssyevd_gpu - unified memory version, small matrix

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.h"
#include "magma_lapack.h"
int main( int argc, char** argv) {
  magma_init();                                  // initialize Magma
  magma_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  magma_int_t n=1024, n2=n*n;
  float  *a, *r;                              // a, r - nxn matrices
  float *a1; // nxn matrix, copy of a used in magma_ssyevd_gpu
  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 = -1.0;
  magma_int_t incr = 1;
  cudaMallocManaged(&w1,n*sizeof(float)); //unified memory for
  cudaMallocManaged(&w2,n*sizeof(float));        //eigenvalues
  cudaMallocManaged(&a,n2*sizeof(float)); //unif. memory for a
  cudaMallocManaged(&r,n2*sizeof(float)); //unif. memory for r
  cudaMallocManaged(&a1,n2*sizeof(float));//unif.memory for a1
// Query for workspace sizes
  float aux_work[1];
  magma_int_t aux_iwork[1];
  magma_ssyevd_gpu(MagmaVec,MagmaLower,n,a1,n,w1,r,n,aux_work,
                            -1, aux_iwork,-1,&info );
  lwork  = (magma_int_t) aux_work[0];
  liwork = aux_iwork[0];       // unified memory for workspace:
  cudaMallocManaged(&iwork,liwork*sizeof(magma_int_t));
  cudaMallocManaged(&h_work,lwork*sizeof(float));
// 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, a1, n,queue);  // copy a -> a1
// compute the eigenvalues  and eigenvectors for a symmetric,
// real nxn matrix; Magma version

  magma_ssyevd_gpu(MagmaVec,MagmaLower,n,a1,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_sprint(5,5,a1,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);
  magma_free(w1);                                 // free memory
  magma_free(w2);                                 // free memory
  magma_free(a);                                  // free memory
  magma_free(r);                                  // free memory
  magma_free(h_work);                             // free memory
  magma_free(a1);                                 // free memory
  magma_queue_destroy(queue);                    // destroy queue
  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
```

4.7.11 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` or `MagmaNoVec` 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`) or upper (`MagmaUpper`) 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_v2.h"
#include "magma_lapack.h"
int main( int argc, char** argv) {
  magma_init();                                  // initialize Magma
  magma_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  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;
  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(MagmaVec,MagmaLower,n,d_r,n,w1,r,n,aux_work,
           -1, aux_iwork,-1,&info );        // workspace query
  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,queue);// 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,queue);// 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_queue_destroy(queue);                       // destroy queue
  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
```

4.7.12 magma_dsyevd_gpu - unified memory version, small matrix

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.h"
#include "magma_lapack.h"
int main( int argc, char** argv) {
  magma_init();                               // initialize Magma
  magma_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  magma_int_t n=1024, n2=n*n;
  double  *a, *r;                           // a, r - nxn matrices
  double *a1;// nxn matrix, copy of a used in magma_dsyevd_gpu
  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;
  cudaMallocManaged(&w1,n*sizeof(double));//unified memory for
  cudaMallocManaged(&w2,n*sizeof(double));       //eigenvalues
  cudaMallocManaged(&a,n2*sizeof(double));//unif. memory for a
  cudaMallocManaged(&r,n2*sizeof(double));//unif. memory for r
  cudaMallocManaged(&a1,n2*sizeof(double));//uni.memory for a1
// Query for workspace sizes
  double aux_work[1];
  magma_int_t aux_iwork[1];
  magma_dsyevd_gpu(MagmaVec,MagmaLower,n,a1,n,w1,r,n,aux_work,
                              -1, aux_iwork,-1,&info );
  lwork  = (magma_int_t) aux_work[0];
  liwork = aux_iwork[0];       // unified memory for workspace:
  cudaMallocManaged(&iwork,liwork*sizeof(magma_int_t));
  cudaMallocManaged(&h_work,lwork*sizeof(double));
// 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, a1, n,queue);  // copy a -> a1
// compute the eigenvalues  and eigenvectors for a symmetric,
// real nxn matrix; Magma version

  magma_dsyevd_gpu(MagmaVec,MagmaLower,n,a1,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_dprint(5,5,a1,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);
  magma_free(w1);                                 // free memory
  magma_free(w2);                                 // free memory
  magma_free(a);                                  // free memory
  magma_free(r);                                  // free memory
  magma_free(h_work);                             // free memory
  magma_free(a1);                                 // free memory
  magma_queue_destroy(queue);                   // destroy queue
  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
```

4.7.13 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` or `MagmaNoVec` 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`) or upper (`MagmaUpper`) 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_v2.h"
#include "magma_lapack.h"
int main( int argc, char** argv) {
  magma_init();                                  // initialize Magma
  magma_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  double  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, info;
  float mione = -1.0f;
  magma_int_t incr = 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(MagmaVec,MagmaLower,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
// Randomize the matrix a and copy a -> r
  lapackf77_slarnv(&ione,ISEED,&n2,a);
  lapackf77_slacpy(MagmaFullStr,&n,&n,a,&n,r,&n);
  magma_ssetmatrix( n, n, a, n, d_r,n,queue); // copy a -> d_r
// compute the eigenvalues  and eigenvectors for a symmetric,
// real nxn matrix; Magma version
  gpu_time = magma_sync_wtime(NULL);

  magma_ssyevd_gpu(MagmaVec,MagmaLower,n,d_r,n,w1,r,n,h_work,
                       lwork,iwork,liwork,&info);

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("ssyevd gpu time: %7.5f sec.\n",gpu_time);   // Magma
// Lapack version                                      // time
  cpu_time=magma_wtime();
  lapackf77_ssyevd("V","L",&n,a,&n,w2,h_work,&lwork,iwork,
                                              &liwork,&info);
  cpu_time=magma_wtime()-cpu_time;
  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_queue_destroy(queue);                  // destroy queue
  magma_finalize();                           // finalize Magma
  return EXIT_SUCCESS;
}
//ssyevd gpu time: 5.11538 sec.
//ssyevd cpu time: 49.32742 sec.
```

```
//difference in eigenvalues: 9.765625e-04
```

4.7.14 magma_ssyevd_gpu - unified memory version, big matrix

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.h"
#include "magma_lapack.h"
int main( int argc, char** argv) {
  magma_init();                              // initialize Magma
  magma_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  double  gpu_time, cpu_time;
  magma_int_t n=8192, n2=n*n;
  float  *a, *r;                          // a, r - nxn matrices
  float *a1; // nxn matrix, copy of a used in magma_ssyevd_gpu
  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, info;
  float mione = -1.0;
  magma_int_t incr = 1;
  magma_int_t ISEED[4] = {0,0,0,1};                         // seed
  cudaMallocManaged(&w1,n*sizeof(float)); //unified memory for
  cudaMallocManaged(&w2,n*sizeof(float));        //eigenvalues
  cudaMallocManaged(&a,n2*sizeof(float)); //unif. memory for a
  cudaMallocManaged(&r,n2*sizeof(float)); //unif. memory for r
  cudaMallocManaged(&a1,n2*sizeof(float));//unif.memory for a1
// Query for workspace sizes
  float aux_work[1];
  magma_int_t aux_iwork[1];
  magma_ssyevd_gpu(MagmaVec,MagmaLower,n,a1,n,w1,r,n,aux_work,
                             -1,aux_iwork,-1,&info );
  lwork  = (magma_int_t) aux_work[0];
  liwork = aux_iwork[0];      // unified memory for workspace:
  cudaMallocManaged(&iwork,liwork*sizeof(magma_int_t));
  cudaMallocManaged(&h_work,lwork*sizeof(float));
// Randomize the matrix a and copy a -> r
  lapackf77_slarnv(&ione,ISEED,&n2,a);
  lapackf77_slacpy(MagmaFullStr,&n,&n,a,&n,r,&n);
  magma_ssetmatrix( n, n, a, n, a1,n,queue);   // copy a -> a1
// compute the eigenvalues  and eigenvectors for a symmetric,
// real nxn matrix; Magma version
  gpu_time = magma_sync_wtime(NULL);
  magma_ssyevd_gpu(MagmaVec,MagmaLower,n,a1,n,w1,r,n,h_work,
                       lwork,iwork,liwork,&info);

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("ssyevd_gpu gpu time: %7.5f sec.\n",gpu_time);//Magma
// Lapack version                                          time
  cpu_time=magma_wtime();
  lapackf77_ssyevd("V","L",&n,a,&n,w2,h_work,&lwork,iwork,
                                               &liwork,&info);
  cpu_time=magma_wtime()-cpu_time;
  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);
  magma_free(w1);                                 // free  memory
  magma_free(w2);                                 // free  memory
  magma_free(a);                                  // free  memory
  magma_free(r);                                  // free  memory
  magma_free(h_work);                             // free  memory
  magma_free(a1);                                 // free  memory
  magma_queue_destroy(queue);                    // destroy queue
  magma_finalize();                             // finalize Magma
  return EXIT_SUCCESS;
}
//ssyevd_gpu gpu time: 5.29559 sec.
//ssyevd cpu time: 51.07547 sec.
//difference in eigenvalues: 9.765625e-04
```

4.7.15 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 or MagmaNoVec 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) or upper (MagmaUpper) 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_v2.h"
#include "magma_lapack.h"
int main( int argc, char** argv) {
  magma_init();                              // initialize Magma
  magma_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  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, info;
  double mione = -1.0;
  magma_int_t incr = 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(MagmaVec,MagmaLower,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
// Randomize the matrix a and copy a -> r
  lapackf77_dlarnv(&ione,ISEED,&n2,a);
  lapackf77_dlacpy(MagmaFullStr,&n,&n,a,&n,r,&n);
  magma_dsetmatrix( n, n, a, n, d_r,n,queue); // copy a -> d_r
// compute the eigenvalues  and eigenvectors for a symmetric,
// real nxn matrix; Magma version
  gpu_time = magma_sync_wtime(NULL);

  magma_dsyevd_gpu(MagmaVec,MagmaLower,n,d_r,n,w1,r,n,h_work,
                      lwork,iwork,liwork,&info);

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("dsyevd_gpu gpu time: %7.5f sec.\n",gpu_time);//Magma
// Lapack version                                         time
  cpu_time=magma_wtime();
  lapackf77_dsyevd("V","L",&n,a,&n,w2,h_work,&lwork,iwork,
                                             &liwork,&info);
  cpu_time=magma_wtime()-cpu_time;
  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_queue_destroy(queue);                        // destroy queue
  magma_finalize();                                 // finalize Magma
  return EXIT_SUCCESS;
}
//dsyevd_gpu gpu time: 16.50546 sec.
//dsyevd cpu time: 91.54085 sec.
//difference in eigenvalues: 1.364242e-11
```

4.7.16 magma_dsyevd_gpu - unified memory version, big matrix

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.h"
#include "magma_lapack.h"
int main( int argc, char** argv) {
  magma_init();                                   // initialize Magma
  magma_queue_t queue=NULL;
  magma_int_t dev=0;
  magma_queue_create(dev,&queue);
  double  gpu_time, cpu_time;
  magma_int_t n=8192, n2=n*n;
  double  *a, *r;                              // a, r - nxn matrices
  double *a1;// nxn matrix, copy of a used in magma_dsyevd_gpu
  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, info;
  double mione = -1.0;
  magma_int_t incr = 1;
  magma_int_t ISEED[4] = {0,0,0,1};                           // seed
  cudaMallocManaged(&w1,n*sizeof(double));//unified memory for
  cudaMallocManaged(&w2,n*sizeof(double));       //eigenvalues
  cudaMallocManaged(&a,n2*sizeof(double));//unif. memory for a
  cudaMallocManaged(&r,n2*sizeof(double));//unif. memory for r
  cudaMallocManaged(&a1,n2*sizeof(double));//uni.memory for a1
// Query for workspace sizes
  double aux_work[1];
  magma_int_t aux_iwork[1];
  magma_dsyevd_gpu(MagmaVec,MagmaLower,n,a1,n,w1,r,n,aux_work,
                              -1,aux_iwork,-1,&info );
  lwork  = (magma_int_t) aux_work[0];
  liwork = aux_iwork[0];      // unified memory for workspace:
  cudaMallocManaged(&iwork,liwork*sizeof(magma_int_t));
  cudaMallocManaged(&h_work,lwork*sizeof(double));
// Randomize the matrix a and copy a -> r
  lapackf77_dlarnv(&ione,ISEED,&n2,a);
  lapackf77_dlacpy(MagmaFullStr,&n,&n,a,&n,r,&n);
  magma_dsetmatrix( n, n, a, n, a1,n,queue); // copy a -> a1
// compute the eigenvalues  and eigenvectors for a symmetric,
// real nxn matrix; Magma version
  gpu_time = magma_sync_wtime(NULL);

  magma_dsyevd_gpu(MagmaVec,MagmaLower,n,a1,n,w1,r,n,h_work,
                       lwork,iwork,liwork,&info);

  gpu_time = magma_sync_wtime(NULL)-gpu_time;
  printf("dsyevd_gpu gpu time: %7.5f sec.\n",gpu_time);//Magma
// Lapack version                                          time
  cpu_time=magma_wtime();
  lapackf77_dsyevd("V","L",&n,a,&n,w2,h_work,&lwork,iwork,
                                              &liwork,&info);
  cpu_time=magma_wtime()-cpu_time;
  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);
  magma_free(w1);                             // free  memory
  magma_free(w2);                             // free  memory
  magma_free(a);                              // free  memory
  magma_free(r);                              // free  memory
  magma_free(h_work);                         // free  memory
  magma_free(a1);                             // free  memory
  magma_queue_destroy(queue);                // destroy queue
  magma_finalize();                          // finalize Magma
  return EXIT_SUCCESS;
}
//dsyevd_gpu gpu time: 16.55437 sec.
//dsyevd cpu time: 95.21645 sec.
//difference in eigenvalues: 1.364242e-11
```

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:

`MagmaAllVec` - all m columns of u are returned in an array u;
`MagmaSomeVec` - the first $\min(m, n)$ columns of u (the left singular vectors) are returned in the array u;
`MagmaOverwriteVec` - the first $\min(m, n)$ columns of u are overwritten on the array A;
`MagmaNoVec` - no left singular vectors are computed.

Similarly the second argument can take the following values:

`MagmaAllVec` - all n rows of v^T are returned in an array vt;
`MagmaSomeVec` - the first $\min(m, n)$ rows of v^T (the right singular vectors) are returned in the array vt;
`MagmaOverwriteVec` - the first $\min(m, n)$ rows of v^T are overwritten on the array A;
`MagmaNoVec` - 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_v2.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(m,n);//optim.block size
  lwork=min_mn*min_mn+2*min_mn+2*min_mn*nb;
  magma_smalloc_pinned(&h_work,lwork); // host mem. for h_work
// Randomize the matrix a
  lapackf77_slarnv(&ione,ISEED,&n2,a);
  lapackf77_slacpy(MagmaFullStr,&m,&n,a,&m,r,&m);        //a->r
// MAGMA
  gpu_time = magma_wtime();
// compute the singular value decomposition of r (copy of a)
// and optionally the left and right singular vectors:
// r = 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.vec.

  magma_sgesvd(MagmaNoVec,MagmaNoVec,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("N","N",&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:  15.00651
//sgesvd cpu time:  115.81860
//difference:  5.943540e-07
```

4.8.2 magma_sgesvd - unified memory version

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.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
  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
  cudaMallocManaged(&a,n2*sizeof(float)); //unif. memory for a
  cudaMallocManaged(&vt,n*n*sizeof(float));//uni.memory for vt
  cudaMallocManaged(&u,m*m*sizeof(float));//unif. memory for u
  cudaMallocManaged(&s1,min_mn*sizeof(float));//uni.mem.for s1
  cudaMallocManaged(&s2,min_mn*sizeof(float));//uni.mem.for s2
  cudaMallocManaged(&r,n2*sizeof(float)); //unif. memory for r
  magma_int_t nb = magma_get_sgesvd_nb(m,n);//optim.block size
  lwork=min_mn*min_mn+2*min_mn+2*min_mn*nb;
  cudaMallocManaged(&h_work,lwork*sizeof(float)); //m.f.h_work
// Randomize the matrix a
  lapackf77_slarnv(&ione,ISEED,&n2,a);
  lapackf77_slacpy(MagmaFullStr,&m,&n,a,&m,r,&m);        //a->r
// MAGMA
  gpu_time = magma_wtime();
// compute the singular value decomposition of r (copy of a)
// and optionally the left and right singular vectors:
// r = 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.vec.

  magma_sgesvd(MagmaNoVec,MagmaNoVec,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("N","N",&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
  magma_free(a);                                     // free memory
  magma_free(vt);                                    // free memory
  magma_free(s1);                                    // free memory
  magma_free(s2);                                    // free memory
  magma_free(u);                                     // free memory
  magma_free(h_work);                                // free memory
  magma_free(r);                                     // free memory
  magma_finalize( );                               // finalize Magma
  return EXIT_SUCCESS;
}
//sgesvd gpu time:  16.51667
//sgesvd cpu time:  115.20410
//difference:  2.810940e-03
```

4.8.3 `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:

`MagmaAllVec` - all m columns of u are returned in an array u;
`MagmaSomeVec` - the first $\min(m,n)$ columns of u (the left singular vectors) are returned in the array u;
`MagmaOverwriteVec` - the first $\min(m,n)$ columns of u are overwritten on the array A;
`MagmaNoVec` - no left singular vectors are computed.

Similarly the second argument can take the following values:

`MagmaAllVec` - all n rows of v^T are returned in an array vt;
`MagmaSomeVec` - the first $\min(m,n)$ rows of v^T (the right singular vectors)

are returned in the array vt;
`MagmaOverwriteVec` - the first $\min(m,n)$ rows of v^T are overwritten on the array A;
`MagmaNoVec` - 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_v2.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(m,n);//optim.block size
  lwork=min_mn*min_mn+2*min_mn+2*min_mn*nb;
  magma_dmalloc_pinned(&h_work,lwork); // host mem. for h_work
// Randomize the matrix a
  lapackf77_dlarnv(&ione,ISEED,&n2,a);
  lapackf77_dlacpy(MagmaFullStr,&m,&n,a,&m,r,&m);       //a->r
// MAGMA
  gpu_time = magma_wtime();
// compute the singular value decomposition of r (copy of a)
// and optionally the left and right singular vectors:
// r = 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.vec.
```

```
  magma_dgesvd(MagmaNoVec,MagmaNoVec,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("N","N",&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:  23.05454
//dgesvd cpu time:  228.58973
//difference:  1.526458e-15
```

4.8.4 magma_dgesvd - unified memory version

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.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
  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
  cudaMallocManaged(&a,n2*sizeof(double));//unif. memory for a
  cudaMallocManaged(&vt,n*n*sizeof(double));//unif .mem for vt
  cudaMallocManaged(&u,m*m*sizeof(double));//unif.memory for u
  cudaMallocManaged(&s1,min_mn*sizeof(double));//un.mem.for s1
  cudaMallocManaged(&s2,min_mn*sizeof(double));//un.mem.for s2
  cudaMallocManaged(&r,n2*sizeof(double));//unif. memory for r
  magma_int_t nb = magma_get_dgesvd_nb(m,n);//optim.block size
  lwork=min_mn*min_mn+2*min_mn+2*min_mn*nb;
  cudaMallocManaged(&h_work,lwork*sizeof(double));//m.f.h_work
// Randomize the matrix a
  lapackf77_dlarnv(&ione,ISEED,&n2,a);
  lapackf77_dlacpy(MagmaFullStr,&m,&n,a,&m,r,&m);       //a->r
// MAGMA
  gpu_time = magma_wtime();
// compute the singular value decomposition of r (copy of a)
// and optionally the left and right singular vectors:
// r = 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.vec.

  magma_dgesvd(MagmaNoVec,MagmaNoVec,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("N","N",&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
  magma_free(a);                                // free memory
  magma_free(vt);                               // free memory
  magma_free(s1);                               // free memory
  magma_free(s2);                               // free memory
  magma_free(u);                                // free memory
  magma_free(h_work);                           // free memory
  magma_free(r);                                     // free memory
  magma_finalize( );                               // finalize Magma
  return EXIT_SUCCESS;
}
//dgesvd gpu time:  25.20418
//dgesvd cpu time:  231.04632
//difference:  7.219722e-12
```

4.8.5 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_v2.h"
#include "magma_lapack.h"
#define min(a,b)  (((a)<(b))?(a):(b))
int main( int argc, char** argv){
  magma_init();                                   // initialize Magma
  double 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;                                      // workspace
  magma_int_t lhwork;                               // size of h_work
  float *taup, *tauq; // arrays describ. elementary reflectors
  float *diag, *offdiag;       // bidiagonal form in two arrays
  magma_int_t  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(m,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
// Randomize the matrix a
  lapackf77_slarnv( &ione, ISEED, &n2, a );
  lapackf77_slacpy(MagmaFullStr,&m,&n,a,&m,r,&m);       // a->r
// MAGMA
  gpu_time = magma_wtime();
// reduce the matrix r to upper bidiagonal form by orthogonal
// transformations: q^T*r*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);

  gpu_time = magma_wtime() - gpu_time;
  printf("sgebrd gpu time: %7.5f sec.\n",gpu_time);
// LAPACK
  cpu_time = magma_wtime();
  lapackf77_sgebrd(&m,&n,a,&m,diag,offdiag,tauq,taup,h_work,
                                            &lhwork,&info);
  cpu_time = magma_wtime() - cpu_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: 2.28088 sec.
//sgebrd cpu time: 13.83244 sec.
```

4.8.6 magma_sgebrd - unified memory version

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.h"
#include "magma_lapack.h"
#define min(a,b)  (((a)<(b))?(a):(b))
```

```
int main( int argc, char** argv){
  magma_init();                                 // initialize Magma
  double gpu_time, cpu_time;
  magma_int_t m = 4096, n = 4096, n2=m*n;
  float *a, *r;                              // a,r - mxn matrices
  float *h_work;                                       // workspace
  magma_int_t lhwork;                               // size of h_work
  float *taup, *tauq; // arrays describ. elementary reflectors
  float *diag, *offdiag;      // bidiagonal form in two arrays
  magma_int_t  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(m,n);        // optimal block size
  cudaMallocManaged(&a,n2*sizeof(float)); //unif. memory for a
  cudaMallocManaged(&r,n2*sizeof(float)); //unif. memory for r
  cudaMallocManaged(&tauq,minmn*sizeof(float)); //mem.for tauq
  cudaMallocManaged(&taup,minmn*sizeof(float)); //mem.for taup
  cudaMallocManaged(&diag,minmn*sizeof(float)); //mem.for diag
  cudaMallocManaged(&offdiag,(minmn-1)*sizeof(float)); //unif.
  lhwork = (m + n)*nb;                          //memory for offdiag
  cudaMallocManaged(&h_work,lhwork*sizeof(float)); //unif.mem.
// Randomize the matrix a                           // for workspace
  lapackf77_slarnv( &ione, ISEED, &n2, a );
  lapackf77_slacpy(MagmaFullStr,&m,&n,a,&m,r,&m);      // a->r
// MAGMA
  gpu_time = magma_wtime();
// reduce the matrix r to upper bidiagonal form by orthogonal
// transformations: q^T*r*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);

  gpu_time = magma_wtime() - gpu_time;
  printf("sgebrd gpu time: %7.5f sec.\n",gpu_time);
// LAPACK
  cpu_time = magma_wtime();
  lapackf77_sgebrd(&m,&n,a,&m,diag,offdiag,tauq,taup,h_work,
                                            &lhwork,&info);
  cpu_time = magma_wtime() - cpu_time;
  printf("sgebrd cpu time: %7.5f sec.\n",cpu_time);
// free memory
  magma_free(a);                                   // free memory
  magma_free(tauq);                                // free memory
  magma_free(taup);                                // free memory
  magma_free(diag);                                // free memory
  magma_free(r);                                   // free memory
  magma_free(h_work);                                 // free memory
  magma_finalize( );                               // finalize Magma
  return EXIT_SUCCESS;
}
//sgebrd gpu time: 2.45375 sec.
//sgebrd cpu time: 12.81832 sec.
```

4.8.7 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_v2.h"
#include "magma_lapack.h"
#define min(a,b)  (((a)<(b))?(a):(b))
int main( int argc, char** argv){
  magma_init();                                   // initialize Magma
  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;                                       // workspace
  magma_int_t lhwork;                               // size of h_work
  double *taup, *tauq;// arrays describ. elementary reflectors
  double *diag, *offdiag;      // bidiagonal form in two arrays
  magma_int_t  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(m,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
// Randomize the matrix a
  lapackf77_dlarnv( &ione, ISEED, &n2, a );
  lapackf77_dlacpy(MagmaFullStr,&m,&n,a,&m,r,&m);      // a->r
// MAGMA
  gpu_time = magma_wtime();
// reduce the matrix r to upper bidiagonal form by orthogonal
// transformations: q^T*r*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);

  gpu_time = magma_wtime() - gpu_time;
  printf("dgebrd gpu time: %7.5f sec.\n",gpu_time);
// LAPACK
  cpu_time = magma_wtime();
  lapackf77_dgebrd(&m,&n,a,&m,diag,offdiag,tauq,taup,h_work,
                                             &lhwork,&info);
  cpu_time = magma_wtime() - cpu_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: 3.54390 sec.
//dgebrd cpu time: 29.55658 sec.
```

4.8.8 magma_dgebrd - unified memory version

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "magma_v2.h"
#include "magma_lapack.h"
#define min(a,b)  (((a)<(b))?(a):(b))
int main( int argc, char** argv){
  magma_init();                               // initialize Magma
  double gpu_time, cpu_time;
  magma_int_t m = 4096, n = 4096, n2=m*n;
  double *a, *r;                               // a,r - mxn matrices
  double *h_work;                                       // workspace
  magma_int_t lhwork;                               // size of h_work
  double *taup, *tauq;// arrays describ. elementary reflectors
  double *diag, *offdiag;     // bidiagonal form in two arrays
  magma_int_t  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(m,n);         // optimal block size
  cudaMallocManaged(&a,n2*sizeof(double));//unif. memory for a
  cudaMallocManaged(&r,n2*sizeof(double));//unif. memory for r
  cudaMallocManaged(&tauq,minmn*sizeof(double));//mem.for tauq
  cudaMallocManaged(&taup,minmn*sizeof(double));//mem.for taup
  cudaMallocManaged(&diag,minmn*sizeof(double));//mem.for diag
  cudaMallocManaged(&offdiag,(minmn-1)*sizeof(double));//unif.
  lhwork = (m + n)*nb;                        //memory for offdiag
  cudaMallocManaged(&h_work,lhwork*sizeof(double));//workspace
// Randomize the matrix a
  lapackf77_dlarnv( &ione, ISEED, &n2, a );
  lapackf77_dlacpy(MagmaFullStr,&m,&n,a,&m,r,&m);       // a->r
// MAGMA
  gpu_time = magma_wtime();
// reduce the matrix r to upper bidiagonal form by orthogonal
// transformations: q^T*r*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);

  gpu_time = magma_wtime() - gpu_time;
  printf("dgebrd gpu time: %7.5f sec.\n",gpu_time);
// LAPACK
  cpu_time = magma_wtime();
  lapackf77_dgebrd(&m,&n,a,&m,diag,offdiag,tauq,taup,h_work,
                                              &lhwork,&info);
  cpu_time = magma_wtime() - cpu_time;
  printf("dgebrd cpu time: %7.5f sec.\n",cpu_time);
// free memory
  magma_free(a);                                   // free memory
  magma_free(tauq);                                // free memory
  magma_free(taup);                                // free memory
  magma_free(diag);                                // free memory
  magma_free(r);                                   // free memory
  magma_free(h_work);                              // free memory
  magma_finalize( );                              // finalize Magma
  return EXIT_SUCCESS;
}
//dgebrd gpu time: 3.77582 sec.
//dgebrd cpu time: 28.54319 sec.
```

Bibliography

[CUBLAS] *CUBLAS LIBRARY User Guide*, Nvidia, June 2017
http://docs.nvidia.com/cuda/pdf/CUBLAS_Library.pdf

[CUSOLVER] *CUSOLVER LIBRARY*, Nvidia , June 2017
http://docs.nvidia.com/cuda/pdf/CUSOLVER_Library.pdf

[MAGMA] *MAGMA Users' Guide* , Univ. of Tennessee, Knoxville, Univ. of California, Berkeley, Univ. of Colorado, Denver, November 2016
http://icl.cs.utk.edu/projectsfiles/magma/doxygen/

[ARRAYFIRE] Chrzęszczyk A., *Matrix Computations on the GPU with ArrayFire-Python and ArrayFire-C/C++. Version 2017*, July 2017
https://www.researchgate.net/publication/319135914_Matrix_Computations_on_GPU_with_ArrayFire-Python_and_ArrayFire-CCVersion_2017

[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/