Effective GPU Strategies for LU Decomposition

H. M. D. M. Bandara* & D. N. Ranasinghe
University of Colombo School of Computing, Sri Lanka
dinesh.methsiri@gmali.com, dnr@ucsc.cmb.ac.lk

***Abstract*— GPUs are becoming an attractive computing platform not only for traditional graphics computation but also for general-purpose computation because of the computational power, programmability and comparatively low cost of modern GPUs. This has lead to a variety of complex GPGPU applications with significant performance improvements. 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 this paper we implement three different variants of the LU decomposition algorithm on a Tesla C1060 and the most significant LU decomposition that fits the highly parallel architecture of modern GPUs is found to be Update through Column with shared memory access implementation.**

***Keywords*—LU decomposition, CUDA, GPGPU**

I. INTRODUCTION

High performance computing is relevant for a wide range of scientific fields including physics, astronomy, and chemistry. Numerical computation is a necessary procedure in all these scientific fields. LU decomposition is one such important computational operation. Matrix factorization is a primary subject matter in linear algebra and applied statistics which has both scientific and engineering significance [1]. The domains of matrix factorization normally involve two aspects called computational convenience and analytic simplicity. In real world, it is not practicable for most of the matrix computations such as matrix inversion, solving linear systems to be calculated in an optimal explicit way. Matrix factorization is used for converting a difficult matrix computation problem in to several easier tasks which can be used to solve those matrix computations.

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 [2]. Recent improvements of GPU's highly parallel programming capabilities such as CUDA [3] have lead to the mapping of wide variety of complex application with tremendous performance improvements. The goal of our work is to find out effective ways of performing LU Decomposition on GPUs. We present its design with different implementation approaches and measure the performance of those approaches.

The paper is organized as follows. Section 2 presents an overview related work. Section 3 describes the design approaches that compose our methods, Section 4 shows the implementations of those design approaches, Section 5 shows the results and performance studies obtained on our test environment and then finally Section 6 concludes the paper.

* Student Author

II. RELATED WORK

A few research works on LU decomposition with the support of highly parallel GPU environments have been carried out and had not been able to accelerate it much. In [4] Ino et al. they have developed and evaluated some implementation methods for LU Decomposition in terms of (a) loop processing (b) branch processing and (c) vector processing. In [5] Gappalo et al. have reduced matrix decomposition and row operations to a series of rasterization problems on GPU. They have mapped the problem to GPU architecture based on the fact that fundamental operations in matrix decomposition are elementary row operations and introduced new techniques such as streaming index pairs, efficient row and column swapping by parallel data transfer and parallelizing computation. Performance results of their implementations were comparable to the performance of optimized CPU based implementation and conclude that GPUs that they used are not suited for the LU decomposition. In [6] Barrachina et al. they have evaluated three blocked variants of Cholesky and LU factorizations with highly tuned BLAS implementations on a Nvidia G80 and an Intel processor and gain considerable performance. They have said that simple techniques of padding, hybrid GPU-CPU computation and recursion have increased the performance of implementation. Baboulin et al. [7] addressed some issues in designing dense linear algebra algorithms which are common for both multi-cores and many-cores.

III. DESIGN

LU decomposition is an algebraic process that transforms a matrix A into a product of a lower triangular matrix L whose elements are only on the diagonal and below, and an upper triangular matrix U whose elements are only on the diagonal and above. The main use of LU decomposition is to solve linear systems. Also it can be used to compute the determinant and the inverse of a matrix.

To illustrate the idea of solving a linear system by using LU decomposition, the system is expressed in matrix form as follows:

$$Ax = b \quad (1)$$

Where A is $N * N$ matrix with the coefficients of

the system, x is a column vector with the unknowns and b is also a column vector with the right-hand side of the system's equation. We can consider matrix A as two sub-matrices by making the LU decomposition of A and equation (1) can be rewritten as follows:

$$LUx = b \quad (2)$$

To solve (2), it is enough to compute two simpler linear systems as in (3)

$$Ly = b \quad (3)$$
$$Ux = y$$

Equations in (3) can be solved by forward and backward substitution of variables. For instance, in (3) as U is an upper triangular matrix, the coefficients of the first equation are all zero except one. So, the first unknown can be determined by a simple division of the right-hand side by the only non-null coefficient. After this calculation, we can proceed in a similar way by the rest of equations of the system. When the problem size n is sufficiently large, forward and backward functions take respectively less time than solving the system in straight forward way.

The question that would arise next is how the LU decomposition of a matrix A can be computed? The algorithm we are using in our project is the right-looking LU decomposition [8].

```
for i ← 0 to N
  for j ← i +1 to N
    Aji ← Aji / Aii   /*update L*/
    for k ← i +1 to N
      Ajk ← Ajk - Aik * Aji   /*update U*/
    end for
  end for
end for
```

Fig. 1 Sequential right-looking Algorithm

Figure 1 shows the familiar recursive right-looking algorithm that computes a block row and column at each step and uses them to update the trailing sub-matrix. In right-looking LU decomposition, the matrix is traversed by columns from left to right. At each step, computations are performed on the current column, and then updates to columns to the right of that column are performed immediately. Because updates are performed as soon as the current column is computed. There are dependencies between outer i loop iterations, and because of that we could not pass this in to GPU in a straight forward way (Figure 1). The dependencies in this algorithm are as follows

- The outer *i* loop iterations cannot be executed independently.
- The inner k loop iterations can be executed independently and those must be processed after completing the assignment operation.

Due to the dependencies mentioned above following reconstructions help to create a suitable parallel algorithm which is showed in Figure 2. That is by Loop decomposition method. By using that method updating L and U is decomposed in to 2 loops. Those loops have to execute in parallel.

```
for i ← 0 to N
  for j ← i +1 to N
    Aji ← Aji / Aii   /*update L*/
  end for
  for j ← i +1 to N
    for k ← i +1 to N
      Ajk ← Ajk - Aik * Aji   /*update U*/
    end for
  end for
end for
```

Fig. 2 Parallel right-looking Algorithm

Algorithm can be parallelized using two loops one for updating lower triangular matrix and one for updating upper triangular matrix. The main challenge of this algorithm is size of the sub-matrix which is involved in updating process getting reduced in the each iteration. In the same time most of the sub-matrix elements are accessed in the process of updating upper triangular matrix (Figure 3).

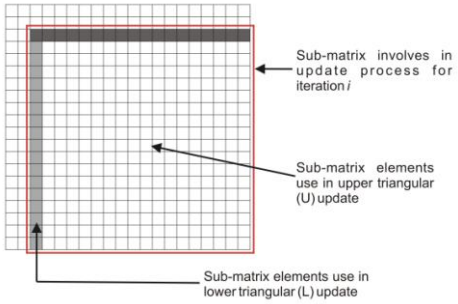

Fig. 3 Sub-matrix in i^{th} iteration

Access patterns of these elements have a high impact on the overall performance of the LU decomposition. In order to achieve better performance, it should be important to have an efficient memory access pattern. Therefore we have concentrated on the use of per-thread local memory (register), per-block shared memory, texture memory and the global memory and also attempted to minimize the communication between main memory and device memory. When we consider about an efficient memory access pattern, blocking becomes the core strategy. Blocking strategy provides two level of parallelism

1) Parallelism among threads within a single block
2) Parallelism among several blocks

The process which updates the lower triangular matrix, only accesses a single column of the sub-matrix. So the efficiency of the algorithm highly depends on the updating process of the upper triangular matrix. Therefore we have proposed 3 different design patterns with a blocking strategy to update the upper triangular matrix. Those are,

- Update through rows
- Update through column
- Update each element

IV. IMPLEMENTATION

In parallel implementation of LU decomposition the CPU is responsible for initializing the matrix and updating lower and upper triangular process management.GPU only concentrates on updating processes. And also CPU is responsible for allocating device memory, copy host variables to device variables and copy the computed device variables back to host variables.

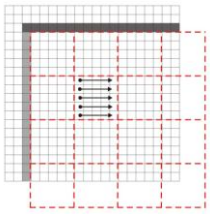

Fig. 4 Update through Row Strategy

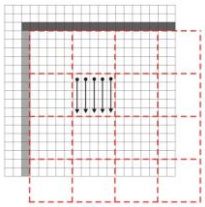

Fig. 5 Update through Column Strategy

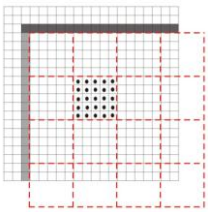

Fig. 6 Update each Element Strategy

If we look closely at the parallel algorithm in Figure 3 we can clarify that some of the data elements in the sub-matrix can be shared. When it executes the i^{th} iteration, i^{th} row and column can be shared.

A. Update Through Row

In this approach, sub-matrix elements which are involved in updating upper triangular process are divided into several blocks and it is also divided into rows. Each row in that block is updated by a different thread. Figure 4 shows the blocking strategy and how thread is allocated to a row in that block. This approach is implemented in two scenarios;

1) *Update through Row with Global Memory Access*: In this scenario there is no need of synchronization among the thread within the block. There is no data dependency among rows in the sub-matrix because each thread has to deal with global memory for all read and write operations and no shared data is kept on on-chip shared memory. Therefore each thread can execute independently.

2) *Update through Row with Shared Memory Access*: Our next level of Update through approach is based on effective use of both shared memory and global memory. Firstly, threads initialize the shared memory block using the values in the global memory. After that threads are involved to do the updating process through the row. In this approach we have to be strongly concerned about the synchronization among the threads in a block to avoid data conflicts inside the CUDA kernel [3].

B. Update Through Column

Also in this strategy a sub-matrix is divided in to several blocks. But the difference of this approach when compared to the previous approach is that blocks are divided into columns. Each column is updated by a thread. Figure 5 shows how this strategy works. Same as the update through row approach this strategy is also implemented in two scenarios;

1) *Update through Column with Global Memory Access*: Same as the Update through Rows with global memory access implementation, there is no data dependency among columns in the sub-matrix. With this kernel implementation, to update columns all the memory read and writes operations deal with the global (device) memory. As a result of that each thread executes their code without a synchronization barrier.

2) *Update through Column with Shared Memory Access*: This implementation is also based on the blocking strategy and collaborates with the memory use of global and shared memory. As we discussed in the Update through Rows with shared memory access implementation, the same set of data can be shared in this approach. So thread synchronization is a must.

C. Update Each Element

This is a simple approach when compared with the other approaches. Same as the previous approaches sub-matrix involves in the updating upper triangular process is divided in to several blocks and each element in a block is updated by a different thread. Figure 6 describes this strategy.

This strategy for LU decomposition also considers about blocking strategy and does not have special memory access pattern column through or row through. In this implementation, all the memory read, write operations are deal with the global memory and no shared memory involvement. As a result of that there are no synchronization barriers. This implementation is able to create more threads and utilized the GPU architecture in better way.

V. PERFORMANCE STUDY

We discuss the evaluation results and some discussion based on those results in this section. To compare the performance of three LU decomposition implementations on GPU, we have used a NVIDIA Tesla C1060 GPU which was connected to a PC with Intel Xeon X5570 Quad-Core processor running at

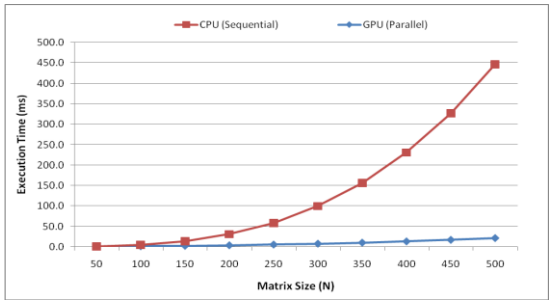

Fig. 7 Execution time comparison of CPU based implementation and simple (without blocking) GPU based implementation

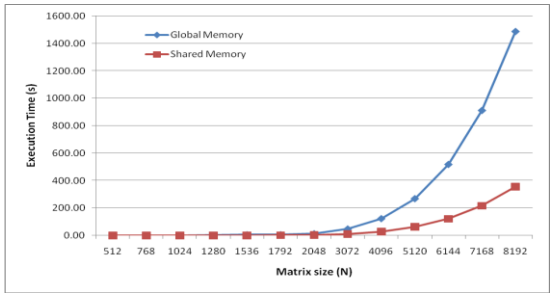

Fig. 8 Execution time comparison of Global memory access implementation and Shared memory access implementation of update through row strategy

2.93GHz with 4GB RAM and running on Linux. .*N***N* square matrices, where the *N* ranged from 256 to 8196 were used for experiments. All of these matrices were initialized to floating point numbers.

A. Results

Before moving in to the GPU strategies, as the first evaluation step, we compared CPU based sequential algorithm implementation and the non blocking simple GPU based parallel implementation. Figure 7 shows the performance comparison derived from those implementations.

1) *Global Memory Vs Shared Memory in Update through Row Strategy*: Here we move on to our first implementation approach Update through row. Under this approach we did two kinds of implementations as we have discussed. One is global memory based implementation and other implementation used a combination of shared memory and global memory. Comparison between these two implementations is shown in Figure 8.

2) Global Memory Vs Shared Memory in Update through Column Strategy: Our second implementation approach is Update through column strategy. Same as in the update through row approach there are two implementations based on the global memory and combination of global and shared memory. Figure 9 shows the comparison between those two implementations

3) Global Memory Vs Shared Memory in Update through Row Strategy: Blocking strategy and all of our three strategies use different kinds of memory

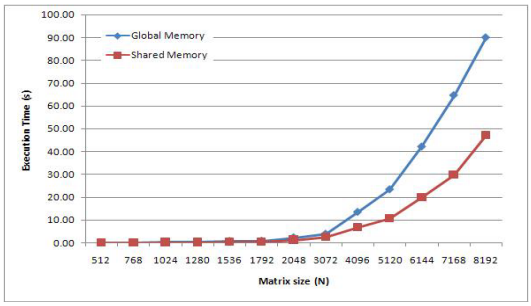

Fig. 9 Execution time comparison of Global memory access implementation and Shared memory access implementation of update through column strategy

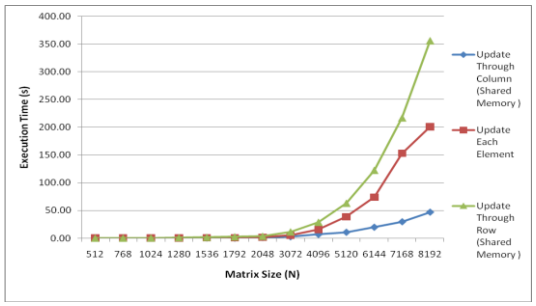

Fig.10 Execution time comparison of shared memory access implementations of update through column and column strategies and update each element implementation

access patterns. We compared each implementation within the strategy. Here we compared update through row with shared memory access implementation, update through column with shared memory access implementation, and update each element implementation to identify the better execution performance. Figure 10 shows the comparison among those implementations.

B. Analysis and Discussion

As the problem size scales up it requires a bigger memory and computational power to complete the job. According to the Figure 8 it is clear how important it is to map general purpose algorithms which need huge computational power for computation into GPU to gain better performance than on the CPU.

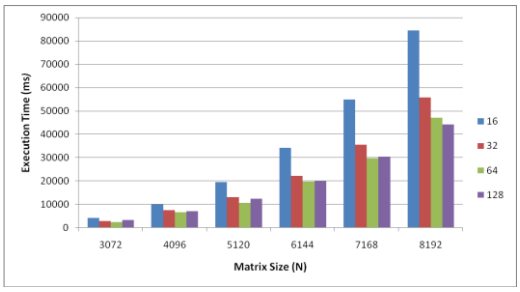

Fig. 11 Execution time comparison for block size in update through column with shared memory access

Even though CUDA facilitates to execute programs using thousands of threads, without an appropriate design which can utilize the environment it became a vain effort. In order to construct an efficient parallel algorithm for CUDA, we have to concern about the various memory levels which are provided by CUDA. Only using device memory which is implemented with

dynamic random access memory (DRAM), without efficient use of other memory levels may lead to poor performance. Rearranging the algorithm to make use of those memories may lead to better performance.

We have implemented both global memory access and shared memory access implementations for both update through row and update through column strategies. From the shared memory implementations we have tried to improve performance of our algorithm by using the on-chip shared memory. The reason for poor performance when accessing global memory is, it has long latency to read and write operations. Access speed of the shared memory is higher than that of global memory, but the problem is the limited size of the shared memory and how to use it effectively. Blocking strategy provides a solution for that and leads to better performance than global memory access implementation. From both Figure 9 and 10 we can illustrate the difference between device memory based and shared memory based implementations. When considered the simplicity of accessing the global memory it does not need extra thread synchronization. But in the shared memory implementations we had to be concerned about the parallelism of threads within a single block because of the complexity of the memory access pattern. But with the fast access of the shared memory it leads to a speedup above the global memory based implementation.

According to Figure 10 which compares shared memory based updating through row implementation, shared memory based updating through column implementation and updating each element implementation, we can see updating through column with shared memory access implementation shows the better performance among those implementations.

This is due to the global memory access pattern of that implementation. When accessing the global memory (DRAM) it prefers to access nearby memory locations rather than accessing a random sequence of locations. If a kernel arranges its data access pattern in that way, it achieves close to the peak global memory bandwidth. Update through column implementation provides this access pattern. As a result of that 32 threads (wrap) read their data from global memory with only one access. This technique is called memory coalesced technique [9]. Update through row implementation shows worst results because it has the unfavourable memory access pattern.

CUDA provides limited amount of on-chip memory. Therefore the programmer has to be careful not to exceed limits when using on-chip memories (shared memory and register memory). On other hand it limits the number of threads that be reside inside stream processor(SM). As a result of that it also limits the threads besides in multiprocessor (MP). We mentioned earlier that usage of local register memory can affect the number of threads inside a stream processor. According to Figure 11 we can see that optimal thread block size was 64 which were utilized our test environment effectively.

VI. Conclusions

In this research we have used the highly parallel architecture of GPU as a solution for the problem of LU decomposition with the support of CUDA. For that we have used right-looking algorithm as the benchmarked algorithm for LU decomposition which has satisfied the parallelism. We have proposed three strategies for implementation of LU decomposition 1) Update through row, 2) Update through column and 3) Update each element

We have tried our main three strategies with blocking to improve the performance. By comparing the global memory and shared memory implementations of first two approaches, the shared memory implementation has lead to a speedup than the global memory implementation.

Our final approach was on updating matrix element by using a thread for each element. By comparing the update each element implementation with shared memory based implementations of update through row and update through column, update through column obtained the best result and update through row showed the worst result. The key idea behind that is that update through column provides the best access pattern of memory by collaborating with hardware features.

References

[1] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, Numerical Recipes in C: The Art of Scientific Computing, Second Edition. Cambridge University Press, 1992, pp. 31-38

[2] 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.

[3] NVIDIA Corporation. NVIDIA CUDA C Programming Guide - Version 3.2, October 2010.

[4] 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

[5] 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.

[6] 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.

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

[8] 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.

[9] P. Micikevicius. Optimizing cuda [online]. Available: http://mc.stanford.edu/cgibin/images/0/0a/M02_4.pdf