Implementation of LU, QR and RNG in Flagon

Liping Liu
Department of Computer Science
University of Maryland, College Park

Abstract: This paper introduces fast LU and QR implementations on GPU which are extended from LAPACK routines. Using fast matrix-matrix multiplication algorithm on GPU, right-looking technique to parallelize the computation, look-ahead technique to override the CPU and GPU computation together with optimal block size on GPU make this implementation outperform its counterparts. It gains around 2~8x speedup over LAPACK routines which are run on CPU as the number of rows in matrix varies from 1,000 to 11,000. The paper also provides detailed information on how to add customer functionality into Flagon which makes possible that developer can use CUDA code without knowing how to do CUDA coding. In this paper, a random generator on GPU is also imported into Flagon.

Key words: LU, QR, random number generator, Flagon

Introduction

Graphics Processor Units (GPUs) have been used for many applications beyond graphics. GPGPU (General-purpose Computing on Graphics Units, also referred as GPGP) is introduced to prove that many computations in applications traditionally handled by CPU can also be handled by GPU. There are a lot of research on using GPU for scientific computations [Fan et al. 2004, Harris et al. 2003, Rumpf and Strzodka 2001], iterative solvers for sparse linear systems[Bolz et al. 2003, Kruger and Werstermann 2003, Goddeke 2005], and other applications. In recent years, there are also some research on proposing GPU implementation for some subroutines in linear algebra package [Galoppo et al. 2005, Volkov and Demmel 2008, Baboulin et al. 2008, Barrachina et al. 2008] in order to provide high performance computing.

The Linear Algebra PACKage (LAPACK) is a software library that uses block-partitioned algorithms for performing dense and banded linear algebra computations on vector and shared memory computers [Choi et al. 1994]. It provides routines for solving systems of linear equation and linear lest squares, eigenvalue problems, singular value decomposition, and also matrix factorizations such as LU, QR and Cholesky. LAPACK is considered as the successor of LINPACK and the EISPACK. LAPACK has also been extended to run on distributed-memory systems in extension package such as ScalLAPACK and PLAPACK.

In this paper, we will introduce fast LU and QR implementations on GPU which are extended from LAPACK routines.

CUDA programming provides some benefits in producing high performance. However, it uses a lot of build-in APIs, thus increasing the usage difficulty for developers. What if we can use CUDA code without knowing its details? Is there such kind of tool or library? Yes, we have. Flagon is the right library for people who want to use CUDA code but don't want to know how to write CUDA code. So far, Flagon can provide CUDPP, CUBLAS, and CUFFT libraries.

In this paper, we will focus on adding GPU-based LU, QR and a random generator in FLAGON. This experience can also server as guidance for adding other libraries with GPU version into Flagon.

We make the following contributions: 1) Introduce a fast GPU-based implementation of LU, QR in detail and provide a random number generator implementation; 2) Provide manual for adding customer functionality in Flagon.

The organization of the paper is as follows: section 2 goes through the literature review, section 3 introduces the design philosophy of LU, QR and RNG, section 4 describes the details of adding customer functionality into Flagon, section 5 analyzes the performance of LU and QR, section 6 introduces the application of LU and QR, the last section makes a conclusion of this paper.

Literature review

LU and QR

Lu and QR are two classic matrix factorization algorithms in solving linear equation problems. The Linear Algebra PACKage (LAPACK) which is a software library written in Fortran 77 originally and in Fortran 90 now provides routines for LU and QR matrix factorization. The implementation of the routines schedules some Basic Linear Algebra Subprograms (BLAS) level 2 (matrix-vector) and level 3 (matrix-matrix) operations. The routines have also been extended to run on distributed-memory systems in ScaLAPACK and PLAPACK. Currently, some variants of LU, QR routines are extended from the LAPACK routines.

Matlab is also embedded with LU, QR routines. The basic version of LU is based on using nested for-loops to do the matrix inversion and matrix-matrix multiplication. So the main running time happens on the three levels for-loop to calculate the matrix-matrix multiplication, which makes the implementation far from efficient.

Among numbers of matrix factorization research topics, one of them is to generate efficient and fast LU and QR routines by using parallel techniques.

There are mainly three bulk-synchronous variants of LU factorization, which are left-looking, right-looking and Crout [Dongarra et al. 1998]. In left-looking, all data accesses happen to the left of the block column being updated which has the only one write access. The matrix elements on the right side are only needed for pivoting purpose. Left-looking is considered the best among the three from the standpoint of data access. In right-looking, it will produce the first k columns of L and first k rows of U while in left-looking it produces same columns of L and U. It's easy to apply the factorization to generate the next block column L and next block row U. If the block size used in right-looking is great than 1, level 3 operation will perform more efficiently than level 2 operation. Right-looking is considered the one of three which has the most potential for parallelism. Crout algorithm is best suited of the three for vector machines with enough memory bandwidth. But no matter which algorithm you use, for any implementation of LU, the computation work includes three routines: 1) panel factorization within the current block column; 2) matrix inversion which is to update the triangular; 3) matrix-matrix multiply which is to update the un-factorized part of original matrix.

There are several LU and QR GPU-based implementations. In [Galoppo et al. 2005], it uses appropriate data representation to match the blocked rasterization order and non-blocked algorithm to swap rows and columns for efficient implementation of partial and full pivoting. As for performance, the author claimed that it ran up to 10Gflop/s for n=4000 without pivoting and 6Gflop/s for n=3500 with partial pivoting on GeForce 7800. In [Barrachina et al. 2008], it evaluated three blocked variants of LU factorization-with padding, hybrid GPU-CPU computation and recursion. It ran up to 50Gflop/s in LU for n=5000 on GeForce 8800 Ultra. In [Baboulin et al. 2008], it proposed an algorithm based on randomization technique for LU pivoting and use a lot of BLAS 3 computations. The author claimed that their LU and QR implementation ran up to 55Gflop/s on Quadro FX5600 for n=19000 using CUBLAS 1.0. In

[Vokvok and Demmel 2008], the author claimed that they have implemented the fastest LU and QR so far. It ran up to 309Gflop/s on GTX280 and E6700 for LU, 340Gflop/s for QR.

In this paper, we will work on [Vokvok and Demmel 2008]'s LU and QR implementation and import it into Flagon as it achieved the highest performance compared with other implementations introduced above. The high performance in LU gains from using several techniques: 1) right-looking for using parallelism; 2) look-ahead technique; 3) fast matrix-matrix multiply routine on GPU; 4) fast matrix inversion on GPU; 5) panel factorization on CPU; 6) optimal block size. The high performance in QR implementation is achieved by using the following techniques: 1) right-looking manner to recursively generate Q and R; 2) fill in the lower triangular part with zeros and a unit diagonal by using a matrix-matrix multiply; 3) use optimum block size by auto-tuning during the process of factorization. The details of the implementation will be introduced in section 3.

Random Number Generator

The generation of pseudo-random numbers is important in computing program. There are some library functions for generating random numbers. However, they usually have some statistical problems or the random number they generate repeats in some period of time. In order to provide high quality-- long period and highly random, there are some implementations with parallelism. One of them is called Mersenne Twister [Internet Link]. It performs in high quality, but the implementation itself is complicated. Here we will work on another simpler implementation, which is posted on Nvidia Forum [Nvidia Forum].

Design Philosophy

LU and QR

LU and QR are two of the most widely used factorizations in the dense linear algebra and usually used as benchmarks for solvers of dense algebra system. In this section, we will introduce a fast GPU-base implementation of LU and QR. From here to end of the paper, we call them GPU-LU and GPU-QR.

For a given matrix A (M,N), LU factorization is to generate a permutation matrix P which is stored in a min(M,N) vector, low triangular matrix L (M,N) and upper triangular matrix U (N,N)in the form of $A = PLU$. The output is generated by applying a series of Gaussian eliminations. In blocked algorithm, several computations are conducted recursively during the whole process. The computation of each round includes (see Fig 1): 1) panel factorization to generate L_{11}, L_{21} and U_{11}; 2) update the right side matrix of current block which includes U_{12} and A_{22} by formula (1) and (2).

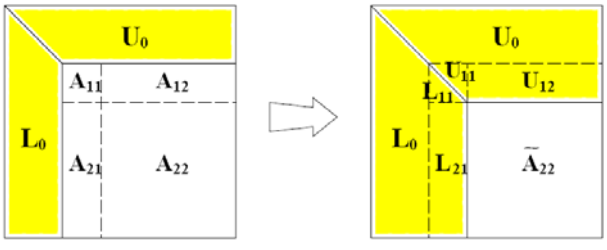


Fig1. The procedure of LU [Choi et al. 1994]

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = p\begin{pmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{pmatrix}\begin{pmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{pmatrix} = p\begin{pmatrix} L_{11}U_{11} & L_{11}U_{12} \\ L_{21}U_{11} & L_{21}U_{12} + L_{22}U_{22} \end{pmatrix} \quad (1)$$

$$U_{12} \leftarrow (L_{11})^{-1} A_{12}$$
$$A_{22} \leftarrow A_{22} - L_{21}U_{12} = L_{22}U_{22} \quad (2)$$

For a given matrix A (M,N), QR is to generate an M*M orthogonal matrix which we call Q and an M*N triangular matrix which we call R. Q is computed by applying a number of Householder transformations to the current block column in the form of $H_i = I - \tau_i v_i v_i^T$ where $i = 1, 2 \ldots nb$, nb is the block size. Several computations are conducted recursively during the whole process. Every round of computation includes: 1) compute the Householder V; 2) compute row panel R11 and R12 for the current block column; 3) update the un-factorized part of original matrix which is on the right side of the current block column by formula (6).

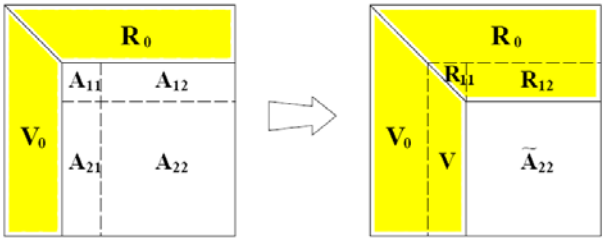


Fig 2. The procedure of QR [Choi et al. 1994]

$$A^{(k)} = \begin{pmatrix} A_1 & A_2 \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = Q \bullet \begin{pmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{pmatrix} \quad (3)$$

Q is computed by applying a series of Householder transformations to A_1

$$Q = H_1 H_2 \cdots H_{nb} = I - VTV^T \quad (4)$$

$$H_i = I - \tau_i v_i v_i^T \qquad \tau_i = 2/(v_i^T v_i) \quad (5)$$

$$A_2 = \begin{pmatrix} A_{12} \\ A_{22} \end{pmatrix} \leftarrow \begin{pmatrix} R_{12} \\ R_{22} \end{pmatrix} = Q^T A_2 = (I - VT^T V^T) A_2 \quad (6)$$

The design of GPU-LU and GPU-QR for the implementation in this paper is obtained from [Volvok and Demmel, 2008]. It's extended from the LAPAPCK LU and QR routine, but with some new features which make it outperform its counterparts.

- Use GPU to compute matrix-matrix multiplies only

The main time for factorization is consumed by updating the whole/part of the matrix once a panel factorization of a block column is done. The updating is achieved by some matrix-matrix multiplication in term of computation. GPU performs much better than CPU in matrix-matrix multiplication. This implementation takes advantage of this feature and makes all the matrix-matrix implementation happen on GPU while leave the panel computation on CPU.

- Use look-ahead to overlap computations on CPU and GPU

After panel factorization is done on CPU, the factorized part on the CPU will be transferred to GPU on the final output location to overlap the computation on the GPU. However, this

requires a lot of memory transfer between CPU and GPU, which consumes some time of the whole factorization. Fortunately the transfer time is relatively small compared with the time used in matrix-matrix multiplication.

- Use right-looking algorithm to have more threads in SGEMM

Right-looking algorithm enables more parallelism in calling matrix-matrix multiplication for updating the whole matrix after panel factorization is done.

- Use row-major layout on GPU in LU factorization

Row-major layout is used on GPU in order to avoid penalized strided memory access in LU pivoting on GPU.

Fig 3 summaries the whole process of GPU-LU and GPU-QR, in which panel factorization is nothing but the LAPACK routines which are done on CPU. Step 3~ step 6 are recursively scheduled until the whole matrix is factorized.

Procedure: [#of row, #of column]
1. Transfer [n, (n-nb)] matrix from CPU to GPU
2. Initially i=0
3. Panel factorization [(n-nb*i), nb], upload factorized part to the expected output position on GPU
4. Update matrix on GPU
5. i+=nb
6. transfer [(n-nb*i), nb] from GPU to CPU
7. recursively do step 3~6 until i reaches n

Fig 3. Procedure of GPU version for LU and QR

RNG

In the implementation, it loads the random state from device memory into local registers, then generates random numbers according to the state, and finally store the state back to device memory. Random state is stored in registers and updated with device call. In this version, we generate pseudo random numbers whose size is a multiply of 6*1024[Nvidia Forum].

How to add customer functionality

Flagon is an open source library for using GPU from Fortran 90/95, without knowing too much C or CUDA. It implements Fortran modules which utilizes device variables on the GPU. Some supporting functions such as general functions, memory functions are provided for data transfer, memory allocation and manipulating device variables.

The current FLAGON package contains three folders: CUDPP library containing CUDPP source code, devObjCpp containing all c/c++ and cu files, and devObjFortran containing all Fortran files.

Flagon provides some build-in functions. The build-in function includes: runtime functions (initialization functions to open and close devObject, memory functions to allocate/deallocate memory for device variable, transfer memory between CPU and GPU, copy memory between device and device), CUDPP function, CUBLAS functions, and CUFFT functions.

Developers can also add customer functionalities in FLAGON. Before introducing how to add functionality, we first describe three methods to connect C/CU with Fortran in Flagon.

1. In devObjFortran folder, the subroutines/functions called by Fortran module are defined directly in cu file which resides in devObjCPP. The CUDPP fortran module uses this method.
2. In devObjFortran folder, the subroutines/functions are called by Fortran module are defined in fortran.c in devObjCPP. CUBLAS fortran module uses this method.
3. In devObjFortran folder, some subroutines/functions are defined as interfaces which are associated with c functions in devObjCPP. CUFFT uses this method.

Developers can use either way to adopt customer functionality. Here in our project, we use method 3 which is to define c functions to associate with fortran subroutines and functions.

We use the following example to show how to add customer functionality by using method 3. We have customer cu file example.cu, cpp file c_devExample.cpp, fortran module file devObjectExample.f90, and fortran test file testDevExample.f90 which is used to test the defined Fortran module.

Example.cu: define GPU kernels, and functions which call these kernels.

```
example.cu

__global_void example_kernel (float* A, int a, int b)
{

/*kernel implementation*/

}

__device_void kernel_support(float* A, int a)
{

/*device kernel implementation*/

}

Extern “C” void example_cpp_function(float* A, int a, int b)
{
/*setup grid and thread*/
/*call cu kenerl*/
Example_kernel<<<grid, thread>>>(A, a, b);
}
```

C_devExample.cpp: call functions in cu file, provide implementation of functions which are associated with Fortran interfaces.

```
C_devExample.cpp

/*In cpp file, call functions in cu file */

Extern “C” void example_cpp_function(float* A, int a, int b);

Extern “C” void cf_example(float* A, int a, int b)
{
/*call some cublas functions here */

Example_cpp_function(A, a, b);

/*other implementations here*/
}
```

devObjectExample.f90: declare Fortran interface, setup the association with C files, provide implementation for functions/subroutines for the module.

```
Module devObjectExample
        Use devObjectHeaders
        Use devObjectFunctions

        Implicit none

        Public: devf_example

        Interface

        subroutine fc_example(A, a, b)
          !DEC$ ATTRIBUTES C,DECORATE,ALIAS:'cf_example' :: fc_example
          integer(4) A
          integer a, b
        end subroutine fc_example

        end interface

        contains

        subroutine devf_example (dev, a , b)
        implicit none

        type(devVar) dev
        integer a, b

        call fc_example (dev%dPtr, a, b)
        end subroutine devf_example
End module
```

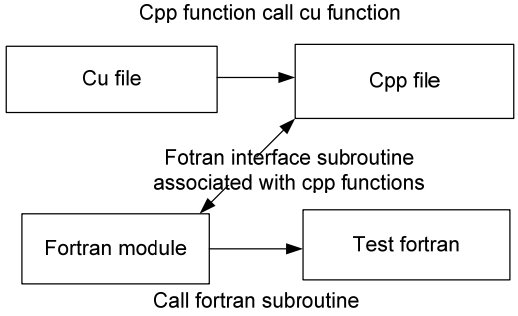
testDevExaple.f90: setup parameters, call subroutines/functions in the module.

```
Subroutine testDevExample
Use devObjectExample
Use devObject
Implicit none

Type(devVar):: dv_A
Parameter a=100
Parameter b=100
Real matrix(a*b)
Call random_number (matrix)
Dv_A=allocate_dv(‘real’, a , b)
Call transfer_r4(matrix, dv_A, .true.)
Call devf_example(dv_A, a, b)

End subroutine testDevExample
```

From the above files, we can draw the following map to show the relationship among these files. This picture contains all information you need when you add customer functionalities in Flagon.



With this picture in mind, it's easy to add new customer functionality in Flagon without changing the existing Flagon structure.

Performance Analysis

We run GPU-LU and GPU-QR which we call gpu_sgetrf (LU) and gpu_sgeqrf (QR) on NVIDIA GT200, and then compare the performance with CPU version-LAPACK routines sgetrf (for LU) and sgeqrf (for QR) which are run on Intel Core 2CPU. We examine the time each routine uses and analyze the speedup the GPU version gains over the CPU version while changing the size of matrix. The matrix in the GPU-based implementation is constrained to be square matrix. We change the number of rows from 1000 to 11000. We also examine the computation error for GPU-LU and GPU-QR.

We run RNG on Nvidia GPU and output the generated random numbers into a text file on CPU. Then, we examine the randomness of the generated number in Matlab.

Configuration

The GPU we use is NVIDIA GT200 with 1080MHz clock and 1024MB memory.
We use Intel Core 2CPU 6600 with 2.4GHz and 2.00G RAM.

Result

The following table shows the GPU-LU, CPU-LU, LU-Speedup, GPU-QR, CPU-QR and QR-Speedup as the size of matrix varies.

Table 1: CPU/GPU time and Speedup for LU and QR

Size of matrix (row)	GPU-LU (ms)	CPU-LU (ms)	LU-Speedup	GPU-QR (ms)	CPU-QR (ms)	QR-Speedup
1000	36.796	44.366	1.2057	53.101	89.467	1.6848
2000	67.603	264.064	3.9061	117.604	497.087	4.2268
3000	159.79	862.464	5.3975	273.574	1563.278	5.7143
4000	319.663	2053.041	6.4225	555.21	3527.239	6.3530
5000	566.291	3758.774	6.6375	989.159	6689.207	6.7625
6000	904.234	6338.352	7.0096	1593.38	11341.754	7.1180
7000	1372.453	9496.77	6.9196	2448.1	17814.553	7.2769
8000	1958.913	14114.3	7.2052	3520.655	26597.563	7.5547
9000	2723.311	19854.508	7.2906	4911.676	37297.179	7.5936
10000	3655.148	26868.673	7.3509	6639.588	50778.505	7.6478
11000	4754.4783	36242.581	7.6228	8667.438	67192.317	7.7523

We draw the following two pictures using the number of the table to show the result more straightforward.

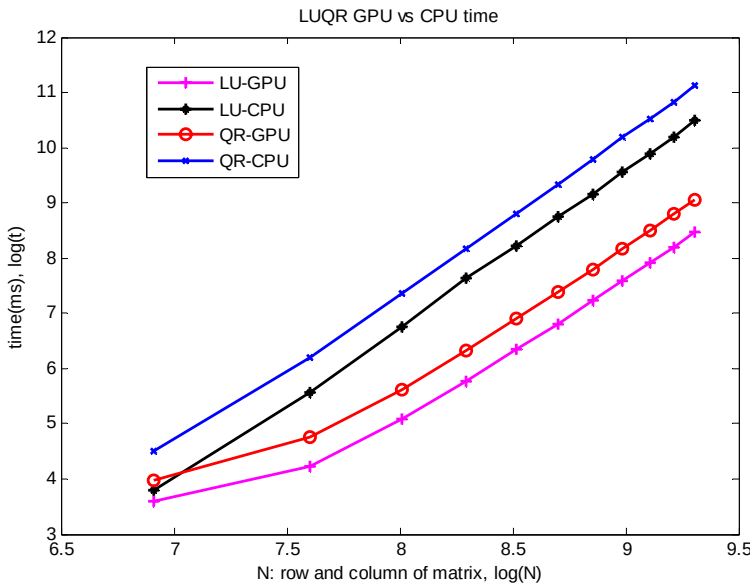


Fig 4. LU, QR GPU vs CPU

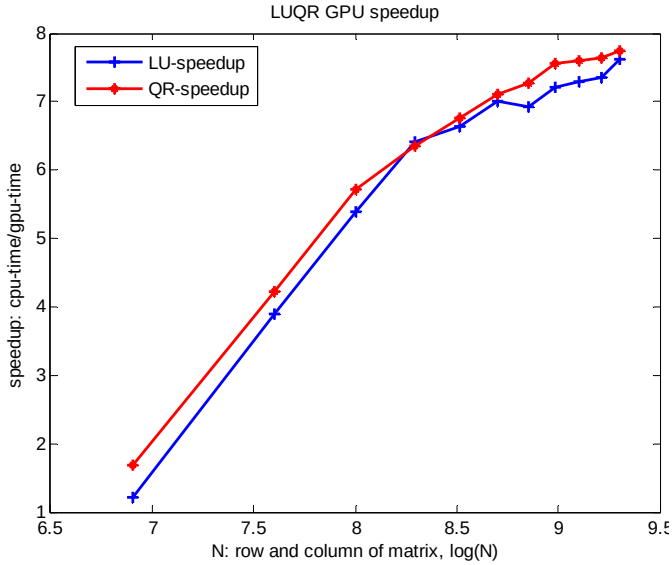


Fig 5. LU, QR speedup

Fig 4 shows the GPU and CPU time for LU and QR as the size of matrix varies. The size is represented as the number of rows in the matrix. In this figure, both CPU and GPU time increases along with the size of matrix. This is obvious as the larger the matrix, the more computations the matrix needs and more time it consumes. From this figure, we can also see that QR consumes more time than LU for both GPU and CPU version. This can be explained when we look at the procedure of LU and QR. There are more matrix-matrix multiplications involved in QR than LU when updating the matrix after panel factorization is done. Therefore, more time is needed for QR.

Fig 5 shows the speedup for LU and QR. The speedup is derived from cpu time over gpu time. From fig 5 and table 1 we see that we gain about 2~8x speedup. From the design of GPU-LU and GPU-QR, we can see the speedup is obtained from a fast matrix-matrix multiply algorithm. The larger proportion of the time matrix-matrix multiplication consumes in the whole time, the more advantage the fast matrix-matrix algorithm has. Compared with LU, QR has more matrix-matrix

multiplications in the procedure which take a larger proportion in the whole time, therefore, QR gets more speedup. With the same reason, we can explain that the speedup for both LU and QR increases along with the size of matrix.

We use the same way with [Volvok and Demmel, 2008] to test the correctness of GPU-LU and GPU-QR. The matrix A is generated with random entries uniformly distributed in [-1, 1]. We multiply the output factors and find its max-norm of its difference with the input matrix. So the error can be expressed as: residual_norm/matrix_norm/eps where residual_norm is obtained from the max-norm of the difference between input matrix and the output factors product and eps is a machine epsilon in IEEE single precision with value $\varepsilon = 2^{-23}$. The purpose of this test is to measure the backward error in the factorization. The result is shown in fig 6.

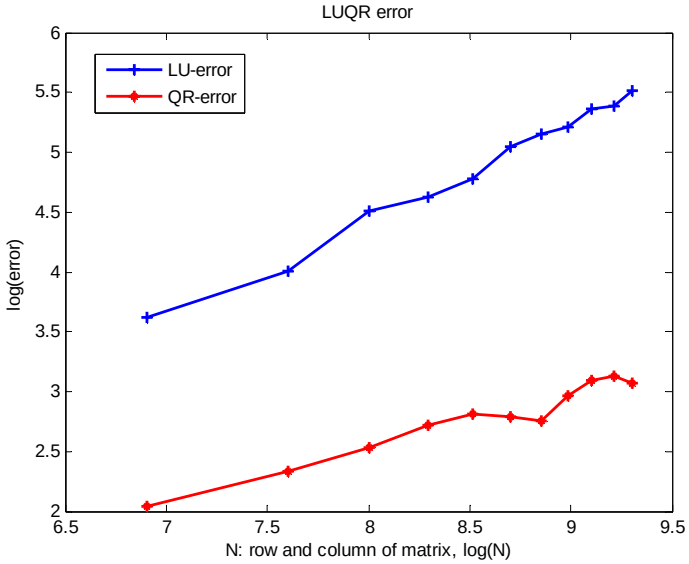


Fig 6. LU QR error

Randomness check

We use Matlab function “runstest” to check the randomness of the random numbers generated by RNG. Runstest is a test of the null hypothesis that the values in a vector come in random order, against the alternative that they do not. We examine the randomness of numbers with different size (1*6*1024~10000*6*1024) and found that all of them come in random number.

Application

In this section, we answer the question: what can we do after we factorize the matrix A in form of LU and QR? It’s well known that LU and QR are two classic factorization algorithms in dense algebra system. LU provides a way to solve linear equation in the form of $AX = B$ without inversing A by using back and forward substitution. QR is widely used in solving linear least squares problems.

Because of the space limit of the paper, we just focus on introducing the application of the GPU-LU algorithm. GPU-LU is extended from the LAPACK LU routine, so it also keeps the decomposition in the form of $A = LU$, where L and U are stored back in the input matrix A. In order to solve the linear equation problem, the first task is to compose the lower triangular matrix L and upper triangular matrix U out of matrix A. U can be directly retrieved by setting the lower triangular part of matrix A at 0 and keeping the upper part as it is in A. However we can’t do the

same thing to retrieve L as the lower part is not only L but also involves permutation during the decomposition. The pivoting vector IPIV which is another output from LU decomposition can be used to permute the lower part of A to generate L. Since we assume that we are dealing with very large matrix, the composition of L and U can also be done on GPU. The second task is to do back and forward substitution to generate vector X.

In summary, the linear equation problem can be solved in the following steps:

1. run gpu_sgetrf for matrix A. Output is decomposed A and pivoting vector IPIV.
2. compose U by setting the lower part of A as 0 and keeping the upper part as it is in A
3. compose L by permuting the lower part of A using ipiv vector and setting the upper part at 0 and the diagonal at 1.
4. use forward substitution to solve LY=B* where B* is permuted by IPIV vector and Y=UX.
5. Based on the solved Y, use back substitution to solve Y=UX and output X

Fig 7. Solve linear equation problem by LU decomposition

Conclusion

We have demonstrated fast GPU-based implementations of LU and QR which are extended from LAPACK routines for LU and QR. We introduce in detail their design philosophy and analyze their key features which produce the high performance. Based on the experiment we conducte, we analyze the performance in terms of time, speedup and error. An obvious conclusion is drawn from the result, which is that the fast matrix-matrix multiplication algorithm generates fast factorization algorithm. The more matrix-matrix multiplication involved in factorization, the more advantage the implementation can obtain. In the end, we describe the application of LU and focus on analyzing steps after LU factorization is completed to solve the linear equation problem.

A random generator on GPU is also introduced. It generates some pseudo random number. We use Matlab functions to check their randomness while changing the size of the numbers.

It's the first time as we know that we provide detailed information for guiding developers on how to add customer functionality into Flagon.

Reference

Fan, Z., Qiu, F., Kaufman, A., and Yoakum-Stover, S. 2004. *GPU cluster for high performance computing*. In ACM/IEEE Supercomputing Conference 2004

Harris, M. J., Baxter, W. V., Scheuermann, T., and Lastra, A. 2003. *Simulation of cloud dynamics on graphics hardware*. In HWWS 03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, Eurographics Association, Aire-la-Ville, Switzerland, 92-101.

Rumpf, M., and Strzodka, R. 2001. *Using graphics cards for quantized FEM computations*. In Proc. Of IASTED Visualization, Imaging and Image Processing Conference (VIIP 01), 193-202

Bolz, J., Farmer, I., Grinspun, E., and Schroder, P. 2003. *Sparse matrix solvers on the gpu: conjugate gradients and multigrid*. ACM Trans. Graph. 22, 3, 917-924

Kruger, J., and Westermann, R., 2003. *Linear Algebra operators for gpu implementation of numerical algorithms. ACM Trans. Graph*. 22, 3, 908-916

Goddeke, D. 2005. *Gpgpu performance tuning*. Tech rep., University of Dortmund, Germany.

Galoppo, N., Govindaraju, N. K., Henson, M., and Manocha, D. 2005. *LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware*, SC05

Volkov V., Demmel, J. W., 2008, *Benchmarking GPUs to Tune Dense Linear Algebra*, SC2008

Choi, J., Dongarra, J. J., Ostrouchov, L. S., Petitet, A. P., Walker, D. W., and Whaley, R. C. 1994. *The design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines.* Engineering Physics and Mathematics Division Mathematical Sciences Section.

Dongarra, J., Duff, I. S., Sorensen, D. C., and Van Der Vorst, H. A. 1998. *Numerical Linear Algebra for High Performance Computers,* SIAM

Barrachina, S., Castillo, M., Igual, F. D., Mayo, R, and Quintana-Orti, E. S. 2008. *Solving Dense Linear Systems on Graphics Processors*, Technical Report ICC 02-02-2008

Baboulin, M., Dongarra J., and Tomov, S. 2008. *Some issues in Dense Linear Algebra for Multicore and Special Purpose Architectures*, Technical Report UT-CS-08-200, University of Tennessee, May 6, 2008 (Also LAPACK Working Note 200)

Internet Link: http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html

Nvidia Forum: http://forums.nvidia.com/index.php?showtopic=64545