Parallel GPU-based Algorithm for Matrix Computations

Tong Zhou, Mingzhou Jin, and Hui Dong

Electrical and Computer Engineering, University of Texas at Austin, Austin, TX, 78705

Abstract: In this report, we explore matrix operations in parallel on GPU including matrix inversion and LU decomposition in order to solve extremely large scale linear systems (matrix dimension up to $10000 \cdot 10000$). The inverse matrices are implemented on both CUDA and OpenCL via Gaussian elimination and modified Gaussian elimination, while LU decomposition is implemented through direct matrix multiplication using blocked and unblocked scenarios on CUDA. The performance of parallel algorithm on GPU shows a huge improvement over the sequential algorithm on CPU. We also find that CUDA has superior performance to OpenCL on NVIDIA GPUs and optimal block size exists which promotes the performance by 86.9% compared with unblocked LU decomposition on CUDA.

I. Introduction

Solving extremely large linear system a fundamental issue nowadays as explosive development of hardware enables people to explore much more complicated and accurate systems than before. For example, in SPICE and Cadence simulations, the VLSI chip simulation can be depicted as a huge set of linear equations which can be node equations. Other examples include solving stiffness matrix in finite-element method simulating large engineering problems, or solving nonlinear equations with high precision via high-order Newton-Raphson method [1, 2].

In recent years, parallel scientific computing has drawn broad attention as parallel algorithm greatly improves the performance and speed of solving linear systems. CUDA is a parallel computing platform and application programming interface (API) model maintained by NVIDIA. OpenCL is another popular open standard, which is friendlier to cross-platform parallel programming. These platforms allow users to employ graphics processing units (GPU) for general purpose data processing. We in the result section will discuss the performance between CUDA and OpenCL.

The most basic method to solve matrix equations is to use Gaussian elimination (GE) to find x in the following linear equations in matrix form $\boldsymbol{Ax} = \boldsymbol{b}$.

By iteratively doing elementary row operations, the matrix will finally transform into a reduced row echelon form. Currently, Gaussian elimination with pivoting (to prevent small denominator during operation) is the most efficient and accurate approach to solve linear system, with a computation cost of $O\left(N^3\right)$. With GE, we are able to perform most of the matrix operations including calculating determinants and solving linear equations.

On the basis of GE, further processing method like LU factorization, QR factorization, and Cholesky decomposition are developed to compute the inverse matrix A^{1}, to avoid repeated steps of GE for every b. The reason is mostly due to, in practical engineering applications, the matrix A is big and usually remains unchanged while the right hand side vector b keeps changing. The cost of factorization a matrix is $O\left(N^3\right)$. Once we have determined the factorization, the cost of solving $LUx = b$ is only $O\left(N^2\right)$.

The whole project is composed of two parts. In the first part, we achieved matrix inversion, determinant calculation and solving linear system with parallel computing, and compared the speed among sequential computing, CUDA and OpenCL. In the second part, we demonstrate LU factorization of matrix of different dimensions and block sizes. The remaining part of this report will start with the experimental setup for all the four types of matrix operations, which are then followed by the results and discussions about block-based algorithm and performance difference between CUDA and OpenCL.

II. Experiments

In this section we will discuss how we conduct inverse matrix, matrix determinant, solving linear systems, and LU factorization in parallel on GPU on Stampede TACC. There are two commonly used algorithms to solve all above problems, knows as matrix multiplication and Gaussian elimination (or Gauss-Jordan elimination in modified version) [1]. We implement remaining three matrix applications based on Gaussian elimination on CUDA and OpenCL and realize LU factorization using the definition of matrix multiplication to solve directly L and U on CUDA.

1. Solving Linear System via Gaussian elimination/Modified Gaussian elimination

In this section, we implement Gaussian elimination based algorithm [5] which is in completely reverse approach from what we do in LU factorization. In the experiment, we will compare the performance of CUDA and OpenCL by converting the same codes from the former version to the latter platform. In addition, we compared the speed of traditional GE and modified GE (also known as Gaussian-Jordan Elimination, GJE). Traditional GE only reduce the matrix to an upper triangular matrix, while GJE will reach a matrix form where each column contains a leading non-zero coefficient 1 and zeros elsewhere. Modified GE is considered to have more parallelism and no back substitution, and thus a much faster speed.

In the experiment, we employed inter-iteration parallel method which is called “partial pivoting” (compared to complete pivoting in which the maximum element of the whole remaining matrix is used as pivot). For iteration i, each row j in matrix A will perform the following calculation:

$$\mathrm{A}[j][\ ] = \mathrm{A}[j][\ ] \quad m[j][i] \cdot \ Pivot \tag{1}$$

The multiplier array $m[j][i]$ is determined before each iteration. The parallel algorithm perfectly fits CUDA/OpenCL structure. The following table presents the work load when we implement parallel determinant calculation on GPU.

Table 1: GE/GJE work load

Programs	**Inverse Matrix/Linear solver/Determinant**		
Algorithm	Gaussian elimination/Modified Gaussian elimination		
Framework	CUDA/OpenCL		
Mode	Sequential		Parallel
Dim of Matrix	5/100/500/1000/2000		5/100/500/1000/2000
LDA of Matrix	1000/2000/5000/10000	1000/2000/5000/10000	10000

2. Unblocked/Blocked LU factorization

There are primarily two reasons of implementing blocked LU factorization instead of unblocked LU factorization. One is the memory is always limited for extremely large matrix storage. The other one is that the latency associated with moving data is greatly reduced is they are moved as blocks, rather than as vectors or scalars [2].

We implement sequential LU factorization and unblocked CUDA LU factorization on different sizes of matrices up to $10000 \cdot \ 10000$, and investigate the effect of block size in blocked CUDA LU factorization based on the maximum matrix size. The work load in shown in Table 2

Table 2: LU Factorization Workload

Program	**LU factorization**		
Framework	CUDA		
Mode	Sequential	Unblocked	Blocked
LDA of Matrix	1000/2000/5000/10000	1000/2000/5000/10000	10000
Block size	1	1	10/50/100/200/500 /1000/2000

III. Results and Discussion

1. Solving linear systems via Gaussian Elimination/Modified Gaussian Elimination

We solve for x in $\boldsymbol{Ax} = \boldsymbol{b}$ where the size of A is up to $2000 \cdot 2000$ using 3 algorithms (sequential/GE/GJE) on 2 platforms (CUDA/OpenCL). We first compare the performance of sequential algorithm and parallel algorithm on CUDA, and then compare the performance of CUDA and OpenCL platforms as shown in Fig. 1. From the left panel, we can see a huge increase of speed of 88.33% when the matrix dimension is $2000 \cdot 2000$. The interesting point is that, even though we could implement parallel algorithm on OpenCL, performance in OpenCL is not better than sequential algorithm. The reason may be OpenCL, as a portable program, is not specifically optimized for NVIDIA GPUs and huge data movement between global memory and device memory would cost a lot more time than shared memory.

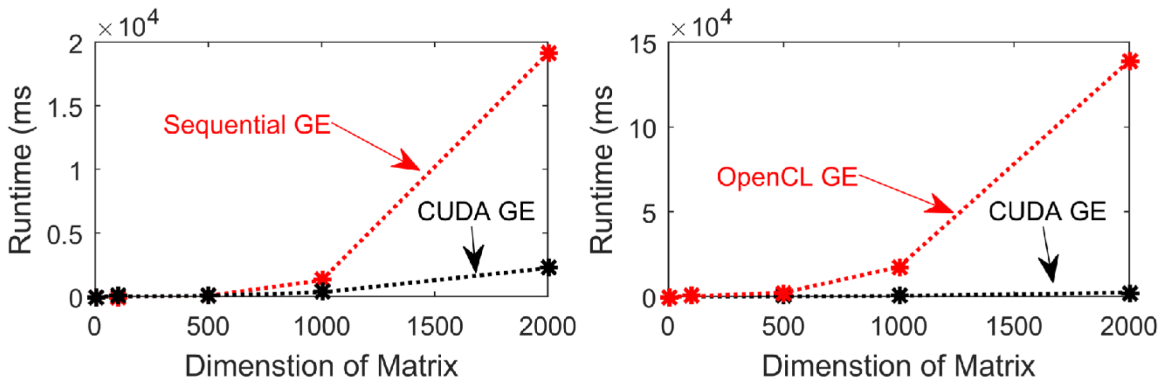

Figure 1: Left: comparison of Gaussian elimination in sequential algorithm and CUDA implementation. Right: comparison of Gaussian elimination on CUDA and OpenCL platforms.

We now check the improvement of parallel Gaussian elimination with modified version as we mentioned in experiment setup. The performance has been improved by 40.00% on CUDA platform and 37.04% on OpenCL when the matrix size is $500 \cdot 500$.

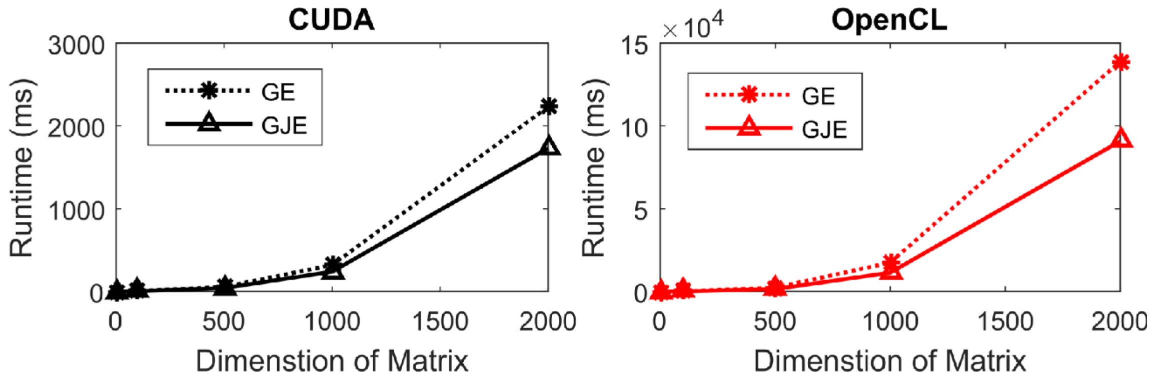

Figure 2: Left: speed up by modified Gaussian elimination on CUDA; Right: speed up by modified Gaussian elimination on OpenCL platform

As a byproduct of solving linear system, we can derive inverse matrix from the adjunct matrix. In this experiment, we again observe that OpenCL does not improve the performance of sequential algorithm.

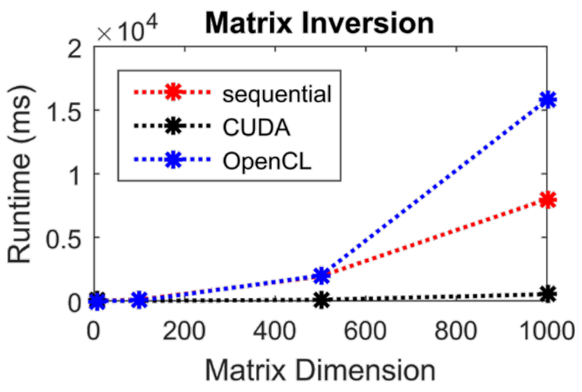

Figure 3: Comparison of sequential algorithm, GJE on CUDA, and GJE on OpenCL on performing matrix inversion with matrix size up to 1000x1000

We then can calculate the determinant of a matrix knowing its inverse. According to Cramer's rule, we have $\left(A[i]^{-1}\right)_{11} = \det A(i-1)/\det A(i)$ where $i = n, n-1, \ldots, 2$. Back solving these equations, we get

$$\det A = \left(\prod_{i=1}^{n} \left(A[i]^{-1}\right)_{11}\right)^{-1}.$$

2. Unblocked/Blocked LU factorization

We first implemented sequential LU factorization and unblocked CUDA LU factorization based on direct matrix multiplication. From Fig. 4, we can clearly see that the runtime is greatly reduced, especially when the matrix size is large. However, as the matrix size becomes larger, the speedup tends to become flat due to latency of huge data communication which will eventually limit or even deteriorate the performance of CUDA parallel program.

To reduce the latency of moving data, we implemented blocked LU factorization and investigated the effect of block size on program performance. From the results shown in Fig. 5, we clearly see there is an optimized value for block size around 200 with 86.98% improvement in speed. In real application, the runtime can be reduced by 70.42% if we simply change the block size from 1 (unblocked LU) to 10.

IV. Conclusion

We have implemented solving large linear system via Gaussian elimination/Modified Gaussian elimination method and matrix multiplication method both in sequential and parallel. We find that CUDA exhibits better performance than OpenCL possibly because of specific optimization of CUDA on NVIDIA GPU. We then further explore more advanced LU factorization algorithm on CUDA based on block concept, and improve General CUDA LU decomposition performance by 86.98% by reducing latency of moving large data.

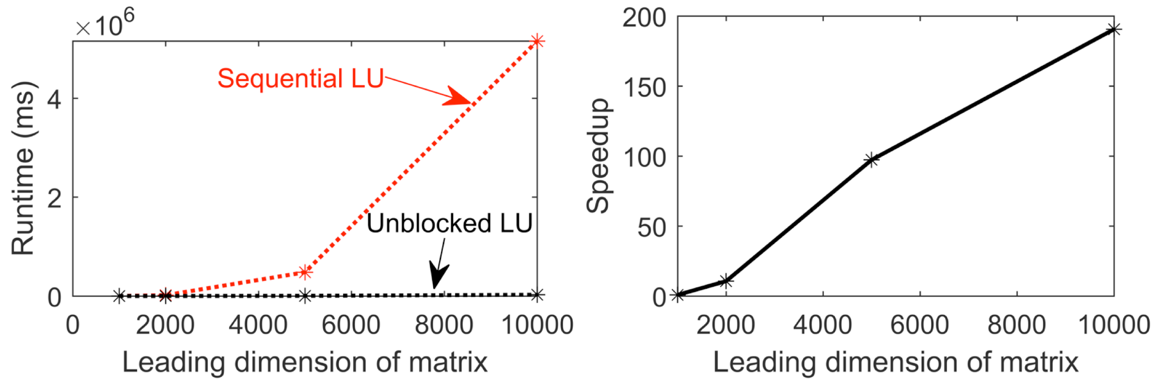

Figure 4: Left: Comparison of runtime between sequential and GPU parallel LU factorization on different matrix sizes; Right: Speedup with respect to different matrix sizes by implementing CUDA.

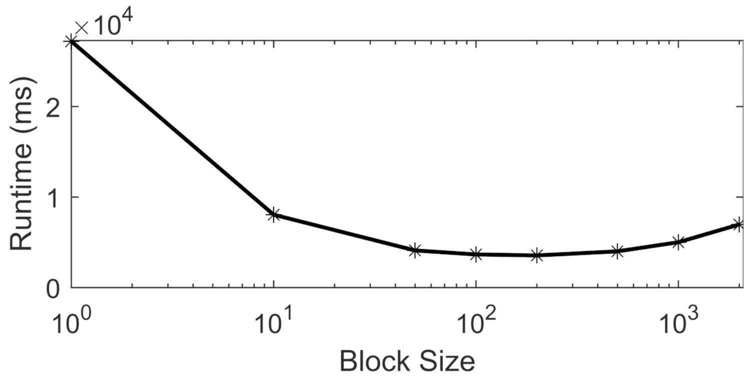

Figure 5: Great improvement of runtime by using blocked LU factorization. The optimized block size appears around 200.

We specially thank to Dr. Vijay Garg for inspiring instructions throughout this semester. We also acknowledge useful discussion with Miao Qi and Asad Malik during the progress of the project.

References

[1] V. Kindratenko, Ed., Ch.1, *Numerical Computations with GPUs*. Cham: Springer International Publishing, 2014.
[2] Nachieket Kapre, Andre DeHon, *Performance Comparison of Single-precision SPICE Model-Evaluation on FPGA, GPU, Cell, and multi-core Processors,* 2009
[3] A. Petitet, H. Casanova, J. Dongarra, Y. Robert, and R. C. Whaley, "LAPACK Working Note 139 A Numerical Linear Algebra Problem Solving Environment Designer's Perspective," 1998.
[4] G. Sharma, A. Agarwala, and B. Bhattacharya, "A fast parallel Gauss Jordan algorithm for matrix inversion using CUDA," *Computers & Structures*, vol. 128, pp. 31–37, Nov. 2013.
[5] http://www.hpcc.unn.ru/mskurs/ENG/DOC/pp09.pdf