CSCI-GA.3033-004
Graphics Processing Units(GPUs): Architecture and Programming
LU Decomposition using CUDA
Ying Qu

Abstract: Graphics processing units are developing very fast these years and becoming more and more popular not only restrict to the traditional graphics processing and computation but also for more common and general computation in many fields since its advantage of parallel computing to make the speed faster and the cost lower. These developments lead to many kinds of complex GPGPU applications with significant improved performances. LU decomposition is a linear algebra problem well used to be fundamental and basic step in many computationally intensive scientific applications, which always is the most time consuming step during the solution when the size of the matrix n is large enough. In this paper I implement three methods of the LU decomposition. Method one is using c language, method two is using cuda with global access, and method three is using cuda with shared memory and thread synchronization. These paper will write a survey summary of the previous work and make a comparison of the three methods I implemented and illustrate all kinds of optimizations I made.

Keywords — LU decomposition, CUDA, shared memory

I. INTRODUCTION

LU decomposition is one of the crucial computational operation in numerical computation which is a generally necessary procedure in a variety of scientific fields such as mathematics, physics, chemistry and astronomy etc. High qualified performance of LU decomposition will bring a lot benefit and may lead to a development leap of scientific research because of its practicability in a wide range of fields. Matrix factorization is a primary subject in applied statistics and linear algebra which is important to engineering and scientific fields. Analytic simplicity and computational convenience are normally considered in the matrix factorization. Back to reality from theory, it is usually difficult to solve a matrix computation problem explicitly without decomposition. An understandable method is to converting a difficult matrix computation problem into easier and fundamental tasks step by step, which is also the key principle of using LU decomposition.

The modern GPU is a powerful graphics engine as well as a highly parallel programmable processor providing fast arithmetic and memory bandwidth that are superior to traditional CPUs' capabilities [1]. The development of GPUs' highly parallel programming capabilities such as CUDA has resulted in a tremendous and significant

performance improvements in many ranges of scientific applications. The main tasks of my work are to find the shortcomings of the previous implementations of LU decomposition and to design a most effective way of performing LU decomposition on GPUs. I present my design in c language, cuda version one with global access and cuda version two with shared memory and thread sychronization.

Section 2 presents a detailed problem definition. Section 3 shows the survey result of previous implementations in literature. Section 4 is the design structure of my program. Section 5 is to illustrate the optimizations I have done. Section 6 is the comparison of the experimental results.Finally, section 7 is the conclusion.

II. PROBLEM DEFINITION

Specific topic:
Implement an n dimensional LU decomposition program in CUDA using shared memory and thread synchronization.
An LU decomposition for a matrix takes an input matrix A, and produces on output matrices L and U, where A=LU and L is lower triangular matrix with diagonal 1's, and U is a upper triangular matrix as follows (A = (ai,j), L = (li,j), and U = (ui,j)):

$$\begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ l_{2,1} & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n,1} & l_{n,2} & \cdots & 1 \end{pmatrix} \begin{pmatrix} u_{1,1} & u_{1,2} & \cdots & u_{1,n} \\ 0 & u_{2,2} & \cdots & u_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_{n,n} \end{pmatrix}$$

Matrix factorization is used for converting a difficult matrix computation problem into several easier tasks which can be used to solve those matrix computations. The LU decomposition represents a fundamental step in many computationally intensive scientific applications and it is often the costly step in the solution process because of the impact of size of the matrix. In algorithm, use the divide and conquer method to solve the problem.

Inputs:

a. User can define the dimension of the input matrix n, and then the system will randomly produces numbers through 0-99 to fulfill the matrix A to test for the output of the LU decomposition.
b. A text file of a n dimensional matrix A
c. The user is also assumed to have a GPU that supports the use of CUDA.

Outputs:

a. The result of matrix L and matrix U through iteration.
b. The sequential code of the LU decomposition problem
c. The cuda code of the LU decomposition problem by basically computing the kernel use through GPU.

d. The improvement of cuda code using shared memory and thread synchronization
e. A final write-up demonstrating how the GPU-based approach works, why it is useful, and how much of a speed improvement we obtain relative to the standard sequential approach.

Why is it suitable for GPU?
The traditional sequential method to compute the matrices L and U through iteration would be a huge amount of work and a waste of time. Through parallel computing and multithread operation, we can computing the each row of the matrix at the same time to significantly reduce the time cost.

III.RELATED WORK

A few research works on LU decomposition in GPU environment have been carried out but have not acclerated the problem significantly. In [2], they have developed and evaluated methods of loop processing, branch processing and vector processing. There result shows that (1) for dependent loops, the switching strategy using a render texture avoids copies in the VRAM, reducing execu- tion time by 50%; (2) there is a tradeoff relation between the CPU branch and the GPU branch, and the CPU branch provides higher performance for the decomposition of matrices larger than 512 × 512; (3) the efficiency of floating-point operations is at most 30%, and as Fatahalian et al. state for matrix multiplication, GPUs also require higher cache bandwidth in order to provide full performance also for LU decomposition; and (4) GPUs usually provide different decomposition results from those obtained using a CPU, mainly due to the floating-point division error that increases the numerical error with the progress of decomposition. They concludes that GPUs are not so suited well for LU decomposition. In [3], they have reduced row operations and matrix decomposition on GPUs. They have mapped the problem to GPU architecture based on basic calculation operations in matrix decomposition are primary row operations and they introduced new techniques such as streaming index pairs, efficient row and column swapping by parallel data transfer and parallelizing computation to utilized multiple vertex and fragment processors. They've compared the LU decomposition without pivoting, with partial pivoting and with full pivoting. The results of their implementations were comparable to the performance of optimized CPU based implementation and conclude that GPUs they used are not suited for the LU decomposition. In[4], they have implemented three blocked vaiants of Cholesky and LU factorizations with highly tuned BLAS implementations on a Nvidia G80 and an Intel processor and gain considerable performance. They have stated that simple techniques of padding, hybrid GPU-CPU computation and recursion have increased the performance of implementation. In[5], author have addressed some issues in designing dense linear algebra algorithms which is generally used for multicores.

IV. DESIGN

LU decomposition:

Suppose the input matrix is A, the method to decompose it into upper triangular matrix and lower triangular matrix is as follows,

```
for (k = 0; k < size; k++)
        {
                for (i = k+1; i < size; i++)
                {
                        A[i*size+k] = A[i*size+k] / A[k*size+k]; // update L

                        for (j = k+1; j < size; j++)
                                A[i*size+j] = A[i*size+j] - A[i*size+k] * A[k*size+j]; //
update U
                }
        }
```

This is a sequential right-looking algorithm[6], which can be divided into two parts: Traverse the matrix's column from left to right, to update the current row of the matrix, thus to get the current column of the matrix L. Then update the sub matrix to get the current row of the matrix U. Since it is independent to update the column of L and the row of U, therefore I can change my method as follows,

```
for (k = 0; k < size; k++)
        {
                for (i = k+1; i < size; i++)
                {
                        A[i*size+k] = A[i*size+k] / A[k*size+k]; // update L
                }

                for (i = k+1; i < size; i++)
                        for (j = k+1; j < size; j++)
                                A[i*size+j] = A[i*size+j] - A[i*size+k] * A[k*size+j]; //
update U
        }
```

For the outside loop k, the update of L and U depends on each other. U matrix must update after the L. However in the inner loop i, L matrix and U matrix are independent with each other. From this breach, we can use GPUs' parallel processing to implement the update of the inner loop i.

On GPU, there is shared memory on each Streaming multiprocessor (SM). During the execution of a GPU kernel, each CUDA thread-block requires some shared memory space. The total number of active thread-blocks depends on how much shared memory each thread-block consumes. For the right-looking update, each CUDA block requires a working set of three dense matrix blocks in shared memory: one for the matrix block in the current row of U, the other for the block in the current column of L, and the third for the block being updated.

Data Structure:

I use one-dimensional array of type float to store the input data. Because after LU decomposition of the matrix, it turns out to be lower triangle matrix L and a upper triangle matrix U, in order to save the space, L and U can be placed in the position of the original matrix.

The realization of GPU:

1) Allocate GPU buffer, copy the test data to GPU:

```
float *d_A;
cudaMalloc((void**)&d_A,sizeof(float)*wA*wA);
cudaMemcpy(d_A,A,sizeof(float)*wA*wA,cudaMemcpyHostToDevice);
```

2)How to partition data and how to decide on threads, blocks and grids
Update the iterative process of L and U matrix. Since the outer loop of updating L and U is relative, only the inner loop can be used with parallel processing. Here I used two kernels to control L matrix and U matrix separately.

a. The process of updating L matrix is to deal with a row of data, I use one dimension thread structure, and I assign the thread to be 512. If the size of the data in one row is larger than 512, use block control thread loop processing cycle manner, until all of the processing data is updated and completed.

b. To update the matrix U is to update the sub-matrix, which is to deal with two dimensional data, here I use

two dimension thread structure, and I assign the thread to be (32,32). If the size of the data is larger than 32*32, then use two dimensional block control thread loop processing cycle manner, until all of the processing data is updated and completed.

V. OPTIMIZATION

The most challenging part of this algorithm is that the size of the sub-matrix is reducing every time of the updating iteration,while at the same time, most of the sub-matrix elements are accessing when updating the upper triangular matrix. The access method of the sub-matrix elements directly influence on the performance especially speed of the LU decomposition, therefore, I have focused on the per-thread local memory, per-block shared memory and the global memory. Considering about the efficient memory access method, the core strategy to concentrate on is the use of block. Blocking strategy provides parallelism both among blocks and threads in a single block.

I use shared memory to store and update the column and row's data of function updateProcessKernel_shared, therefore when updating the sub matrix, the threads in the same block can share data from the shared memory. Since it is necessary to know the current position's row and column data, thus I use the former half part of the shared memory to store the data of the row, the latter to store the column data, which is shown as follows,

```
extern __shared__ float shared[];
        int AdjustX = -(blockDim.x*blockIdx.x+k+1);
        int AdjustY = -(blockDim.y*blockIdx.y + k+1);
        if (j==0)
        {
                shared[i+AdjustX] = d_A[i*wA+k];
                shared[blockDim.x+i+AdjustY] = d_A[k*wA+i];
        }
        __syncthreads();
```

Suppose the dimension of the sub-matrix is N, then in general we need 2*N*N times memory read, N times memory write. The advantage of using shared memory is that we only need 2*N times global memory read and N times memory write. It reduced the memory read times significantly when N is large enough. The data that originally need to read from the data memory changed to read from the shared memory data, which greatly improves the memory access speed and makes it more efficient.

VI. EXPERIMENTAL SETUP

1)A machine that can run GPU and CPU. The programming environment is cuda 5.5 version, computational ability 2.0 in linux or visio studio.

2) The testing matrix width is 128, 512, 1024, 2048, 4096, 8192.

3) To check for correctness: Since A is decomposed into LU matrices, which satisfies A=LU. Here I use a verify answer function to multiply L matrix with U matrix by comparing to the original A matrix. If the accuracy difference is within 1.000000E-01, I see it as correct.

V. EXPERIMENTAL RESULTS

I use the testing version code for CPU, GPU-GLOBAL and GPU-SHARED, to eliminate the time of printing output and verify the correctness of the answer. I test every N of every version for 10 times to get the average time. The time shown on the table below is the average time of computing the L and U matrix, excluding the time of verify answer and print out result.

MATRIX DEMENSION(N)	CPU TIME(MS)	GPU-GLOBAL TIME(MS)	GPU-SHARED MEMORY TIME(MS)
N=128	4015	3.350560	2.884676
N=512	252963	343.891541	260.993469
N=2048	16461210	24260.601562	22841.216797
N=4096	111774512	222886.234375	207354.250000

We can also get the output L and U matrix in a file which names LURes.txt from the SaveLuResult function.

VII.CONCLUSION

In this research, I have used the highly parallel architecture of GPU as a solution for the problem of LU decomposition with the support of CUDA. I have used the right-looking algorithm as the crucial algorithm for this LU decomposition problem which also satisfies the parallelism problem. As we can see from the result, GPU version saves a lot of time comparing to CPU version. Through parallel computing and multithread operation, we can computing the each row of the matrix at the same time to significantly reduce the time cost. The shared memory method saved a little time comparing to the global memory access. For further work, I want to try some other approaches by using other algorithm to get a comparison and find the best way to solve the LU decomposition.

VIII.REFERENCES

[1] D. Luebke S. Green J. E. Stone J. D. Owens, M. Houston, and J. C. Phillips, Gpu computing. Proceedings of the IEEE, 96(5):879–899, May 2008.

[2] F, Ino, K.Goda, M, Matsui, and K. Hagihara. Performance study of lu decomposition on the programmable gpu*. In Proceedings of the 12th IEEE international conference High erformance Computing, HiPC05, page 83–94, Washington, DC, USA, 2005. IEEE Computer Society

[3] N. Galoppo, N. K. Govindaraju, M.Henson, and D. Manocha. LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware. In Proceedings of the 2005 ACM/IEEE conference on Supercomputing, SC '05, Washington, DC, USA, 2005. IEEE Computer Society.

[4] S. Barrachina, M. Castillo, F. D. Igual, R. Mayo, and E. S. Quintana-Ortí. Solving dense linear systems on graphics processors. In Proceedings of the 14th international Euro- Par conference on Parallel Processing, Euro-Par '08, pages739–748, Berlin, Heidelberg, 2008. Springer-Verlag.

[5] M. Baboulin, J. Dongarra, and S. Tomov. Some issues in dense linear algebra for multicore and special purpose architectures.

[6]J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst, Numerical Linear Algebra on High-Performance Computers, 2nd ed. Society for Industrial Mathematics, January 1987.