LU and QR Factorization on GPUs for High-performance

A dissertation submitted for the Degree of Master of Science in Computer Science

M.A.C. AMODHA

University of Colombo School of Computing

2019

DECLARATION

The thesis is my original work and has not been submitted previously for a degree at this or any other university/institute.

To the best of my knowledge, it does not contain any material published or written by another person, except as acknowledged in the text.

Student Name: M.A.C. Amodha

Registration Number: 2016/MCS/004

Index Number : 16440041

Signature: Date:

This is to certify that this thesis is based on the work of

Mr./~~Ms~~. M.A.C. Amodha

under my supervision. The thesis has been prepared according to the format stipulated and is of acceptable standard.

Certified by:

Supervisor Name: Mr. K.P.M.K. Silva

Signature: Date:

ABSTRACT

GPUs have become very interesting, especially with the General Purpose Graphics Processing Units. With the ability to program the GPUs, their computation capabilities with the processing power and their competitive low cost have enabled the development of numerous kinds of interesting GPGPU application programs resulting in substantial accomplishments in terms of the performance.

The LU and QR factorizations represent an underlying process of a large number of scientific application programs with complex and computationally expensive modules. But in here, the solution process has a high impact on the matrix size for the performance because of the costly computations.

Proposed methodology for the GPU only LU and QR factorization algorithms were implemented using block matrix factorization where the input matrix is considered as multiple matrices when performing the factorization steps. GPU only factorization algorithms are implemented on a NVIDIA MX130 GPU. For LU factorization, the suggested GPU only algorithm implementation starts to perform well with the square matrix 6144 and upwards. With the suggested GPU only QR factorization implementation, it was possible to execute matrix sizes up-to 1024x1024.

The evaluation of the implemented algorithms clearly depicted that the output matrices are accurate when computed and compared with the input matrix. Finally, it is believed that the work accomplished through this research work has facilitated for the betterment of the learning community as well as the parallel computing and computer science research community.

ACKNOWLEDGEMENT

First of all, I am so much grateful for my supervisor Mr. K.P.M.K. Silva who provided the guidance to carry out this research work successfully from beginning to the end and for introducing me to this revolutionary parallel computing technology for GPUs. Also, the dedicated lecturer panel of University of Colombo School of Computing, who taught us through the degree program, would be acknowledged for the knowledge and wisdom I have gathered while being a student there and giving me the opportunity to apply the knowledge which I gained through the courses.

Also, I would like to acknowledge online NVIDIA Developer Community and Stack Overflow Community for sharing their experiences and providing suggestive solutions for the technical problems I came across. Then it is necessary to mention the name of NVIDIA Corporation for creating the parallel computing platform, namely CUDA (Compute Unified Device Architecture) as well as for creating the application programming interface model for CUDA, and also for sharing their research work publicly to be used by the others.

Finally, I would like to convey my thankfulness to my family who helps me and bear with me to make this one-year-long research project to successfully culminate.

TABLE OF CONTENTS

TABLE OF FIGURES

LIST OF TABLES

1 INTRODUCTION

1.1 Motivation

General Purpose Graphics Processing Unit (GPGPU) has a high processing capability and the large number of cores inside the GPU has enabled parallel execution with high performance for computer applications [1]. Most of the applications have not taken advantage of the GPU cores completely. So it is interesting to see how the GPU is completely used in-order to achieve high performance in an efficient way.

High-performance GPU only execution of Cholesky Factorization [2] has been successfully implemented by Azzam Haidar and others. In their research paper, they have raised the importance of the development of other highly required factorization routines, such as the QR and the LU factorization as their future directions. Therefore, there is a need for the development of high-performance LU and QR factorizations implemented fully on GPUs. Currently, there are algorithms available for LU and QR factorization which run on multi-core CPU processors and hybrid CPU-GPU processors [1] with also some GPU only implementations. But there is not much literature available to accelerate the performance of these factorization algorithms. So the motivation of this research project is to *implement high-performance LU and QR factorization algorithms which provide better results than the existing factorization algorithms*.

1.2 Project

1.2.1 Problem Domain

The emerging accessibility of the advanced-technology along with the advanced-architecture computers incorporates a vital result on all domains of scientific computation, together with algorithmic program analysis and software development in numerical algebra. Linear algebra particularly, the answer of the linear systems of equations lies at the center of furthermost calculations in scientific computing [3].

Numerical linear algebra is known as the study of algorithms related to mathematical questions for carrying out linear algebra computations which typically includes matrix-matrix operations on computers in order to provide accurate and approximate answers. It's usually a basic a part of computer science domains, like computational fluid dynamics and lots of different areas [3]. Those type of software depends deeply on the analysis, development, and implementation of progressive algorithms for addressing numerous numerical algebra complications in terms of a solution with the available numerical techniques. Problem is commonly converted and reduced to a problem of linear equation systems. Because of this reason, the solution is normally represented in the form of matrices [4].

1.2.1.1 LU Factorization Method

LU factorization decomposes a matrix into a product of two matrices. The first matrix as a lower triangular matrix and the other matrix as an upper triangular matrix. Sometimes the product of lower and upper triangular matrices includes a permutation matrix likewise. In order to solve systems of linear equations or to calculate the determinant of a matrix, LU factorization is used in numerical analysis. However, LU method is much more advanced and complex when compared with the Gaussian method but more efficient for solving an equation system [5].

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$$

Figure 1.1 LU Factorization in the Form of A = LU

As Figure 1.1 indicates, LU factorization is able to resolve a system of equations with the steps listed below.

Set up the equation as shown in Equation 1.

$$Ax = b \qquad \text{Equation 1}$$

As the next Step, find LU factorization for matrix A and the result will produce the Equation 2.

$$(LU)x = b \qquad \text{Equation 2}$$

Let the value of y be according to Equation 3 and solve Equation 4 for y.

$$y = Ux \qquad \text{Equation 3}$$

$$Ly = b \qquad \text{Equation 4}$$

Take the values for y and solve Equation 3 for x. This will give the solution to Equation 1.

1.2.1.2 QR Factorization Method

Any real square matrix A can be factorized into a product of an orthogonal matrix Q and an upper triangular matrix R as shown in Equation 5. In numerical linear algebra, QR factorization is regularly used to solve the linear least squares problems and also for a particular eigenvalue algorithm QR factorization is taken as the foundation [6].

$$A = QR \qquad \text{Equation 5}$$

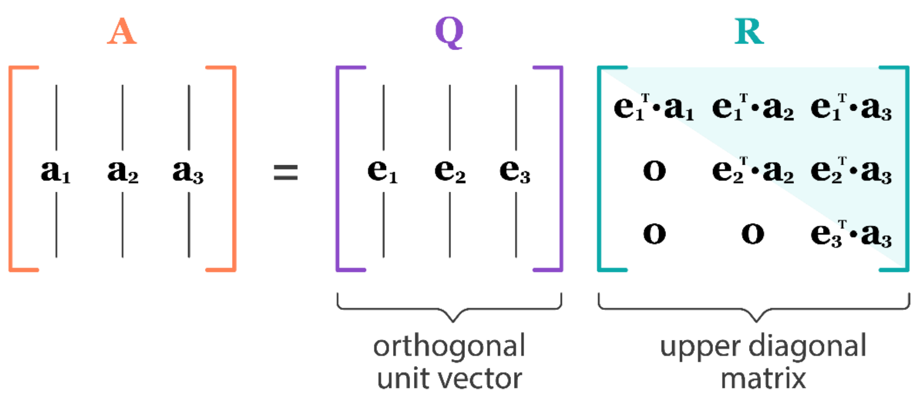


Figure 1.2 QR Factorization in the Form of A = QR

QR factorization is displayed in Figure 1.2. In the QR Factorization, R matrix can be computed using Equation 6 and according to Figure 1.2,

- A is a square matrix
- Q is an orthogonal matrix
- R is an upper triangular matrix.

$$R = Q^T A \qquad \text{Equation 6}$$

1.2.2 The Problem

In order to reach high performance through parallelism, there are some available architectures and techniques [7] and with these techniques and architectures, there are several types of drawbacks which directly has an effect on the performance. Tuning challenges occur when a computer CPU is having a slow processing power or when the kernel design is complex. Because of this reason, the GPU has to wait a long time causing expensive CPU-to-GPU communications which directly causing reduced performance [2]. In most of the hybrid factorization algorithms, panel factorization is computed on the CPU. So the GPU has to wait for that calculation to finish to start its calculations

[2]. Another reason for not getting high performance in factorization is using complex algorithms to perform the calculations. Complex algorithms have a large number of codes to be executed and a high number of kernel calls will be causing reasons for performance decrease [2], [4]. It is difficult to reach high performance from an algorithm with the presence of these issues.

1.3 Exact Computing Problem

The exact computing problem can be presented as sub-questions as shown below:

a) Find currently existing GPU only LU & QR factorization algorithms and the gaps of those implemented algorithms.

b) What are the ways of implementing LU and QR factorizations for high-performance completely on GPUs (GPU only)? [2], [8], [9]

c) How to perform both panel factorization and trailing matrix update in the GPU, using different or same GPU streams without affecting performance? [2], [10]

d) How to improve the algorithm/application that helps boosting the performance in following paths?
    i. *Algorithmic optimization* path [2]
    ii. *Kernel optimization* path [11]
    iii. *Implementation design* path [12]

1.3.1 Research Contribution

By conducting this research, the following contributions are offered to the field of computing and computer science:

Improved and resource efficient LU and QR factorization algorithms to run only on GPUs to accomplish high-performance.

Performance analysis of the implemented LU and QR factorizations which can be used for comparison against future developments.

Developers to use the expected findings of this research for their own application implementations.

1.3.2 Aims and Objectives

Aim of this research is to develop a high-performance GPU only Implementation for LU factorization and QR factorization. And the objectives of this research are listed below.

<u>Objectives</u>

- LU and QR factorizations should be able to execute successfully in GPU only implementations.
- Existing expensive communication should be removed in CPU-to-GPU interactions.
- Tuning problems also should be removed.
- Should be able to reach high performance in LU and QR factorizations.

1.3.3 Scope of the Research

In this research, LU factorization and QR factorization is only going to be considered. First, we shall attempt to implement LU factorization on a GPU only cluster and then move to implement QR factorization. Both LU and QR factorization algorithms/applications are to be executed on a GPU only cluster. This includes the panel factorization as well as trailing matrix update executed on GPU cluster. Tuning challenge problems and CPU-to-GPU expensive communication problems are going to be discussed in this research. All the work to be performed in a NVIDIA™ GeForce MX130 GPU (mid-range performing GPGPU). Factorization algorithms will be implemented using Compute Unified Device Architecture (CUDA) platform for the selected factorization algorithms. GPU only Cholesky factorization algorithm is successfully implemented in a NVIDIA GPU using CUDA platform [2], because of that reason CUDA programming platform will be used to implement LU and QR factorization algorithms for GPU only execution in this research

2 LITERATURE REVIEW

2.1 Area of Study

GPUs containing immensely parallelly executable computing processors which are programmed in C programming language and with the extensions of the C programming language. In order to program these parallel processors, it is not compulsory to aware of the graphics algorithms or terms related to the terminology. But with the knowledge and with the understanding of these algorithms, it is much more easy to identify the pros and cons with the relevant computational patterns. With the help of the past, it is possible and able to clarify the explanations about architectural design selections of the GPUs in the present, which includes vastly multithreading, highly parallel structure and bandwidth-centric memory interface design. Understandings about the historical advancements will also likely to give the framework for the future direction and projection of GPUs as computing devices [1].

2.1.1 Direct Memory Access

Direct memory access is used between a CPU and a GPU to perform the data copy operations. This process needed a dedicated memory in DRAM and an indirect way of allocating the memory by an application [1]. An especially dedicated hardware mechanism is now included in the modern computer systems to transfer data between the input/output device and the DRAM of the system. This mechanism is named as direct memory access. In this mechanism, the operating system performs an operation established by:

- the starting address of the data in the Input/Output device buffer memory
- the starting address of the DRAM memory
- number of bytes to be copied
- the direction of the copy.

Following advantages can be gained using this direct memory address [1] mechanism:

- Execute input/output independent programs in the CPU, when the direct memory access mechanism is copying the data.
- Copy the data between devices at a rapid speed than a normal processor by using an especial hardware mechanism.

2.1.2 Data Parallelism

Data parallelism can be used in this research to perform calculations of the matrices which are not related to the particular set of columns or rows. So those independent rows and columns can be computed parallelly. Parallelization through multiple processors in the environment of parallel computing is identified as data parallelism. The data has been distributed through multiple nodes or threads, which is operated in parallel. Related data is processed in parallel by working on each element and the elements are stored on regular data structures such as matrices and arrays. When evaluating the performance of the programming model in terms of efficiency and effectiveness, the locality of the data references plays a significant role and such data is relying on the size of the cache and the memory allocation defined by the application program [13].

2.1.3 Task Parallelism

LU and QR factorization algorithm codes are planned to separate as independent tasks so that the different processing cores can engage in different smaller tasks. Because multiple independent tasks can be utilized widely in parallel programming. Normally task parallelism is achieved by dividing tasks into smaller independent tasks of an application. When there are two independent tasks exists, task parallelism also exists. When an application gets larger so does the number of independent tasks, as a result of that, a large number of tasks can be executed parallelly. So accomplishing the performance goals on parallel programming applications depends heavily on the task parallelism and plays a key role with the efficiency [1].

2.2 Literature Review

2.2.1 Numerical Linear Algebra

The developments in linear algebra are designed according to the advanced-architecture of the computers. Scientific applications and engineering applications are widely using numerical linear algebra operations. There is a standard for the basic linear subprograms (BLAS) in order to perform the numerical linear algebra operations. These standard libraries of linear algebra functions consist mainly of three levels. When the level of the linear algebra function increases so does the number of operations performed by the related function accumulate accordingly. D. B. Kirk and W. W. Hwu have shown an example of a vector addition which is a level-1 function. Matrix and vector operations are performed in the level-2 functions and these operations using vectors (x,y) and scalars (α, β) along with the matrices. They have raised the importance of these BLAS functions for solving linear systems and eigenvalue analysis since these functions are used as building blocks of the numerical linear algebraic functions. Kirk and Hwu have identified that different BLAS function implementations will perform in different ways in both parallel computers and sequential computers [1].

2.2.2 Matrix Computation

The focus in [3] was to analyze the impact and the performance of the dense and sparse matrices. Jack Dongarra and Victor Eijkhout have developed templates for sparse matrix computations. They believed that modern computers with advanced-architecture have a high impact in the area of scientific computation along with the numerical linear algebra software development research area. This article has discussed the numerical linear algebra design in order to make full use of the advanced-architecture computers with the proposed developments. Jack Dongarra and Victor Eijkhout have focused on four basic concerns [3] shown as following:

- The inspiration for the work
- Define standards and implement the standards (to be used in linear algebra libraries)
- Algorithm design (design concept along with the parallel implementation)
- Future directions for the research.

They have started to improve the development of the sparse matrix computations templates and they want to apply these templates for the dense matrix computations as the future work [3].

2.2.3 Multi-core LU & QR factorization (CPU-GPU)

In [4] Caner Ozcana and Baha Sena have presented an algorithm to deal with the dense linear systems in the CUDA programming platform. Because of the high arithmetic throughput of GPUs, Caner Ozcana and others were able to strengthen the performance with a suitable data representation along with the reduced row computations on GPU. But the main concern was a comparison of diverse systems which consists of numerous GPUs and CPUs for different linear systems in terms of the runtimes. They have evaluated the performed algorithms and what they see was better performance is obtained with GPU computing. The application that Caner Ozcana and Baha Sena developed as the solution of linear equation systems, consists of a significant performance improvement. They have tested it on core2duo computer which includes 16 CUDA cores on the first time and they were able to gain a 431% performance rate on a linear equation system from the GPU compared with the CPU. They implemented LU numerical linear algebra routine that consists of appropriate data representation with a GPU accelerated implementation. The implementation has focused on reducing the row computations on GPU and they have provided significant performance improvement on sparse linear systems and suggested that the same approach can be used it to explain dense linear system [4].

Radomir Stanković and others have presented [8] five different LU and QR factorization implementations. They have analyzed the efficiency of CPUs and GPUs for the runtimes and the developments were carried out using:

- Intel MKL
- Eigen C++ library
- MATLAB

These implementations were performed on a multi-core CPU by them. Rest of the implementations have been handled on a GPU with the usage of NVIDIA cuSOLVER library in the CUDA platform and with Parallel Computing Toolbox in the MATLAB platform. Results were generated using inputs as single and double-precision matrices with the floating-point representation where the elements are generated randomly. This research article [8] has shown that the both GPU implemented LU factorizations were achieved the best performance when compared with rest of the implementations and those matrices were able to fit into the global memory of the GPU. Intel MKL implementations were identified as the fastest method for the LU factorization with larger matrix sizes and for the QR factorization with all the matrix sizes that have executed in this research.

Robert Andrew, Nicholas Dingle has analyzed that many of the performance issues of QR factorization were associated with kernel invocations of high frequencies [11]. Using the CUDA development platform, they implemented four GPU updating algorithms and identified that for certain matrix sizes those implementations perform better than the GPU only QR decomposition. A high number of kernel calls have a direct association for the performance drawback and they suggested to increase and improve the number of rows in a strip with a reduced number of kernel invocations and apply several depending rotations in a thread block inside the kernel with loops and synchronizations. They also pointed out that with the use of NVIDIA Kepler architecture's dynamic parallelism, kernel invocation overheads can be reduced on the GPU. Also, Robert Andrew and others have discovered that in some circumstances updating is faster than the full factorization and in some, where it is not. In this article, algorithms are consists of operations and closed-form expressions are used as the foundation to determine the runtimes in the GPU implementations.

Peng Du and others [9] research on integrating the CUDA computing directly into the ScaLAPACK framework, and speed-up the LU and QR routines for a certain level by carefully managing the GPU-CPU data transfers. But Peng Du and others were not able to remove the CPU-to-GPU expensive communications. Their main focus was to convert most of the ScaLAPACK routines to support GPU computing so when GPUs are presented, application codes that already utilize ScaLAPACK framework are able to reach some sort of an automatic speedup. They suggested that it is beneficial to keep data onto GPUs as much as possible. They showed that for LU factorization where pivoting forces more frequent data transfer, minimizing the data amount helped largely to reduce the performance impact. Peng Du and others have identified to take multiple GPUs per node into consideration as their future work and to convert more algorithms. They have shown the direction to conduct larger scale experiments to further confirm the design in the future.

In [12] also describes an implementation of a parallel LU decomposition on GPU cluster for dense matrices. E. D’Azevedo and others have developed a software to reach the high performance by increasing the software complexity, integrating magmaBLAS implementation to the software and to use a left-looking out-of-core algorithm when the available memory on the GPU device is lower than the problem size. But they were not able to avoid the tuning challenges of slow CPUs along with the low CPU-to-GPU bandwidth which has disturbed to reach a certain good level of high performance. They have identified, optimizations that may need to be included such as finding asynchronous operations to transfer the data on CPU and GPU devices, tuning separately matrix block size in ScaLAPACK library and development of the look-ahead computations in the algorithm to minimize the runtime of the LU factorization by considering the critical path as their future work.

Yulu Jia and others did [10] research on the LU factorization on the shared memory environment and proposed a multi-GPU, multi-core hybrid LU decomposition algorithm which supports both multiple GPUs and CPUs. This hybrid algorithm works with static scheduling and dynamic scheduling. But the suggested LU decomposition algorithm has used some CPU cores to perform the panel factorization and to update the trailing submatrix, remaining CPU cores along with all the available GPUs cores has been used. Since panel factorization is done in the CPU, hybrid algorithms overlap with the CPU work, and the expensive CPU-to-GPU communication is also a drawback for the performance. In this article [10] Yulu Jia and others have shown that the main concern is the speed and the time of the execution when solving the LU factorization for a large matrix size and they have planned to avoid the unnecessary data copying between the CPU and the GPU by using GPU non-resident memory technique as their future work.

Zhongchao Lin, Yan Chen and others have shown that faster speed can be reached with the GPU based two-level out-of-core algorithms for the situations with large element method. An airborne array problem is solved in this paper on a CPU/GPU hybrid cluster with the following computer specifications:

- 128GB RAM
- 10GB GPU memory
- 1TB storages of HDD

With these computer specifications, they were able to achieve a speedup of 1.6 times against the implemented parallel CPU version. The same technique can be used for on-board antenna systems which consist of complex and larger platforms to increase the performance in finding the radiation patterns as described in this paper [14]. With the involvement of CUDA and MPI frameworks, suggested implementation were able to execute on CPU/GPU hybrid cluster. Physical memory and GPU memory bottlenecks in the electrically large complex problems are addressed with the designed two-level out-of-core algorithms. Asynchronous communication has used by Zhongchao Lin and others to allow communication and computation overlap in the algorithms. They have shown that when compared with the traditional out-of-core LU solver, the two-level out-of-core LU solver performed better with 1.6x times for the large problems which having difficulty to fit in the physical or GPU memory [14].

In [15], Azzam Haidar, Mawussi Zounon, Ahmad Abdelfattah, Stanimire Tomov and Jack Dongarra have presented an improved GPU kernel for very tiny matrix operations which had a significant speedup, better than the vendor libraries. And also Azzam Haidar and others have discussed that the design of the GPU kernels is the reason for the performance decrease to the small matrices algorithms. They proposed the strategies and the analysis of the respective algorithms in order to achieve the complete utilization and the performance from the GPU. Methodology and theoretical analysis also have been developed by them for tiny matrices to gain better performances. The suggested methodology described using LU and Cholesky decompositions as test cases to show that the hardware performance near to the theoretical upper bound can be achieved. Highly optimized GPU kernel design for the novel algorithms was investigated by them and this particular GPU kernel is used for undersized baches of LU and Cholesky decompositions. The motivation for this research is the demanding need in the areas such as astrophysics applications in the scientific simulation domain. Following Methods are incorporated in the proposed design for [15], [16]:

- Register blocking
- Ideal memory traffic
- Tunable concurrency.

Sencer Nuri Yeralan, Timothy A. Davis and others have done research on sparse matrix decomposition which including a combination of both regular and irregular operations and computations. They have stated that gain high-performance on the available cores in the GPGPU was very challenging and they have addressed this challenge with a multifrontal QR decomposition concept and the performance achieved is considerably high with compared to a highly enhanced multi-core CPU. All the communicated data is stored on GPU and a lot of frontal matrices were decomposed concurrently on highly parallel nature and the algorithm has extended to support more parallelism. The communication-avoiding QR decomposition supports further parallelism with the dense matrices and the sparse multifrontal method supports further parallelism with the sparse matrices [17].

In [18], Cheng Chen and others have highlighted that the dense LU factorization is a serious factorization algorithm that broadly used in the problems of dense linear algebra. According to them, Hybrid LU factorization implementations designed in a way to make full use of the heterogeneous systems. But the available heterogeneous algorithms are usually based on CPU and those algorithms mostly rely on CPU cores and perform a large number of data transmissions through the PCI bus. Because of this reason, performance efficiency and resource efficiency of the complete computer system will be decreased. But according to this paper, they have described an implementation of coprocessor-resident LU factorization in order to increase the performance efficiency along with the energy efficiency by freeing the CPU with the massive computation operations and by removing the data transmission through PCI bus. In order to preserve efficiency, they have carried out improvements to CPU operations, MPI operations and to coprocessor operations and all the improvements were performed on a supercomputer and the output has shown that their LU implementation can be reached high performance and it is possible to avoid the barriers of the energy and the performance efficiency.

Felix Loh, Parameswaran Ramanathan and others have identified and shown that GPUs are vulnerable to burdens like alpha particle strikes and power fluctuations when trying to minimize the transistor feature size with the intention of improving the methodology along with the technical aspect. So that they raised the importance of technique which is able to assure the accuracy of the operations even in the middle of a fault. They have developed and analyzed three fault-tolerant schemes for QR factorization, and also they have presented a technique which is able to avoid the errors and faults with having different time spans only for NVIDIA GPUs namely as transient fault injection technique. They showed that the technique in this research is comparatively low cost, has a better ability to scale and holds a good success rate from this research [19].

With a minimized communication, a dense vector set can be orthonormalized by and single value QR decomposition. Single value QR decomposition has shown a remarkable performance when compared with the available orthogonalization algorithms. In the orthonormalization algorithms, communication is the place where the most expensive computations occurred other than the arithmetic computations. Ichitaro Yamazaki and others have studied the steadiness and efficiency of different Single value QR decomposition developments on multi-core CPUs along with a GPU in this research. Their focus was with the triangular solver for the dense vectors because it performs the most of the decimal computations of the single value QR decomposition. As a component of the study, they have examined a versatile modified version of single value QR decomposition. It has the choice to either expand the direction of the orthogonal error or to use the triangular solution at runtime [20].

Wei Tan, Shiyu Chang and others have shown in the [21] that the matrix factorization has a high potential in the areas of feature extraction, word embedding, collaborative filtering, and data compression. Numerous improvement methodologies have suggested but the least square is recognized because of the ability to parallelism, firm conjunction and merge of the unclassified inputs and ability to handle easily. And also they have observed that the current matrix factorization developments have done for a specific set of computers and it is insufficient. They explained the reason for this was because for a large-scale computer network has a bottleneck in the data communication where a single computer does not have to face any. Alternating least square on GPU is an encouraging trend. They have proposed a unique approach to expanding and improving the matrix factorization with including the approximate computing along with the memory utilization. The previous activities were related to increasing data reuse of the GPU memory. In modern methods, they tried to shrink avoidable operations without disturbing the convergence of the implementations and algorithms. All their developments are openly accessible for future researches [21].

In [16], Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov and Jack Dongarra have presented novel implementation design along with the improvement methodology for matrix inversion and for LU decomposition. They have pointed out that this kind of complications occurs in numerous scientific programs which belongs to the domain of astrophysics and mathematics. They have shown that different kind of mindset is required for the development of GPU kernel design for the tiny matrices. They have also taken the benefit of the tiny matrices to eradicate the in-between row swapping in the kernel inversions and in the decompositions. They were able to perform their work on a Pascal P100 GPU and 6x times, 14x times performance enhancement was gained respectively in the decomposition and matrix inversion against the cuBLAS implementation.

Gil Shabat, Yaniv Shmueli, Yariv Aizenbud and Amir Averbuch have shown that algorithms which are randomized have a high effect and contributes critically for the low-rank approximations of larger matrix sizes. In [22], randomized singular value decomposition is enhanced to a LU factorization implementation with random algorithms. Numerous fault limits are being presented by them which are correlated to sub-Gaussian matrices by relying on the results derived from this research. The limits of the error can be improved based on the known random algorithms and since the singular value decomposition algorithm is completely parallel and it can be performed effectively on GPU. They have presented the algebraic model and the comparison to other factorization approaches to clarify the efficiency of the proposed model [22]. In [23], Ryan G.McClarren has described and explained the LU factorization and the categories of matrices available in mathematical, scientific calculations and their mutual arrangements.

Evan Coleman and Masha Sosonkina have presented an analysis about the implementation of how to compute an incomplete LU decomposition. For this approach various techniques and methodologies used in order to enhance a much better parallel algorithm. This investigation has included numerous methodologies to validate and to understand the practicability of the suggested implementation. When it comes to the errors and other available tests, it is shown that changes in the point of algorithmic view can validate intersection of the incomplete decomposition along with the proposed suggestions which then will be lead to increase the efficiency of the resulting dynamics [24].

In [25], Johan Thunberg, Johan Markdahl and Jorge Gonçalves have addressed a synchronization in a distributed manner by rotating the columns in the matrices. The synchronization and the respective rotations are based on the control design. Dynamic control laws have been designed by them in order to address this type of synchronization complications. QR decomposition methodology along with a combination of auxiliary variables are the foundation for this control laws. The reasons and the advantages of using the QR decomposition methodology because of the capacity to separate the dynamics for a particular number of columns in matrices using this technique. They have shown that a closed loop system can be achieved the synchronization with this suggested implementation for quasi-strong collaboration graph topologies inside the control design.

2.2.4 GPU-only Implementation

Carlos Martins, Ricardo Chaves and others have developed a load balance solution that can efficiently distribute the workload of linear algebra operations between the CPU and the GPU(s) of a heterogeneous system. This has targeted the acceleration of the LU factorization due to its importance in the scientific and numeric fields but also in the LaPACK library. They have proposed two different solutions in their research. The multi-device solution that aims at distributing the workload efficiently on the CPU and the available GPUs. The GPU-only approach is focused on performing the factorization solely on the GPU without the need for constant data transfers during the execution. The limitation of this GPU only approach is the implementation of the less efficient factorization step. Because they have said that the factorization algorithm is hard to plot into the GPU architecture [26].

Michael Anderson, Grey Ballard and others have described the development of the Communication-Avoiding QR decomposition that can be executed completely on a GPU [27]. They have shown that the decrease in memory traffic produced by Communication-Avoiding QR factorization has allowed them to outperform the available parallel GPU implementations of QR decomposition for a large category of tall-skinny matrices. They have outperformed the Intel's Math Kernel Library up to 12x by the performance speed and 30x faster than Intel's Math Kernel Library on a multicore CPU [27].

NVIDIA also proving a library which is named as the cuSOLVER library for the factorization algorithms that to be performed entirely on the GPU. This vendor specific factorization library functions are available for Cholesky factorization, LU factorization and for QR factorization [28].

Azzam Haidar and others presented [2] their performance investigation, algorithm design concepts, and the improvements needed for the implementation of high-performance GPU-only algorithms for the dense Cholesky factorization. Since the hybrid algorithms are challenging to perform parallelize tasks on CPUs, Azzam Haidar and the team has developed a very efficient algorithm to be executed completely on GPU for the Cholesky factorization. GPU-only kernels eradicate the costly CPU-to-GPU data transmission and the tuning challenges related to slow CPU or low CPU-to-GPU bandwidth. They have provided sufficient evidence to prove that memory bound procedures can be planned, improved, and adjusted for GPU architecture in a way to be competitive with CPUs and reach their theoretical limits [2] with the Cholesky factorization.

They have raised the importance/need of the development in other highly needed routines, such as the QR and the LU decompositions as their future directions. Therefore, there is a need of the development of high-performance LU and QR factorization for GPU only implementation, where the performance of the application/algorithm is addressed in terms of algorithmic optimization path, kernel optimization path and implementation design path.

2.3 Summary of Literature Review

Studied resources and the literature review can be displayed in the following format as shown below in Table 2.1.

Citation in the Refernce	Research Type			Factorization Algorithm			Execute on		
	Comparison	Invent Something	Extension For Exsiting	Cholsky	LU	QR	CPU only	CPU and GPU	GPU only
[3] Numerical linear algebra algorithms and software			X	sparse matrix computations			X		
[4] Investigation of the performance of LU decomposition method using CUDA		X			X	X	X	X	
[8] A performance analysis of computing the LU and the QR matrix decompositions on the CPU and the GPU	X		X		X	X		X	
[11] Implementing QR factorization updating algorithms on GPUs			X			X		X	
[9] Providing GPU Capability to LU and QR within the ScaLAPACK Framework			X		X	X		X	
[12] Parallel LU Factorization on GPU Cluster			X		X			X	
[14] An Efficient GPU-Based Out-of-Core LU Solver of Parallel Higher-Order Method for Array Problems	X	X			X			X	
[15] A Guide for Achieving High Performance with Very Small Matrices on GPU		X	X	X	X			X	
[16] Factorization and Inversion of a Million Matrices using GPUs: Challenges and Countermeasures		X	X	X	X			X	
[17] Algorithm 980: Sparse QR Factorization on the GPU		X				X		X	
[18] LU factorization on heterogeneous systems	X		X		X			X	
[19] Transient Fault Resilient QR Factorization on GPUs		X				X		X	
[20] Stability and Performance of Various Singular Value QR Implementations on Multicore CPU with a GPU	X	X				X	X	X	
[21] Matrix Factorization on GPUs with Memory Optimization and Approximate Computing			X	X	X	X			X
[22] Randomized LU decomposition			X		X			X	
[24] Self-stabilizing fine-grained parallel incomplete LU factorization	X		X		X			X	
[10] Multi-GPU Implementation of LU Factorization			X		X			X	
[25] Dynamic controllers for column synchronization of rotation matrices: A QR-factorization approach			X			X		X	
[26] Parallelization of the LU Decomposition on Heterogeneous Systems	X				X			X	X
[27] Communication-Avoiding QR Decomposition for GPUs		X				X			X
[28] cuSOLVER		X		X	X	X			X
[2] High-performance Cholesky Factorization for GPU-only Execution		X		X					X

Table 2.1 Summary of the literature review

3 RESEARCH METHODOLOGY

3.1 Methodology

Experimental research methodology has been chosen to conduct this research. Because when implementing LU and QR factorizations on GPU, it will provide the understanding with the reasons by indicating what type of outcome occurs when an identified variable is manipulated in a controlled environment. Using this experiment, it is possible to answer "what-if" questions that related to the research questions, without a specific expectation about what this LU and QR factorization implementation reveals, or to confirm prior results [29]. The results can be used either to support or to disprove the hypothesis developed based on the research questions if this experiment is carefully implemented.

3.1.1 Experimental Research Methodology

The experimental research methodology is an organized and scientific approach to this research in which is allowed to manipulate one or more identified potential variables, and then to be measured any change in other variables. In simple terms, it is planned to conduct a true experiment along with a control group and one effect is only tested at a time.

3.1.1.1 Advantages of Experimental Research Methodology

While adjusting the independent variables related to the research question, Unwanted irrelevant variables are possible to be eliminated. In other research methodologies, control over irrelevant variables are usually higher. Experimental research methodology involves influencing the independent variable to observe the effect on the dependent variable. As a result of that cause and the effect relationship among these variables are possible to be determined. This methodology has strict conditions and control over the experiment. Because of this reason, the experiment can be performed repeatedly or a number of times and check the results. Reproduction is really significant because when comparable results are derived at different times, the confidence is very high with the results [30].

3.1.1.2 Disadvantages of Experimental Research Methodology

Simulated conditions that do not always represent the realistic, can be created with an experimental research due to the fact that all other variables are firmly organized. Because the circumstances are firmly organized and do not usually represent the reality, the output matrices of the input matrices may not be valid measurements of their behaviors in a non-experimental situation [30]. Some other disadvantages of the experimental research methodology are listed follows:

- Unnecessary variables are not continuously possible to remove
- Experiment situation or scenarios may not be related to the real world
- Human errors also play a significant role in the validity of the research.

3.1.2 Related Technologies to Solve the Research Question

Following technologies can be used to find a solution for the research problems and some characteristics of these technologies are listed below.

3.1.2.1 Linear Algebra PACKage

LAPACK offers the solutions for concurrent linear equations systems which is developed using Fortron 90 [31]. This library normally uses Basic Linear Algebra Subprograms (BLAS) to the fullest for the computation of the solution. Level 3 BLAS computer operations are available and designed in this LAPACK package. LAPACK uses multiple CPU processor cores when performing calculations and it is a CPU only approach. Matrix multiplication, triangular systems with several upper triangular solutions and block matrix operations are included in the LAPACK package due to the reason of the coarse granularity, in order to achieve higher proficiency and productivity in the level 3 BLAS operations. With the custom modified and upgraded implementations of the programs which are provided by the manufacturers to the high-performance computers, are likely to be rich in the terms of efficiency [31].

3.1.2.2 Open Multi-Processing - OpenMP

In OpenMP also it is possible to perform tasks in multiple processor cores and OpenMP will also be used to evaluate the research output with the OpenMP output results for the LU and QR factorization algorithms. OpenMP is an API (Application Programming Interface) developed in Fortran, C and C++ programming languages which support multiprocessing structures and shared memory architectures. Developers are able to program flexible, adaptable and modest parallel application programs with the help of a scalable and portable model of the OpenMP which includes the defined library methods and functions, environment variables and compiler directives. OpenMP uses multiple CPU processor cores when performing calculations and it is a CPU only approach. OpenMP has the ability to produce interfaces and applications, extending from the standard desktop computer to the supercomputer for parallel execution [32].

OpenMP programs have also been tested on distributed shared memory systems by researchers. By using MPI (Message Passing Interface) an OpenMP, hybrid application model is able to perform on a computer for parallel execution. In such cases, MPI has the responsibility of parallelism between nodes and OpenMP is taken care of the parallelism within a multi-core node [33], in order to outspread OpenMP for non-shared memory applications and to convert OpenMP applications into MPI application interfaces [34].

3.1.3 Selected Technology to Solve the Research Problem

Compute Unified Device Architecture has been chosen as the technology to implement LU factorization and QR factorization on a GPU. CUDA is a parallel computing platform and application programming interface model created by NVIDIA. It has allowed using a CUDA enabled GPU for general purpose processing. The CUDA programming platform is a software layer that gives direct access to the virtual instruction set and parallel computational elements of GPU for the implementation and execution of compute kernels [35].

OpenMP and LAPACK technologies are also to be used in this research to evaluate with the GPU only implementation of the QR factorization and GPU only LU factorization implementation.

High-performance Cholesky factorization has been implemented successfully in GPU only execution [2] by using NVIDIA CUDA cuSOLVER and cuBLAS library. And also communication avoiding QR algorithm is also implemented using CUDA programming platform [27]. Because of these reasons, CUDA has been chosen as the implementation technology to conduct this research.

3.1.3.1 Processing Flow On CUDA

CUDA flow of processing can be described as following and the graphical representation can be shown in Figure 3.1 below.

1. Copy data to GPU memory (from CPU memory to GPU memory)
2. GPU kernel initiation (initiated by CPU)

3. GPU code execution (CUDA code execute parallelly in the kernel)
4. Results copy to CPU memory (from GPU memory to CPU memory).

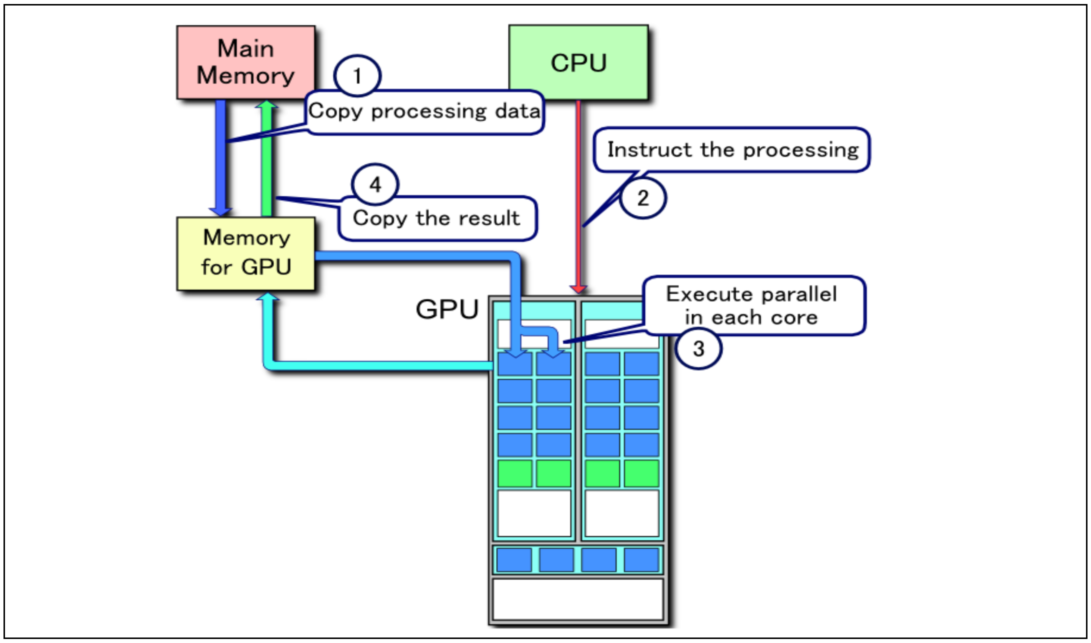


Figure 3.1 CUDA Processing Flow

3.1.3.2 Advantages of CUDA

CUDA has numerous benefits and advantages when compared with the typical general-purpose computation on GPUs when it is come to the graphics API usage. The main advantage is code can be read from random memory addresses in the memory, this is known as scatter reads. A shared memory region of CUDA can be shared between threads. It can be applied as a user-managed cache which provides the potential to a higher bandwidth while using texture lookups in this shared memory concept. CUDA also allows data downloads at a rapid speed and faster read/write operations from and to the GPU. Full provision for integer and bitwise computations and tasks, together with integer texture lookups can also be listed as advantages of CUDA [35].

3.1.3.3 Limitations of CUDA

Whether for the host computer or the GPU device, all CUDA source code is now processed according to C++ syntax rules. As with the more general case of compiling C code with a C++ compiler, therefore, it is possible that old C-style CUDA source code will either fail to compile or will not behave as originally intended [35].

Interoperability with rendering languages such as OpenGL is one-way, with OpenGL having access to registered CUDA memory but CUDA not having access to OpenGL memory. Unlike OpenCL, CUDA enabled GPUs are only available from NVIDIA [35].

3.2 Research Design

To avoid the performance drawbacks of the LU factorization and QR factorization, expensive CPU-GPU communications should be removed and the solution is to implement LU and QR factorization on completely on GPU execution [2], [8], [9]. And also to find such way to perform both panel factorization and trailing matrix update in the GPU, using different or same GPU streams without getting affected to the performance drawback [2], [10]. These are the research questions going to be addressed by this research.

Inputs for this research will be some random matrices generated by an equation. Then in the process that input matrices will perform LU factorization and QR factorization using the GPU and will perform necessary tasks. The output of these implemented algorithms will be matrices in the form of LU factorization and QR factorization. But the output of this research will be a LU factorization and QR factorization algorithms which can be executed on an entirely GPU environment along with high-performance capabilities. When it comes to the features of this solution algorithms of this research, the main feature is high-performance and the next main feature is not using the CPU to perform the factorization processes. Research design can be converted to a graphical representation of components as shown in Figure 3.2 below.

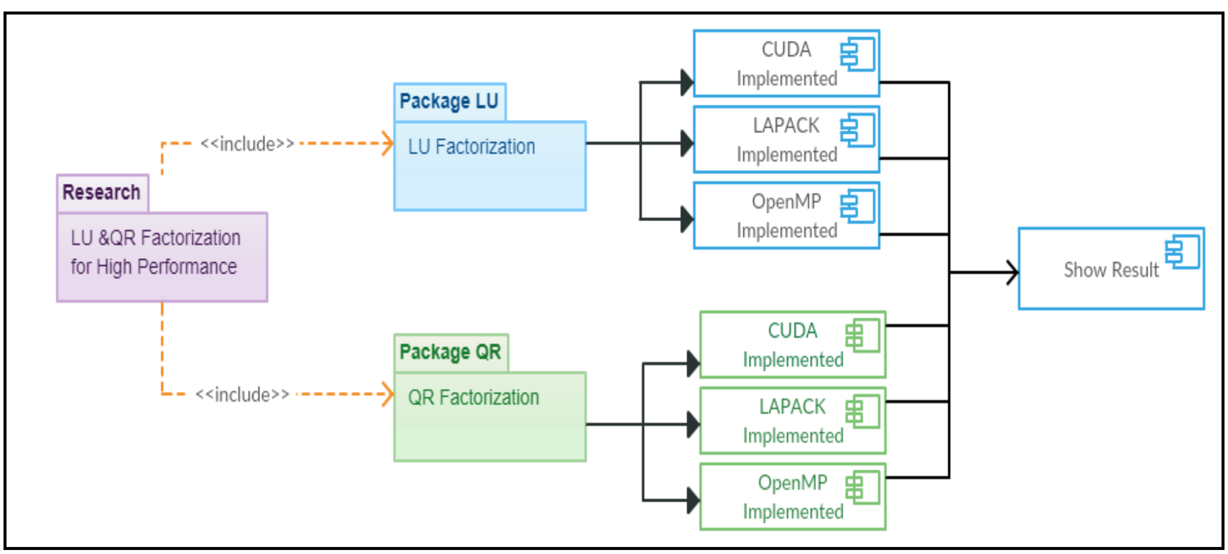


Figure 3.2 Graphical Representation of the Research

3.2.1 GPU only LU Factorization Design.

In this research, LU factorization is to be implemented in GPU only execution using **block LU factorization** concept. There are several reasons to select the block LU factorization concept to this research such as,

- Block LU factorization work with blocks of data having b^2 elements, performing $O(b^3)$ operations. The O(b) ratio of work to storage which means that the processing elements with an O(b) ratio of computing speed to both input and output bandwidth can be tolerated. Because of this reason, we can expect faster results with the block LU factorization algorithm [36], [37].
- Block LU factorization algorithms are usually powerful and efficient in matrix multiplication. And LU factorization consists of considerable matrix multiplications in the algorithm. Due to these facts, this is a benefit for the reason that almost every up-to-date parallel machine is decent at matrix multiplication especially GPUs [36], [37].
- Block algorithms are able to deal with matrices by considering arrays of tiny matrices. Because of these reasons, block LU factorization has identified as quite beneficial for this research implementation [37].

The structure of the block LU factorization can be graphically described as shown in Figure 3.3 below.

$$\begin{array}{|c|c|}\hline A_{00} & A_{01} \\ \hline A_{10} & A_{11} \\ \hline \end{array} = \begin{array}{|c|c|}\hline L_{00} & 0 \\ \hline L_{10} & L_{11} \\ \hline \end{array} * \begin{array}{|c|c|}\hline U_{00} & U_{01} \\ \hline 0 & U_{11} \\ \hline \end{array}$$

Figure 3.3 Structure of the Block LU factorization

In the GPU only LU factorization algorithm implementation we have planned and designed the algorithm using both cuSOLVER and cuBLAS routines together where they are necessary to be implemented. The pseudo code of the algorithm is shown below in Figure 3.4.

```
Implementing Block LU Factorization Algorithm Pseudo Code

FOR A <-- 1 to noOfBlocks
        DO i<-- 1 to columnsInBlock
                DO i<-- 1 to rowsInBlock
        initiateMatrixValues();
                END DO
        END DO
END FOR

DO
        L00.U00 <-- DGETRF(A00)
        DO i<-- 1 to columnsInBlock
                DO i<-- 1 to rowsInBlock
                        representTwoMatricesSeperately()
                END DO
        END DO

        U01 <--DTRSM(L00,A01)
        L10 <--DTRSM(U01,A10)
        L11.U11 <-- DGEMM(L10,U01,A11)
        L11.U11 <-- DGETRF(A11)
        DO i<-- 1 to columnsInBlock
                DO i<-- 1 to rowsInBlock
                        representTwoMatricesSeperately()
                END DO
        END DO
END DO
```

Figure 3.4 Implementing Block LU Factorization Algorithm Pseudo Code

3.2.2 GPU only QR Factorization Design.

In this research, QR factorization is to be implemented in GPU only execution using **Block Householder QR factorization** concept. There are several reasons to select the block householder QR factorization concept to this research such as,

- Householder reflectors using QR algorithms are known to be numerically stable than the QR algorithms using Cholesky QR and the Gram-Schmidt process. In the householder approach, the householder vectors are broken up in such a way that communication is minimized [38], [39].

- The trailing matrix updates for several Householder vectors can be delayed and done all at once using matrix-multiply for one block. This allows for higher arithmetic intensity on machines with a memory hierarchy. Because of this reason, it leads to better performance. For the very same reason, this is called blocked Householder QR factorization because it allows the updates to the trailing matrix to be blocked in cache [38], [39].

In the GPU only QR factorization algorithm implementation we have planned and designed the implementation not to use cuBLAS routine functions in order to save the amount of time to initiate the cuBLAS handles in the CPU memory. The pseudo code of the algorithm is shown below in Figure 3.5 and Figure 3.6 is described as the important two functions mentioned in Figure 3.5 in the pseudo-code definition.

```
FOR A ← 0 to rowIndex          //Matrix Value initiation
    DO I ← 0 to columnsIndex
        initiateMatrixValues();
    END DO
END FOR

FOR col ← 0 to (columnSets -1 )          //Generate R matrix
    FOR row ← 0 to (panelsForRows -:
        ProcessPanelFactorization()   //Function defined in pseudoCode
        IF (col + ColumnsPerBlock) < squareMatrixSize THEN
            ProcessTrailingMatrixUpdate()      //Function defined in pseudoCode
    END FOR
END FOR

FOR col ← 0 to (columnSets -1 )          //Generate Q matrix
    defineStartEndRows()
    FOR i ← 0 to squareMatrixSize
        generateV()
    END FOR
    FOR i ← 0 to squareMatrixSize
        createH-Matrix()
        prevQ[i] = Q[i]
    END FOR
    Q[] ← GEMM (prevQ[] *H[])
END FOR
```

Figure 3.5 Block householder QR Factorization Pseudo Code

```
FUNCTION  ProcessPanelFactorization()
      initializePanel();
      FOR c ← 0 to columnsInThePanel
            defineStartEndRows()
            FOR i ← vStart  to vEnd
                  getThePanelFirstColumnInnerProductSummation()
            END FOR
            FOR i ← vStart  to vEnd
                  updatePanelColumnValues()
            END FOR
            FOR i ← 0  to RowsPerBlock
                  computeZandStoreToW()
            END FOR
            FOR colNext ← (col + 1)  to ColumnsPerBlock
                  FOR pRow ← vStart  to vEnd
                        applyReflectorRemainingColumnsPanel()
                  END FOR
            END FOR
      END FOR
END FUNCTION
FUNCTION  ProcessTrailingMatrixUpdate()
            blockCol ←  pc + ColumnsPerBlock +  RowsPerBlock
            FOR i ← 0  to (RowsPerBlock * ColumnsPerBlock)
                  defineStartEndRows()
                  mRow ←   (i % RowsPerBlock)
                  MCol   ←   (i/ RowsPerBlock)
                  IF (pr +mRow) < m && (pc + mCol) < m THEN
                        IF (mRow > vStart && mRow < vEnd) THEN
                              y ← matrix[(pr +mRow) * m (pc +mCol)]
                        ELSE IF (mRow == vEnd) THEN
                              y ← 1
                  Y[mRow + mCol *RowsPerBlock]  ←  y
            END FOR
            FOR currentColumn ← 0 to columnsToBeUpdated
                  FOR i ← 0  to RowsPerBlock
                        IF i< RowsPerBlock THEN
                              FOR j ← 0  to RowsPerBlock
                                    FOR k ← 0  to ColumnsPerBlock
                                                ywT ← Y[i + k * RowsPerBlock] * W[j + k * RowsPerBlock]
                                    END FOR
                                    val ←  val + ( ywT * updatedRowValue)
                              END FOR
                              Matrix[pr + i+ (blockCol + currentColumn)] ← val
                  END FOR
            END FOR
END FUNCTION
```

Figure 3.6 Panel Factorization and Trailing Matrix Update Pseudo Code

4 IMPLEMENTATION

4.1 LU Factorization Implementation on GPU

In order to perform the LU factorization, there should be an input matrix. Then the factorization algorithm will get that particular matrix as an input and then perform the steps of the factorization.

4.1.1 Initiate Input Matrix

The typical way to do this is to initiate the matrix in the CPU and then allocate the memory in the GPU and then copy the matrix into the GPU memory as shown in Figure 4.1. But in our implementation, we have initiated the matrix in the GPU memory as shown in Figure 4.2.

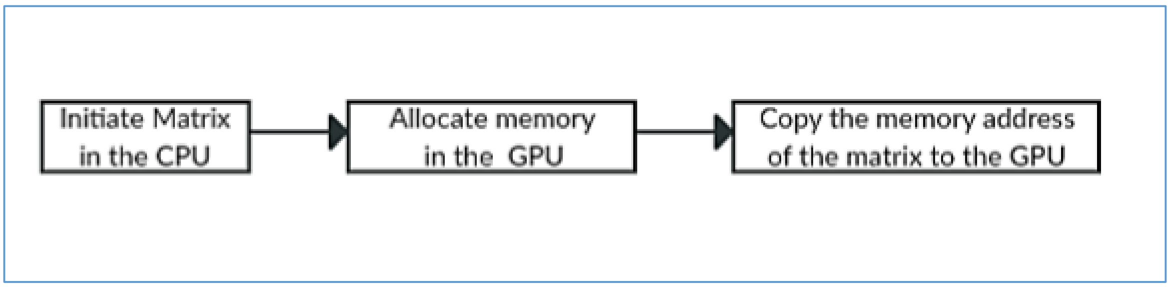


Figure 4.1 Matrix initiation in normal way

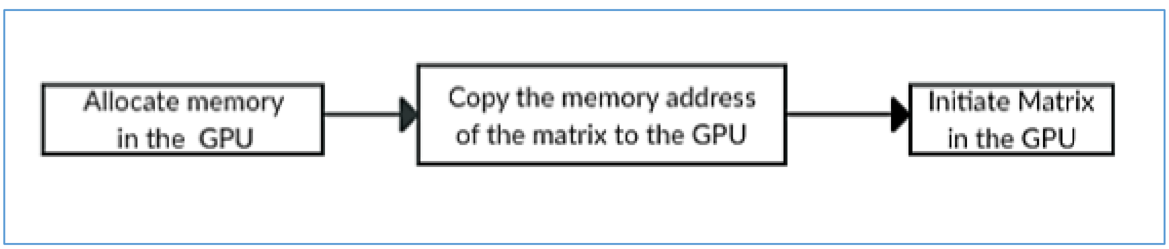


Figure 4.2 Matrix Initiation in the implemented way

4.1.2 GPU-only Implementation using cuSOLVER library

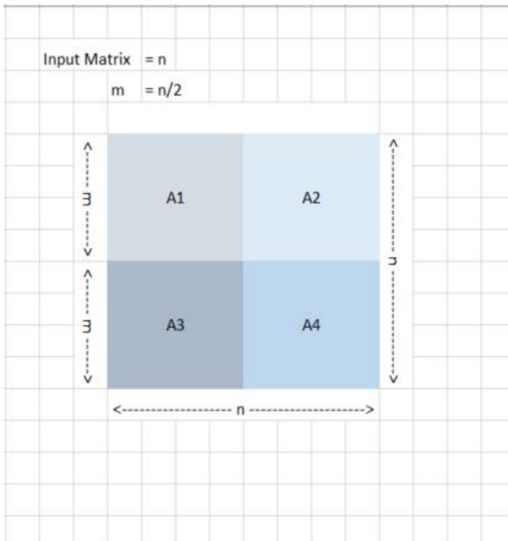
We have implemented LU factorization using the cuSOLVER library and then we have optimized it to generate results faster with better runtimes. But the ability to perform improvements to this implementation is really difficult because of the functionality inside the cuSOLVER functions are hidden even in the development level. Because of this reason we have planned to implement the LU factorization on a GPU-only execution in a different way.

4.1.3 Suggested Way to Implement the GPU-only LU factorization

The concept of this blocked LU factorization is used in this implementation[40], [41]. The input matrix which is to be factorized using the LU decomposition is divided into [m*m] sized 4 matrices using Equation 7 as shown below.

$$\mathrm{m} = [(\text{input matrix size}) / 2] \qquad \text{Equation 7}$$

This separation of the input matrix into 4 several matrices is described below using Figure 4.3.



Matrix size	elements in the matrix	size of a small block of matrix	elements of the small matrix	Total no of elements of 4 block matrics	Check for any differences
n	n*n	m	m*m	4*(m*m)	[n*n] - [4*(m*m)]
32	1,024	16	256	1,024	-
64	4,096	32	1,024	4,096	-
128	16,384	64	4,096	16,384	-
256	65,536	128	16,384	65,536	-
512	262,144	256	65,536	262,144	-
1,024	1,048,576	512	262,144	1,048,576	-
2,048	4,194,304	1,024	1,048,576	4,194,304	-
3,072	9,437,184	1,536	2,359,296	9,437,184	-
4,096	16,777,216	2,048	4,194,304	16,777,216	-
5,120	26,214,400	2,560	6,553,600	26,214,400	-
6,144	37,748,736	3,072	9,437,184	37,748,736	-
7,168	51,380,224	3,584	12,845,056	51,380,224	-
8,192	67,108,864	4,096	16,777,216	67,108,864	-

Figure 4.3 Separation of input matrix into 4 small matrices

With this matrix, we have tried to perform blocked LU factorization by using these identified [m*m] 4 matrices. This representation is below shown in Figure 4.4 since the GPU-only LU factorization is based on equations in this representation.

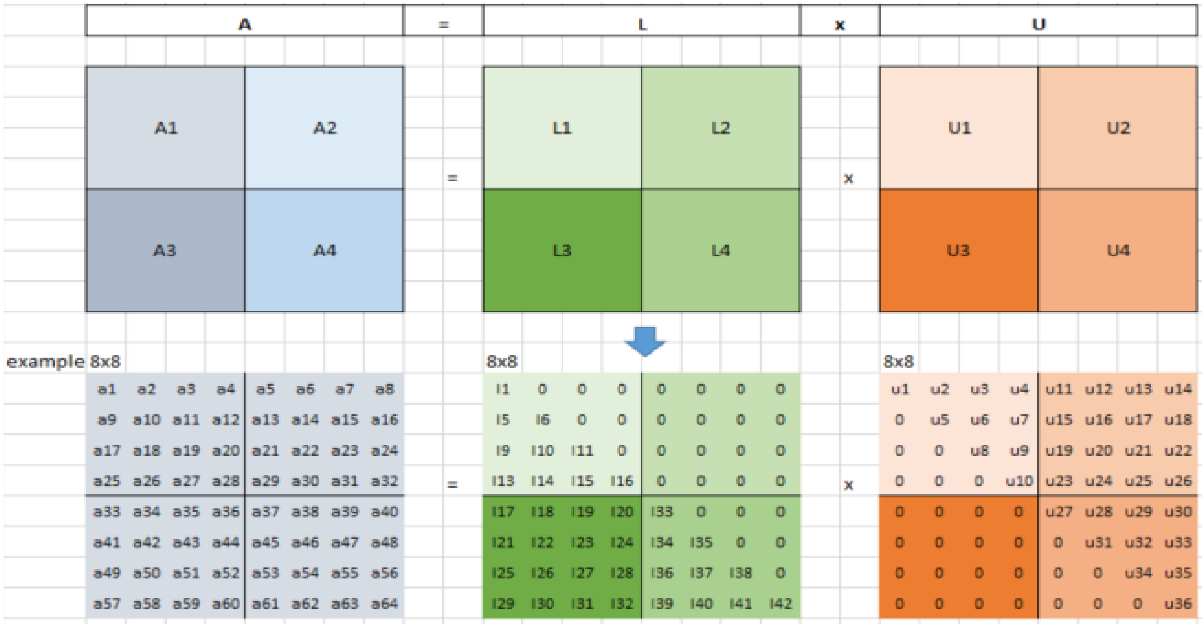


Figure 4.4 Representation of blocked LU factorization

In this way, LU factorization has 4 steps to be performed. L2 sub-matrix and U3 sub-matrix consist of nothing but zero in every element of those matrices. L1, L3, L4, U1, U2, U4 are to be found using different techniques/ways.

<u>Compute and derive L1 and U1</u>

In order to derive L1 and U1 matrices, we have implemented GETRF routine to be executed in the Nvidia MX130 GPU as a GPU-only code function using the cuSOLVER library for the block of A1 matrix. This step has included solving A1 matrix to LU decomposition as the panel/block factorization. In Figure 4.5, derivation of L1 and U1 and matrices are shown below. The result of this step has become inputs for other steps, so in order to perform other steps, L1 and U1 matrices computation has got the highest priority. A1 can be shown in Equation 8 below.

$$A1 = (L1.U1) + (L2.U3) \qquad \text{Equation 8}$$

Since L2 and U3 matrices are zero, A1 matrix can be re-represented as shown in Equation 9.

$$A1 = (L1.U1) \qquad \text{Equation 9}$$

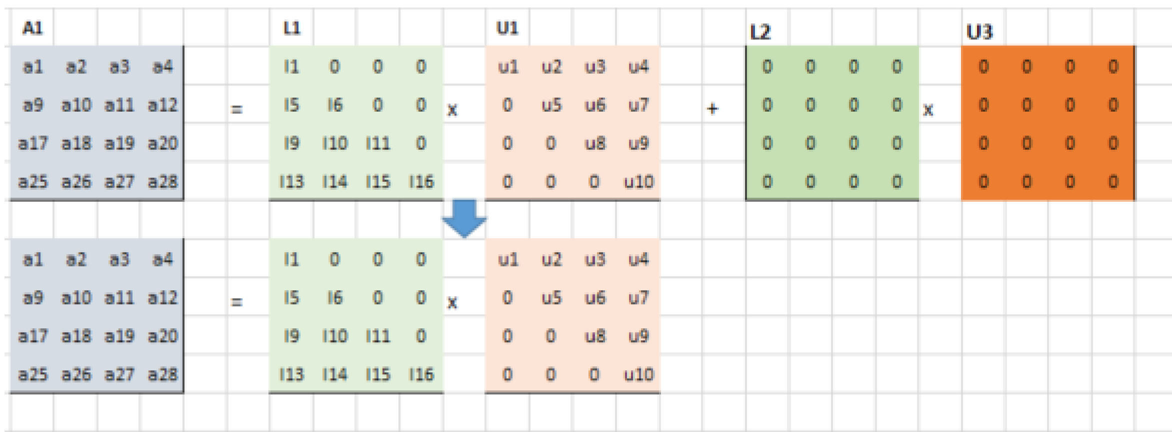


Figure 4.5 Panel/Block Factorization of A1 Matrix

<u>Compute and derive U2 and L3</u>

Once the L1 and U1 matrices have been computed, the next step is available to be implemented. In here A2 can be represented as following matrix equation as shown in Equation 10.

$$A2 = (L1.U2) + (L2.U4) \qquad \text{Equation 10}$$

Since the L2 is a zero matrix. L2 and U4 matrix multiplication is also a zero matrix. So the A2 matrix can be represented as the following equation. Graphical representation of A2 matrix can be shown in Figure 4.6 below.

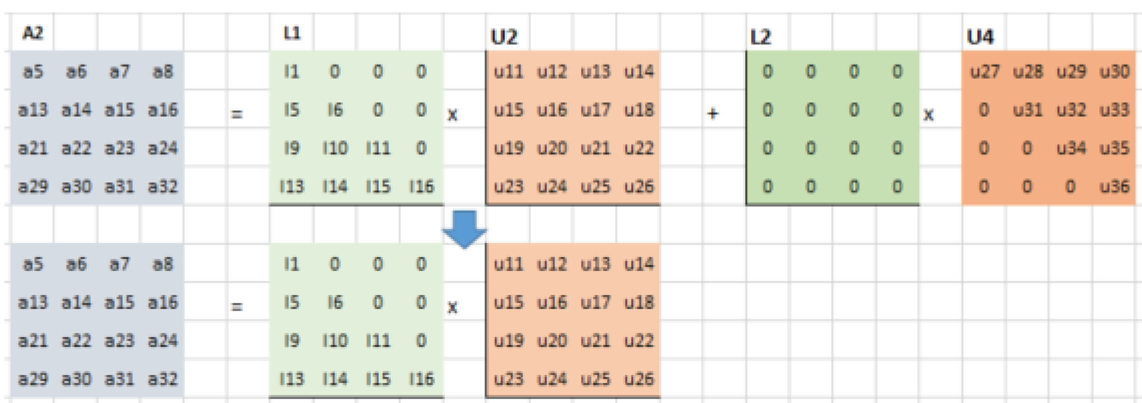


Figure 4.6 Representation of A2 sub-matrix

By following the above steps again for the sub-matrix A3, matrix equation and representation A3 matrix can be shown in Figure 4.7 and the A3 matrix can be derived from Equation 11 shown below.

$$A3 = (L3.U1) + (L4.U3) \qquad \text{Equation 11}$$

U3 is a zero matrix, so A3 can be rewritten in the following way as shown in Equation 12.

$$A3 = (L3.U1) \qquad \text{Equation 12}$$

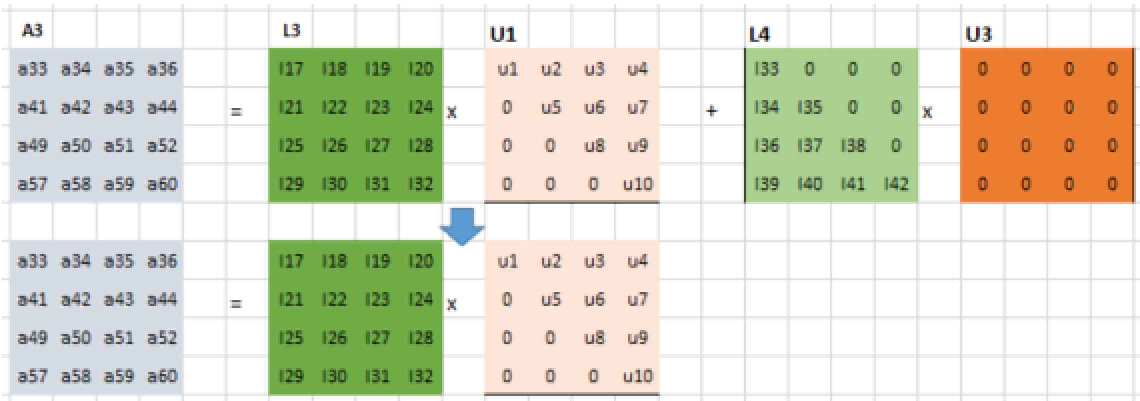


Figure 4.7 Representation of A3 sub matrix

By using above two equations U2 and L3 matrices can be obtained. To obtain U2 matrix the dependent matrices are A2 and L1 matrices and to obtain L3 matrix the dependent matrices are A3 and U1 matrices. So the above two matrix equations/operations are independent of each other and can be executed through parallelly.

When it comes to the implementation of these two matrix operations in the GPU as GPU-only executable codes, two possible routines are available for this operation,

- GEMM – General Matrix-Matrix
- TRSM – Triangular Solving Matrix

First, we have tried to implement an operation using GEMM routine. To perform this operation output matrix should be existing as a single matrix. But in here, Equation 13 and the output matrix is U2. So in order to compute U2 first, we have to get the $L1^{-1}$ and then perform the matrix multiplication using GEMM routine. Since this routine looks heavy on the computation we have measured the elapsed times for the GEMM and TRSM routines for some matrices and the results are shown below in Figure 4.8 as a graphical representation. In this Figure 4.8 matrix size is represented in X-axis and runtime is represented in milliseconds in Y-axis.

$$A2 = L1.U2 \qquad \text{Equation 13}$$

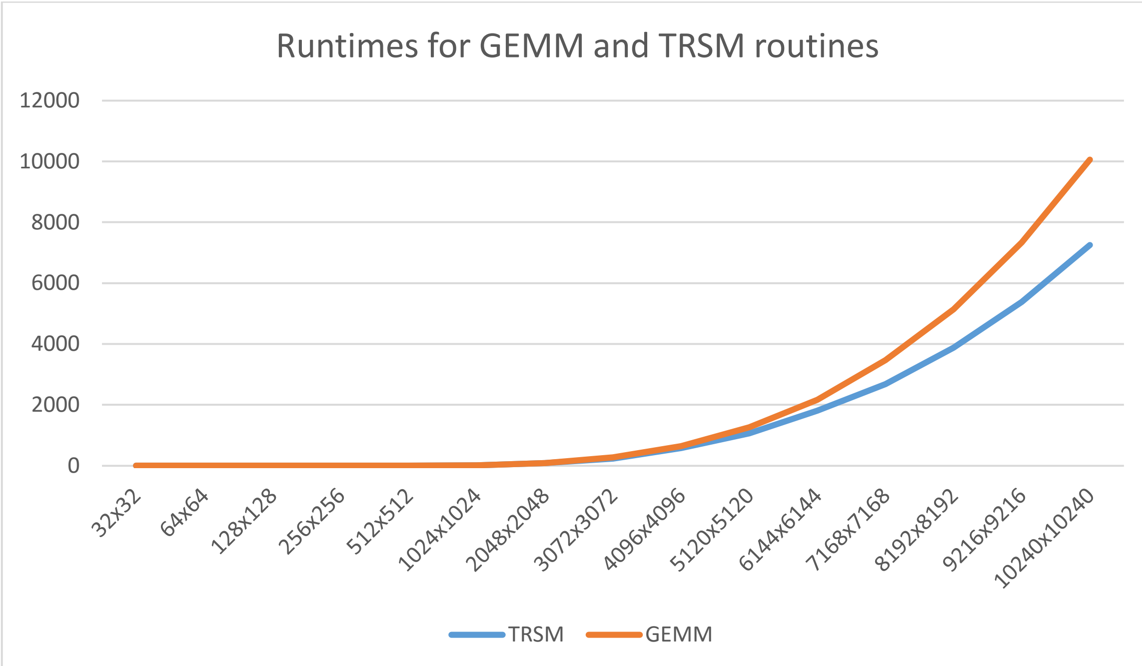


Figure 4.8 Runtimes for the GEMM and TRSM routines for different matrix sizes

When the matrix size is getting large TRSM routing has the highest efficiency when the runtime is compared. So for deriving U2 and L3 matrices the best option is to implement the TRSM routine for these two matrix operations. These two matrix operations were implemented parallelly using cuBLAS DTRSM function with two different CUDA streams since the two tasks are independent of each other.

<u>Compute and derive L4 and U4</u>

This is the most time-consuming step in the whole process. In order to compute the A4 matrix using L4 and U4 matrix, the equation can be shown below in Equation 14 and Equation 15 respectively.

$$A4 = (L3.U2) + (L4.U4) \quad \text{Equation 14}$$

$$(L4.U4) = A4 - (L3.U2) \quad \text{Equation 15}$$

To compute the L4 and U4, we have to perform the L3 and U2 matrix multiplication and then subtract from A4 matrix. To perform the matrix multiplication GEMM routine has been used and implemented using cuBLAS DGEMM function for GPU-only execution. This representation is graphically shown in Figure 4.9 below.

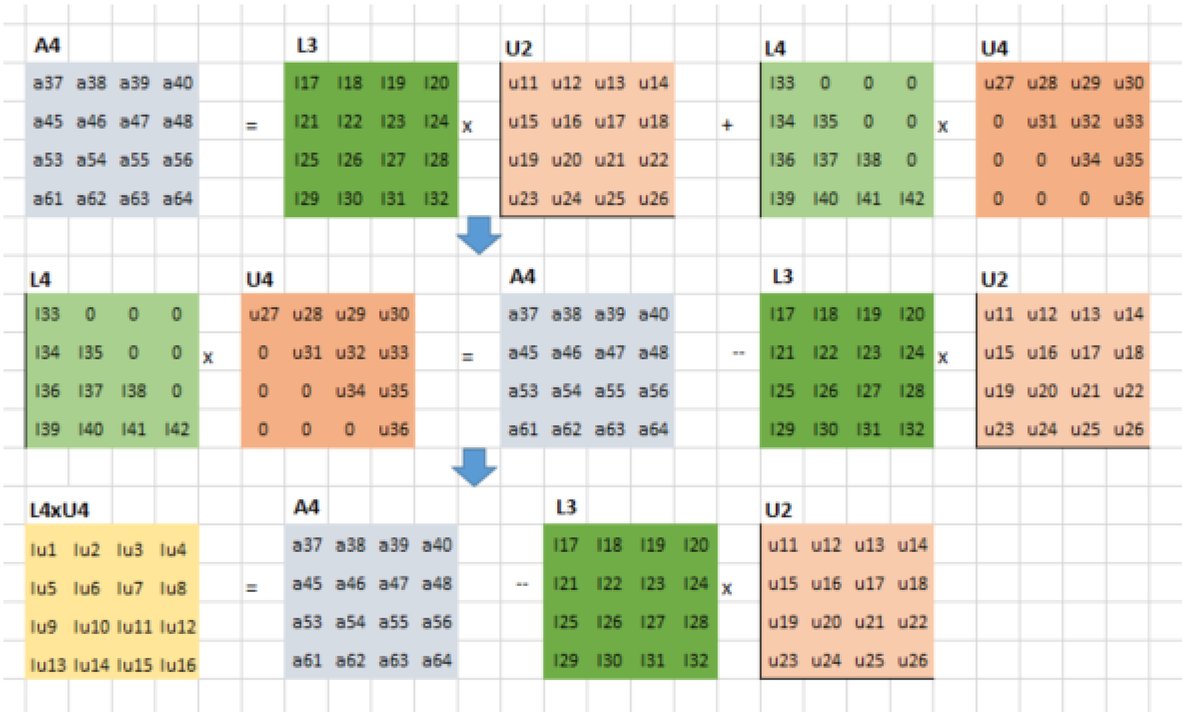


Figure 4.9 Deriving L4.U4 matrix

But in here L4.U4 matrix is represented using one matrix. In order to represent as two matrices again DGETRF routine has been used for the GPU-only implementation via the cuSOLVER library. L4 and U4 matrices decomposition is represented in Figure 4.10 below.

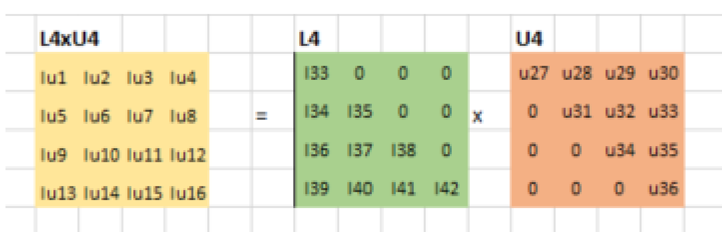


Figure 4.10 Decomposing L4U4 matrix into two matrices

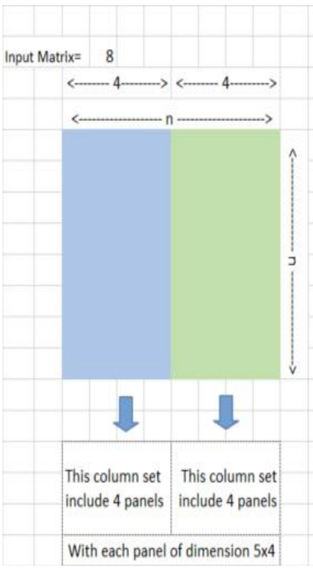
In this, we have redesigned the LU factorization and has implemented the LU factorization algorithm for a GPU-only execution.

4.2 QR Factorization Implementation on GPU

The same approach has been taken for the GPU only implementation of the QR factorization. By using this approach for the QR factorization will increase the calculating complexity of the algorithm. Suggested approach/algorithm will perform the factorization using block panels.

4.2.1 Determine the square matrix size

Based on the user input for the matrix size, the square matrix is derived using an equation and calculate block panel sizes accordingly. Determining square matrix size is shown in Figure 4.11 below.



Entered Matrix size	no of panels (p)	Can performable matrix Size	elements in the matrix	Rows per Block	Columns per Block	No of Panels for Total Rows	No of Panels for Total Columns
n	(n- x)/(x-y)	x+ p(x-y)	n*n	x	y	[(n-x)/(x – y)] + (n-x)%(x - y)	(n/y) + n % y
8	3	8	64	5	4	4	2
16	1	16	256	16	4	1	4
32	1	32	1,024	32	4	1	8
64	1	64	4,096	64	4	1	16
128	4	128	16,384	32	8	5	16
256	8	256	65,536	32	4	9	64
512	8	512	262,144	64	8	9	64
1,024	1	1,020	1,048,576	512	4	6	256
2,048	1	2,044	4,194,304	1,024	4	6	512
3,072	2	3,064	9,437,184	1,024	4	11	768
4,096	1	4,092	16,777,216	2,048	4	6	1,024
5,120	4	5,104	26,214,400	1,024	4	21	1,280
6,144	2	6,136	37,748,736	2,048	4	11	1,536

Figure 4.11 Determine the matrix size

In Figure 4.11 the variable x and y are the numbers of rows and number of columns of the block panel. Based on the user input matrix size n, number of panels and the square matrix that can be performed without an error is computed using Equation 16 as below shown, where p is represented by the no of panels.

$$SquareMatrixSize = x + p(x - y) \qquad \text{Equation 16}$$

4.2.2 Suggested Way to Implement the GPU-only QR factorization

The concept of the blocked Householder QR factorization is used in this implementation [27], [42]. After determining the square matrix, block panel size and the number of row-column panel distribution, the panel factorization for a block panel can be performed. The first panel can be represented in Figure 4.12 as shown below.

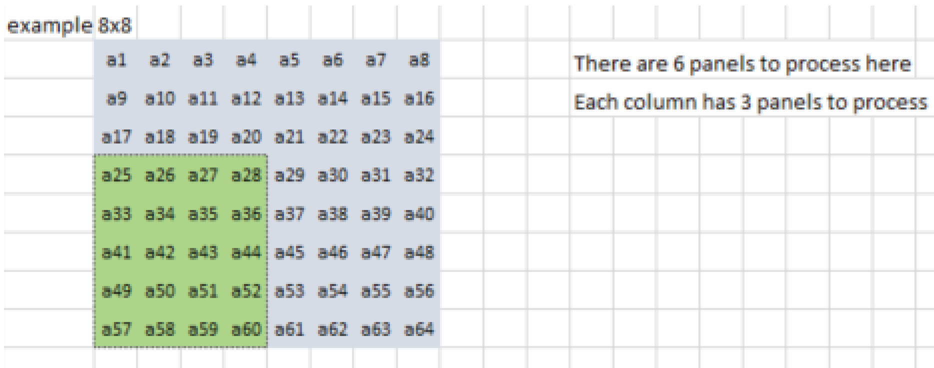


Figure 4.12 first Panel to be panel factorized

In the first panel of the first column set, take the first column and get the sum of the inner product of those elements, according to Figure 4.12 inner product equals to $[(\mathrm{a25})^2 + (\mathrm{a33})^2 + (\mathrm{a41})^2 + (\mathrm{a49})^2 + (\mathrm{a57})^2]$. Then the square root of that sum value will be calculated using Equation 17.

$$\sqrt{innerProductSum} = \sqrt{(\mathrm{a25})^2 + (\mathrm{a33})^2 + (\mathrm{a41})^2 + (\mathrm{a49})^2 + (\mathrm{a57})^2} \qquad \text{Equation 17}$$

The sign will be based on the leading element of the block panel, if the leading element is less than zero the sign will be negative otherwise the sign will be positive (negative = -1, positive = +1).

As the next step, u value is computed as shown in Equation 18, where u will be used to update the elements in the selected columns. Tau value of this column is also calculated here using Equation 19 for the later use.

$$u = leadingElement + (sign * \sqrt{innerProductSum}\ ) \qquad \text{Equation 18}$$

$$\mathrm{Tau} = \mathrm{sign} * \left(\frac{\mathrm{u}}{\sqrt{innerProductSum}}\right) \qquad \text{Equation 19}$$

Then the first element value and the respective column values can be calculated using Equation 20 and Equation 21 respectively.

$$\mathrm{firstElementValue} = -\mathrm{sign} * \sqrt{innerProductSum} \qquad \text{Equation 20}$$

$$otherElementValue = otherElementValue/u \quad \text{Equation 21}$$

This part can be graphically represented in the below shown Figure 4.13 and it iterates through the number of elements in the block panel column.

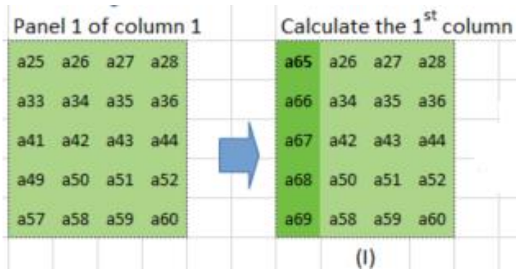


Figure 4.13 Compute the first column of the block panel

Get the first column updated element values and generate the Z values using W, Y^T and V in the form of Equation 22 shown below.

$$Z = W * Y^T * V \quad \text{Equation 22}$$

After that, applying the reflector values to the other column elements are based on the previous column calculated values. This step can be shown in the below Figure 4.14 as a graphical representation.

Figure 4.14 Applying the reflector value in the other columns of the block panel

This process should be applied recursively to the rest of the columns of the block panel by considering the sub-panels in the block panel. The implemented loop can be depicted in a graphical way, as shown in Figure 4.15 below.

Figure 4.15 Panel factorization of panel 1 of column 1

In Figure 4.15 the bold color matrix elements are the leading elements of that sub panel, which was used to calculate the square root of the inner product summation of the sub panel's first column in this GPU only QR factorization implementation.

After the panel factorization of the block panel 1 is in the shape of Figure 4.16 in the implemented QR algorithm.

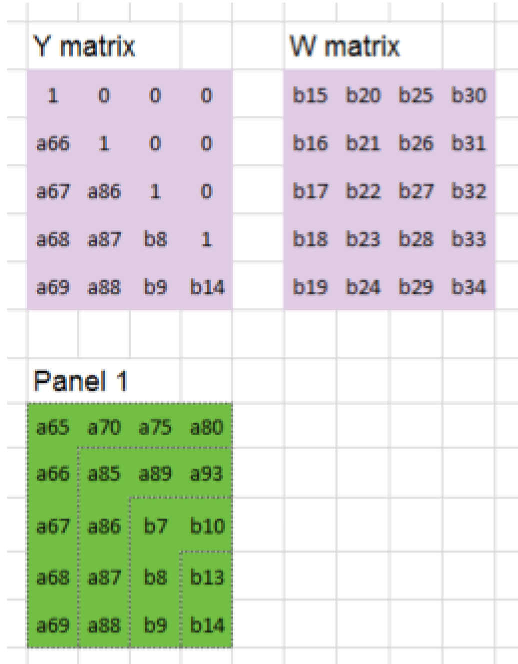


Figure 4.16 Matrices after the panel factorization in the QR GPU only algorithm

Now the algorithm is completed the panel factorization computation of the block panel 1 and the trailing matrix update of the respective matrices of the same rows which are not affected to the computation so far is targeted in this implementation. Trailing matrix update is processed using one column at a time. The algorithm has used Equation 23 to update the current column values of the elements, where 'I' is an Identity matrix of the same square matrix size. This iterative process also shown in the below Figure 4.17 to illustrate how the looping occurred in the GPU only implemented block QR algorithm.

$$A = (I + YW^T)A \qquad \text{Equation 23}$$

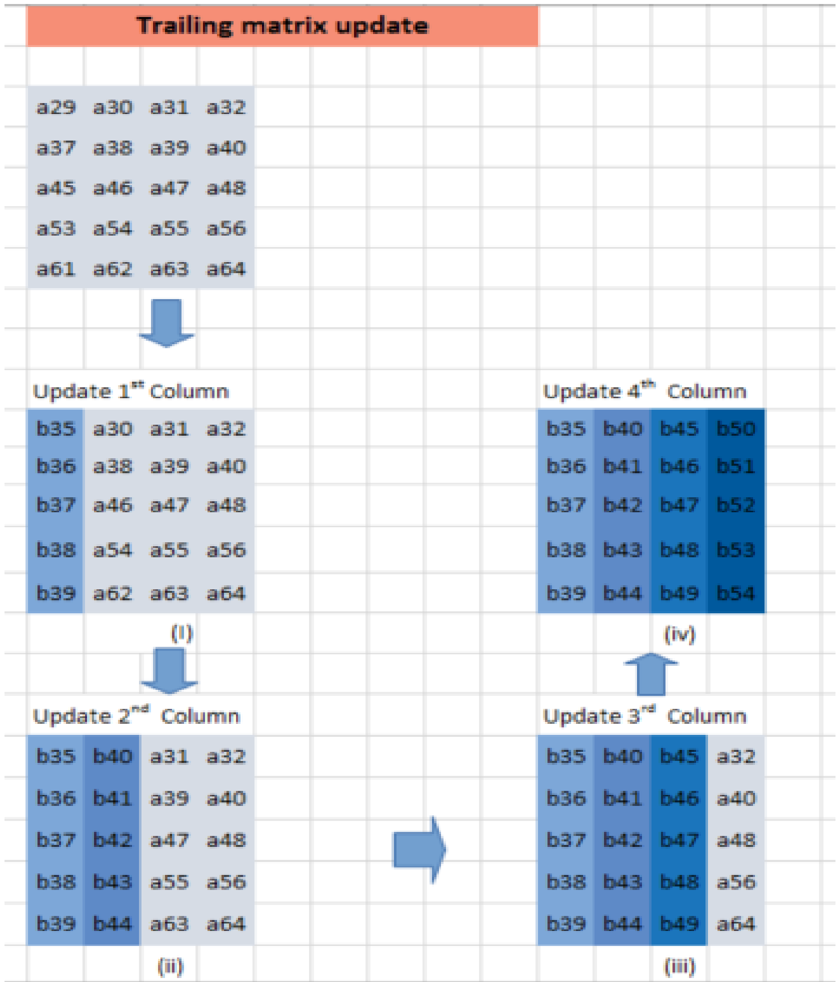


Figure 4.17 Trailing matrix update on the panel 1 of column 1

This is a very expensive calculation to be performed, even with the above-depicted process of computing both panel factorization and trailing matrix update on the panel 1 of column 1 in the block panel. There are considerable panels to be performed in the next steps. So this process is recursively computed each column set wise and row panels in a column set wise in order to compute both panel factorization and trailing matrix in the necessary places of the matrix. Pseudo-code to the implemented algorithm can be shown in Figure 4.18 below to describe the implemented algorithm in an abstract way.

```
FOR each column in the columnSets
        FOR each rowPanel in the panelsForRows
                ProcessPanelFactorizationParallely(SelectedPanel)
                ProcessTrailingMatrixUpdateParallely(SelectedPanel)
        END FOR
END FOR
```

Figure 4.18 Pseudo-code for the implemented block QR factorization

At this stage, algorithm is completed computing the R matrix, but the R matrix is not in the upper triangular form. The elements below the diagonal are removed and derived the upper triangular matrix R. This is shown graphically in Figure 4.19 below.

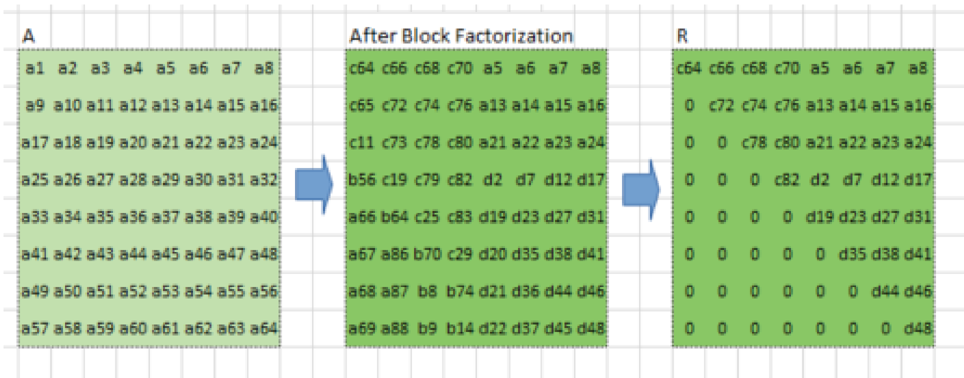


Figure 4.19 Remove the elements below the diagonal in R matrix

Now the last part of the implementation is to derive the Q matrix from the matrix R. Pseudo code of the implemented code block in this GPU only QR factorization is below shown in Figure 4.20.

```
FOR each column in the columnSets
        FOR each rowPanel in the panelsForRows
                ProcessDeriveQfromR(SelectedPanel)
        END FOR
END FOR
```

Figure 4.20 Deriving Q from R matrix

In this code segment, the algorithm is recurring through the panels and then compute the Q at the end of this loop. The derivation of Q is implemented as recursive steps but for the understanding purposes one recursion is shown in Figure 4.21, Figure 4.22, Figure 4.23, Figure 4.24 and Figure 4.25 of the panel 1 first recursion in the process.

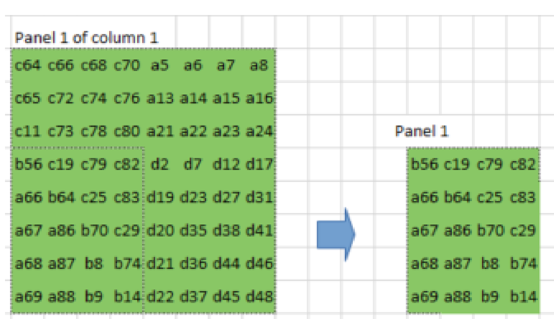


Figure 4.21 Identifying the panel 1 From the R (copy of R matrix)

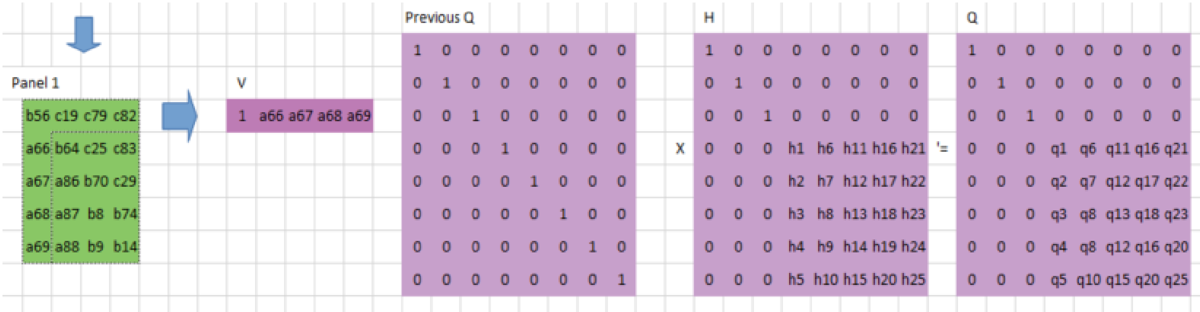


Figure 4.22 Calculate Q based on the 1st column of the panel 1

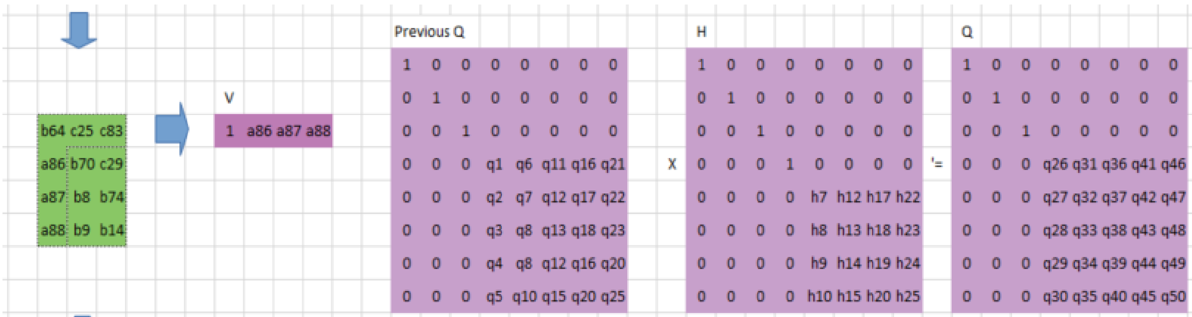


Figure 4.23 Calculate Q based on the 2nd column of the panel 1

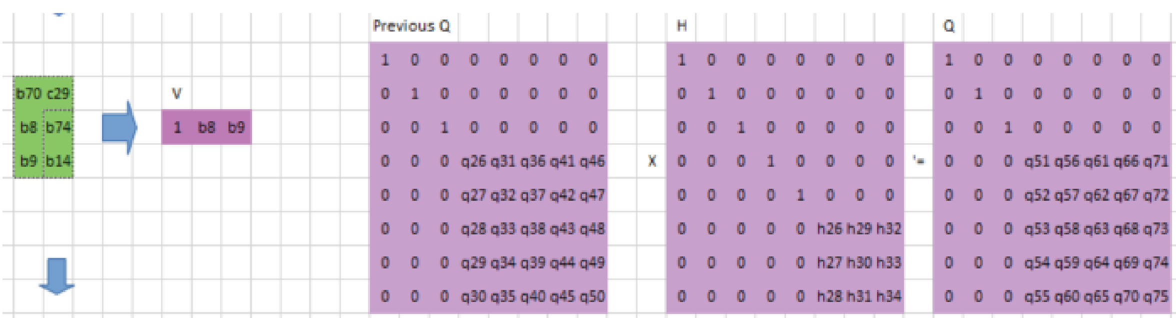


Figure 4.24 Calculate Q based on the 3rd column of the panel 1

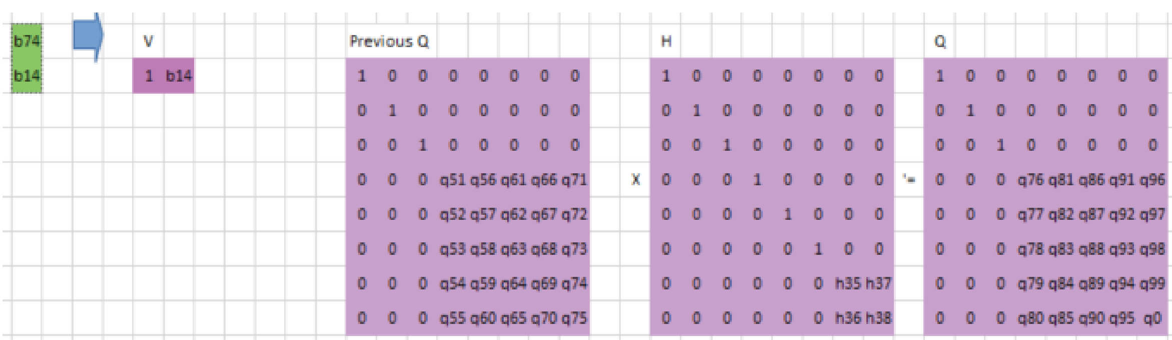


Figure 4.25 Calculate Q based on the 4th column of the panel 1

In this implementation code segment, initially Q matrix is an identity matrix and with V values of the columns in the iterations H matrix is computed. Then the Q matrix is copied to the previous Q matrix and then performed the previousQ x H matrix multiplication. To perform the matrix multiplication no library is used with GEMM routine. Because to save the time to create the handle and streams in the cuBLAS library function calls. After iterating through all the row panels in every column set, the Q matrix is computed. And this way is implemented in this GPU only QR approach.

5 EVALUATION AND RESULTS

5.1 Evaluation Procedure

In this research, there are two different algorithms to be evaluated. First one is LU factorization and the next one is QR factorization. So the two factorization algorithms are evaluated separately. Following scenarios/points will be evaluated in this research evaluation.

In LU and QR factorizations, currently available expensive CPU-to-GPU communications and tuning challenges are going to be evaluated in the CUDA platform in terms of the performance (execution runtimes) to perform the algorithms. LU and QR factorizations are going to be evaluated on a NVIDIA MX130 GPU and the implementation to be executed on the GPU only environment.

As a safety precaution to the hardware device (The computer), runtimes are to be taken in a much cooler environment to avoid the excessive heating of the hardware device.

To compare the results against our implementation, implement the best possible LU and QR factorizations on multicores as well. In such ways like using cuSOLVER, LAPACK and OpenMP too. The plan is to input the input matrices to the cuSOLVER, LAPACK and OpenMP implemented LU and QR factorization algorithms and get the results with spent time. And then execute the same input data to the GPU only implemented LU factorization and QR factorization algorithms and get the results along with the spent time. Now the evaluation can be performed to the two different algorithms in separate ways, in another words, a result analysis of GPU only implementation execution against both multicore implementation execution and the NVIDIA cuSOLVER GPU only implementation execution.

In this way, it possible and logical to perform the evaluation under these conditions and criteria to perform a valid evaluation. The same dataset is being used to both GPU only and multicore implementations due to the fact that the input can be kept as a constant, so the result should be different when the process is being different.

5.2 Result Analysis of GPU only LU Factorization Implementation

5.2.1 Accuracy

In order to validate the implemented GPU only LU factorization algorithm, **L** resulting matrix and **U** resulting matrix is separately multiplied using a third party application and compared with the input matrix. The input matrix is divided into 4 matrices and the output also a combination of 8 matrices. Structure of the input matrices are shown in Figure 5.1 below.

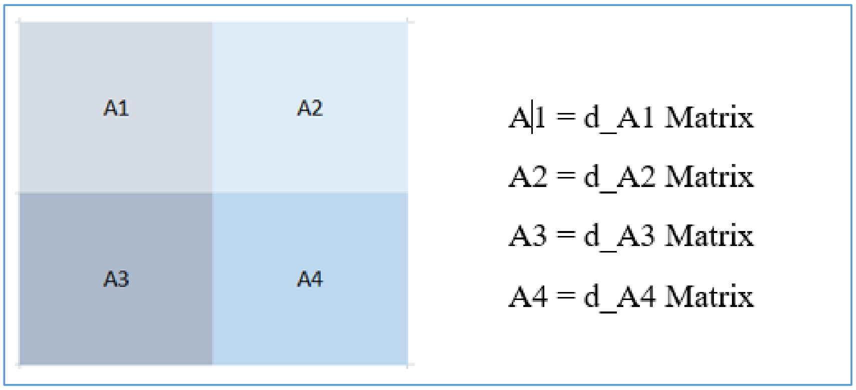


Figure 5.1 Structure of the Input matrix for LU factorization

<u>4x4 Matrix</u>

The input matrix and the output matrices for the 4x4 LU matrix factorization is shown below respectively in Figure 5.2 and Figure 5.3.

```
d_A1 Matrix
1.000   5.000
2.000   6.000
d_A2 Matrix
9.000   13.000
10.000  14.000
d_A3 Matrix
3.000   7.000
4.000   8.000
d_A4 Matrix
11.000  15.000
12.000  16.000
```

Figure 5.2 Initiated 4x4 input matrix

```
d_L1 Matrix
1.000   0.000
2.000   1.000
d_U1 Matrix
1.000   5.000
0.000   -4.000
d_L3 Matrix
3.000   2.000
4.000   3.000
d_U2 Matrix
9.000   13.000
-8.000  -12.000
d_L4 Matrix
1.000   0.000
0.000   1.000
d_U4 Matrix
0.000   0.000
0.000   0.000
```

Figure 5.3 Output matrices for the 4x4 LU factorization

Obtained the output matrices are multiplied separately and those results are shown in Figure 5.4 below.

Input matrix A:

1.000	0.000	0.000	0.000
2.000	1.000	0.000	0.000
3.000	2.000	1.000	0.000
4.000	3.000	0.000	1.000

Input matrix B:

1.000	5.000	9.000	13.000
0.000	-4.000	-8.000	-12.000
0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000

Matrix product A*B

1.000	5.000	9.000	13.000
2.000	6.000	10.000	14.000
3.000	7.000	11.000	15.000
4.000	8.000	12.000	16.000

Figure 5.4 Third party application Matrix Multiplication

Compare the accuracy of the computation is derived using Equation 24 and depicted below in Figure 5.5.

$$Accuracy = ThirdPartyMatrixMultiplication(A) - Input\ Matrix(B) \quad \text{Equation 24}$$

Input matrix A:

1.000	5.000	9.000	13.000
2.000	6.000	10.000	14.000
3.000	7.000	11.000	15.000
4.000	8.000	12.000	16.000

Input matrix B:

1.000	5.000	9.000	13.000
2.000	6.000	10.000	14.000
3.000	7.000	11.000	15.000
4.000	8.000	12.000	16.000

Matrix difference A + B

0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000

Figure 5.5 Compare the accuracy of 4x4 matrix

5.2.2 Speed

The runtime of the implemented LU factorization algorithm is shown below in Figure 5.6.

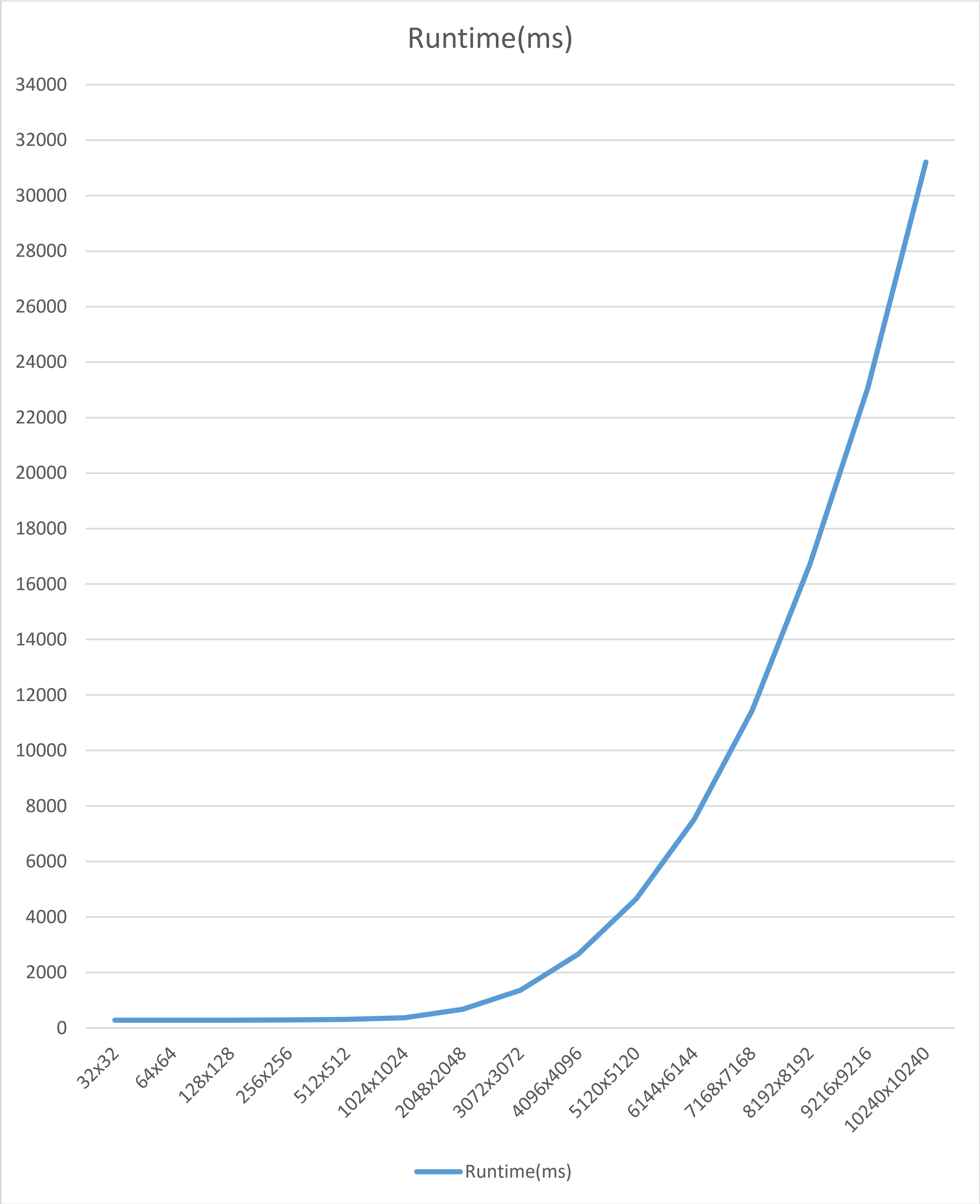


Figure 5.6 Runtimes of the GPU only LU Factorization

5.2.3 Performance

For different square matrix sizes, execution runtime of the suggested GPU only LU factorization along with the LAPACK, OpenMP and cuSOLVER implemented execution runtimes are listed below in Table 5.1. The algorithms are performed using an intel^{TM} core i5-8250U CPU and NVIDIA MX130 GPU.

Runtimes in milliseconds of LU factorization Implementations												
Matrix Size			LAPACK CPU LU		OpenMP CPU LU		cuSOLVER GPU LU		GPU Only Implemented LU	
32	x	32	0.25	ms	0.85	ms	157.15	ms	281.12	ms
64	x	64	0.67	ms	2.08	ms	158.40	ms	281.98	ms
128	x	128	3.91	ms	6.51	ms	159.61	ms	282.52	ms
256	x	256	12.15	ms	30.63	ms	165.45	ms	288.12	ms
512	x	512	42.82	ms	139.47	ms	177.16	ms	310.64	ms
1020	x	1020	225.58	ms	1,382.57	ms	242.01	ms	371.69	ms
2044	x	2044	1,599.40	ms	23,893.78	ms	551.58	ms	671.93	ms
3064	x	3064	5,552.87	ms	84,654.17	ms	1,255.73	ms	1,361.52	ms
4092	x	4092	13,388.54	ms	246,066.61	ms	2,531.16	ms	2,666.23	ms
5104	x	5104	26,219.65	ms	420,504.58	ms	4,626.95	ms	4,652.08	ms
6136	x	6136	45,493.97	ms	793,057.21	ms	7,600.19	ms	7,521.72	ms
7168	x	7168	72,237.90	ms	1,207,118.18	ms	11,558.07	ms	11,440.38	ms
8192	x	8192	107,381.24	ms	1,914,888.66	ms	17,391.10	ms	16,723.41	ms

Table 5.1 Runtimes in milliseconds of LU factorization Implementations

Using the runtimes in Table 5.1 derived performance of the GPU only implemented LU factorization against other implementations are list below in Table 5.2 below.

			Performance Against GPU Only Implemented LU					
Matrix Size			LAPACK CPU LU		OpenMP CPU LU		cuSOLVER GPU LU	
32	x	32	-280.87	ms	-280.27	ms	-123.97	ms
64	x	64	-281.31	ms	-279.90	ms	-123.58	ms
128	x	128	-278.61	ms	-276.01	ms	-122.91	ms
256	x	256	-275.97	ms	-257.49	ms	-122.67	ms
512	x	512	-267.82	ms	-171.17	ms	-133.48	ms
1024	x	1024	-146.11	ms	1,010.88	ms	-129.68	ms
2048	x	2048	927.47	ms	23,221.85	ms	-120.35	ms
3072	x	3072	4,191.35	ms	83,292.65	ms	-105.79	ms
4096	x	4096	10,722.31	ms	243,400.38	ms	-135.07	ms
5120	x	5120	21,567.57	ms	415,852.50	ms	-25.13	ms
6144	x	6144	37,972.25	ms	785,535.49	ms	78.47	ms
7168	x	7168	60,797.52	ms	1,195,677.80	ms	117.69	ms
8192	x	8192	90,657.83	ms	1,898,165.25	ms	667.69	ms

Table 5.2 Performance of the GPU only Implemented LU Factorization Algorithm

When the matrix size gets larger the suggested GPU only LU factorization algorithm starting to perform better in terms of the runtime. The algorithm started to perform better in the matrix size of 1024x1024 against the OpenMP implementation and in the matrix size 6144x 6144 and in above sizes, suggested LU factorization algorithm outperform LAPACK LU implementation, OpenMP LU implementation and NVIDIA cuSOLVER implementation as shown in Table 5.2.

5.3 Result Analysis of GPU only QR Factorization Implementation

5.3.1 Accuracy

In order to validate the implemented GPU only QR factorization algorithm, **Q** resulting matrix and **R** resulting matrix is separately multiplied using a third party application and compared with the input matrix.

<u>4x4 Matrix</u>

The input matrix and the output matrices for the 4x4 QR matrix factorization is shown below respectively in Figure 5.7 and Figure 5.8.

```
A Matrix 4 x 4 :
 0.000000  4.000000  8.000000 12.000000
 1.000000  5.000000  9.000000 13.000000
 2.000000  6.000000 10.000000 14.000000
 3.000000  7.000000 11.000000 15.000000
```

Figure 5.7 Initiated 4x4 input matrix

```
R Matrix 4 x 4 :
-3.741657 -10.155927 -16.570196 -22.984465
 0.000000 -4.780914 -9.561830 -14.342741
 0.000000  0.000000 -0.000002 -0.000001
 0.000000  0.000000  0.000000  0.000000

 Time to compute the 4x4 Q matrix = 0.545979 ms
QMatrix 4 x 4 :
 0.000000 -0.836660  0.543821  0.065259
-0.267261 -0.478091 -0.676453 -0.492353
-0.534522 -0.119523 -0.278557  0.788927
-0.801784  0.239046  0.411189 -0.361834
```

Figure 5.8 Output matrices for the 4x4 QR factorization

Obtained output matrices are multiplied separately and those results are shown in Figure 5.9 below.

Input matrix A:

0.000	-0.837	0.544	0.065
-0.267	-0.478	-0.676	-0.492
-0.535	-0.120	-0.279	0.789
-0.802	0.239	0.411	-0.362

Input matrix B:

-3.742	-10.156	-16.570	-22.984
0.000	-4.781	-9.562	-14.343
0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000

Matrix product A*B

0.000	4.000	8.000	12.000
1.000	5.000	9.000	13.000
2.000	6.000	10.000	14.000
3.000	7.000	11.000	15.000

Figure 5.9 Third party application Matrix Multiplication

Compare the accuracy of the computation is derived using Equation 24 and depicted below in Figure 5.10.

Input matrix A:

0.000	4.000	8.000	12.000
1.000	5.000	9.000	13.000
2.000	6.000	10.000	14.000
3.000	7.000	11.000	15.000

Input matrix B:

0.000	4.000	8.000	12.000
1.000	5.000	9.000	13.000
2.000	6.000	10.000	14.000
3.000	7.000	11.000	15.000

Matrix difference A + B

0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000

Figure 5.10 Compare the accuracy of 4x4 matrix

5.3.2 Speed

The runtime of the implemented QR factorization algorithm is shown below in Figure 5.11.

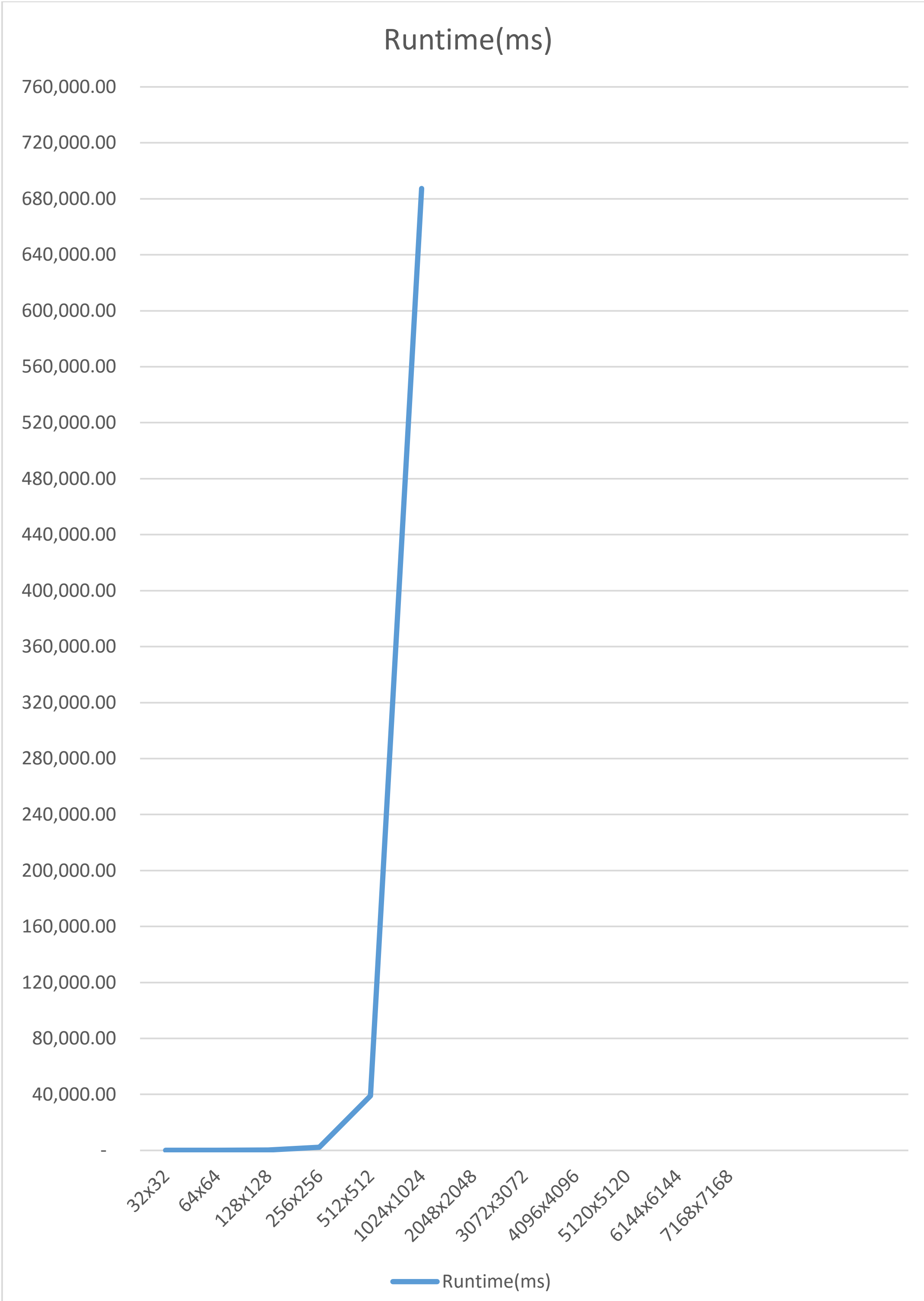


Figure 5.11 Runtimes of the GPU only QR Factorization

5.3.3 Performance

The same approach has been used for the suggested GPU only QR factorization algorithm implementation runtime along with the LAPACK, OpenMP and cuSOLVER implemented QR factorization execution runtimes are listed below in Table 5.3. The algorithms are performed using an intel™ core i5-8250U CPU and NVIDIA MX130 GPU.

Runtimes in milliseconds of QR factorization Implementations											
Matrix Size			LAPACK CPU QR		OpenMP CPU QR		cuSOLVER GPU QR		GPU Only Implemented QR		
32	x	32	0.43	ms	1.79	ms	292.37	ms		2.95	ms
64	x	64	1.74	ms	4.28	ms	292.74	ms		17.65	ms
128	x	128	7.48	ms	13.38	ms	300.39	ms		200.23	ms
256	x	256	32.03	ms	48.15	ms	331.07	ms		2,218.63	ms
512	x	512	168.06	ms	276.04	ms	403.56	ms		38,918.41	ms
1020	x	1020	1,345.56	ms	1,992.98	ms	883.40	ms		687,278.58	ms
2044	x	2044	13,466.73	ms	16,576.75	ms	4,252.15	ms	N/A		ms
3064	x	3064	48,969.83	ms	53,317.38	ms	13,205.76	ms	N/A		ms
4092	x	4092	120,258.09	ms	116,194.27	ms	29,902.91	ms	N/A		ms
5104	x	5104	238,777.74	ms	216,315.42	ms	58,918.76	ms	N/A		ms
6136	x	6136	413,846.63	ms	361,181.52	ms	100,100.82	ms	N/A		ms
7168	x	7168	663,353.07	ms	561,909.18	ms	158,961.61	ms	N/A		ms

Table 5.3 Runtimes in milliseconds of QR factorization Implementations

Using the runtimes in Table 5.3 derived performance of the GPU only implemented QR factorization against other implementations are list below in Table 5.4 below. After analyzing these runtimes against the suggested GPU only QR factorization implementation, there are no matrix sizes available which outperform the runtimes with LAPACK, OpenMP and cuSOLVER implemented QR factorization runtimes. But for every matrix size up to 128x128 matrix, the suggested GPU only QR factorization algorithm performs better than the cuSOLVER library related QR factorization implementation runtimes.

With the suggested GPU only QR factorization implementation, it was only possible to execute matrix sizes up to 1024x1024. Beyond this matrix size, QR factorization will take a massive time to execute the algorithm and mostly failed to produce the two output matrices. And the problem identified here was a matrix multiplication with another computationally expensive code when deriving the Q matrix in the QR factorization.

			Performance Against GPU Only Implemented QR		
Matrix Size			LAPACK CPU QR	OpenMP CPU QR	cuSOLVER GPU QR
32	x	32	-2.51 ms	-1.16 ms	289.42 ms
64	x	64	-15.91 ms	-13.37 ms	275.09 ms
128	x	128	-192.75 ms	-186.84 ms	100.16 ms
256	x	256	-2,186.61 ms	-2,170.48 ms	-1,887.56 ms
512	x	512	-38,750.35 ms	-38,642.37 ms	-38,514.85 ms
1020	x	1020	-685,933.02 ms	-685,285.60 ms	-686,395.18 ms
2044	x	2044	N/A ms	N/A ms	N/A ms
3064	x	3064	N/A ms	N/A ms	N/A ms
4092	x	4092	N/A ms	N/A ms	N/A ms
5104	x	5104	N/A ms	N/A ms	N/A ms
6136	x	6136	N/A ms	N/A ms	N/A ms
7168	x	7168	N/A ms	N/A ms	N/A ms

Table 5.4 Performance of the GPU only Implemented QR Factorization Algorithm

6 CONCLUSION AND FUTURE WORK

6.1 Conclusion

The work described in this thesis considered developing GPU only implementations of LU and QR factorization algorithms for high performance. This research project aimed to introduce new GPU only implementations with the CUDA programming language in an interactive way with using the kernel function calls which support the parallel code execution.

The research focused on new implementations for executing the GPU only LU factorization and QR factorization using the matrix as small blocks and by using these matrix blocks to perform the factorizations. This is known as block matrix factorization. Parallel computing is a popular research area which combines both the fields of computer science and parallel computing. Implemented GPU only LU and QR factorization algorithms are focused on factorization matrices of the square matrices. Based on the input matrix the algorithm will define the block size and perform the steps accordingly. The input matrix is first copied into the GPU memory and the GPU only algorithms are processing the factorization steps using the GPU memory until the factorization is completed.

The research was conducted in a Linux environment with a NVIDIA MX130 GPU. The reason to choose this GPU for the research is NVIDIA MX130 is a CUDA enabled, well-constructed and relatively inexpensive graphics card. The main discussion carried was about two different ways of implementing LU and QR factorizations for high-performance completely using GPU only executions [2], [8], [9], with the advantage gained through the CUDA programming platform. The problem attempted in this project is novel because there is a need for the development of high-performance LU and QR factorizations using GPU only implementations.

6.2 Achievements

Overall, LU factorization and QR factorization algorithms are completely executed on the GPU using the proposed block matrix factorization concept implementation. Due to this reason, expensive CPU-GPU communication has been eliminated in both LU and QR factorization implementations. As a result of this GPU only implementation, both panel factorization and trailing matrix update is processed in the GPU, using different GPU streams. For LU factorization, our implementation starts to perform well with the square matrix 6144 and upwards. Also, this research work was able to capture a few areas/sections of computationally expensive calculations. Suggested implementation was focused to reduce the complexity without affecting the accuracy of the code.

Opportunity to contribute to a state of the art technology which would be the next generation of computer science and parallel computing area was another satisfactory achievement.

6.3 Problems Encountered and Limitations

Only the square matrices were considered in this research. Since the block matrix factorization concept has been used for this research the input matrix should be able to be represented as a 2^n value especially for the LU factorization implementation in order to execute the factorization.

With the suggested GPU only QR factorization implementation, it was only possible to execute matrix sizes up to 1024x1024. Beyond this matrix size, QR factorization will take a massive time to execute the algorithm and mostly failed to produce the two output matrices. And the problem identified here was a matrix multiplication with another computationally expensive code when deriving the Q matrix in the QR factorization. This is the main drawback of this computation towards performance.

With the execution of these two implementations, computer devices are emitting a considerable amount of heat. So the algorithms are recommended to be executed in much cooler environmental conditions to minimize the risk of damaging the computer devices.

6.4 Lessons Learnt and Contributions

The research work gave a vast amount of research and development experience in the parallel computing and computer science domains. Exposure to the online communities in those disciplines provided opportunities to acquire expertise knowledge to accomplish the initial objectives and the main aim.

With regard to the technological aspect, it was a spectacular experience to gather new knowledge in leading-edge technology to provide something useful for society.

Performance analysis of suggested GPU only LU and QR factorization implementation runtimes against the NVIDIA cuSOLVER, OpenMP and LAPACK LU and QR factorization runtimes could be used by the research community. It is hoped that the work mentioned in this thesis contributes to both the fields of parallel computing and computer science.

6.5 Future Work

This research leaves a lot of room for further extensions and improvement in both LU and QR factorization algorithms using on GPU only executions. For example, implemented factorization algorithms are only able to process square matrices with even numbers for GPU only execution. Other types of matrices are to be implemented as future work.

Further research work could be carried out on deriving the Q matrix in the QR factorization in a much faster and an optimized way because the current implementation of deriving the Q matrix in this research is computationally very expensive and algorithm is unable to process after 1024x1024 matrix size. As an improvement, it can further develop to minimize the runtime and maximize the performance to derive the Q matrix which will make the GPU only QR factorization algorithm more efficient as the end result.

7 REFERENCES

[1] D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors*, 2nd edition. Elsevier Inc., 2013.

[2] A. Haidar, A. Abdelfatah, S. Tomov, and J. Dongarra, "High-performance Cholesky Factorization for GPU-only Execution," in *Proceedings of the General Purpose GPUs*, New York, NY, USA, 2017, pp. 42–52.

[3] J. J. Dongarra and V. Eijkhout, "Numerical linear algebra algorithms and software," *J. Comput. Appl. Math.*, vol. 123, no. 1, pp. 489–514, Nov. 2000.

[4] C. Ozcan and B. Sen, "Investigation of the performance of LU decomposition method using CUDA," *Procedia Technol.*, vol. 1, pp. 50–54, Jan. 2012.

[5] "LU decomposition," *Wikipedia*. 27-Jul-2018.

[6] "QR decomposition," *Wikipedia*. 03-Jun-2018.

[7] W. W. Hwu, *GPU Computing Gems*, Emerald Edition. Elsevier Inc., 2011.

[8] D. B. Gajić, R. S. Stanković, and M. Radmanović, "A performance analysis of computing the LU and the QR matrix decompositions on the CPU and the GPU," *Int. J. Reason.-Based Intell. Syst.*, vol. 9, no. 2, pp. 114–121, Jan. 2017.

[9] J. Dongarra, S. Tomov, and P. Du, "Providing GPU Capability to LU and QR within the ScaLAPACK Framework," 2012.

[10] Y. Jia, P. Luszczek, and J. Dongarra, "Multi-GPU Implementation of LU Factorization," *Procedia Comput. Sci.*, vol. 9, pp. 106–115, Jan. 2012.

[11] R. Andrew and N. Dingle, "Implementing QR factorization updating algorithms on GPUs," *Parallel Comput.*, vol. 40, no. 7, pp. 161–172, Jul. 2014.

[12] E. D'Azevedo and J. C. Hill, "Parallel LU Factorization on GPU Cluster," *Procedia Comput. Sci.*, vol. 9, pp. 67–75, Jan. 2012.

[13] "Data parallelism," *Wikipedia*. 17-Oct-2018.

[14] Z. Lin, Y. Chen, Y. Zhang, X. Zhao, and H. Zhang, "An Efficient GPU-Based Out-of-Core LU Solver of Parallel Higher-Order Method of Moments for Solving Airborne Array Problems," *International Journal of Antennas and Propagation*, 2017. [Online]. Available: https://www.hindawi.com/journals/ijap/2017/4309381/. [Accessed: 10-Nov-2018].

[15] A. Haidar, A. Abdelfattah, M. Zounon, S. Tomov, J. Dongarra, and J. Dongarra, "A Guide for Achieving High Performance with Very Small Matrices on GPU: A Case Study of Batched LU and Cholesky Factorizations," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 5, pp. 973–984, 2018.

[16] A. Abdelfattah, A. Haidar, S. Tomov, and J. Dongarra, "Factorization and Inversion of a Million Matrices using GPUs: Challenges and Countermeasures," *Procedia Comput. Sci.*, vol. 108, pp. 606–615, Jan. 2017.

[17] S. N. Yeralan, T. A. Davis, W. M. Sid-Lakhdar, and S. Ranka, "Algorithm 980: Sparse QR Factorization on the GPU," *ACM Trans Math Softw*, vol. 44, no. 2, pp. 17:1–17:29, Aug. 2017.

[18] C. Chen, J. Fang, T. Tang, and C. Yang, "LU factorization on heterogeneous systems: an energy-efficient approach towards high performance," *Computing*, vol. 99, no. 8, pp. 791–811, Aug. 2017.

[19] F. Loh, P. Ramanathan, and K. K. Saluja, "Transient Fault Resilient QR Factorization on GPUs," in *Proceedings of the 5th Workshop on Fault Tolerance for HPC at eXtreme Scale*, New York, NY, USA, 2015, pp. 63–70.

[20] I. Yamazaki, S. Tomov, and J. Dongarra, "Stability and Performance of Various Singular Value QR Implementations on Multicore CPU with a GPU," *ACM Trans Math Softw*, vol. 43, no. 2, pp. 10:1–10:18, Sep. 2016.

[21] W. Tan, S. Chang, L. Fong, C. Li, Z. Wang, and L. Cao, "Matrix Factorization on GPUs with Memory Optimization and Approximate Computing," in *Proceedings of the 47th International Conference on Parallel Processing*, New York, NY, USA, 2018, pp. 26:1–26:10.

[22] G. Shabat, Y. Shmueli, Y. Aizenbud, and A. Averbuch, "Randomized LU decomposition," *Appl. Comput. Harmon. Anal.*, vol. 44, no. 2, pp. 246–272, Mar. 2018.

[23] R. G. McClarren, "Chapter 8 - LU Factorization and Banded Matrices," in *Computational Nuclear Engineering and Radiological Science Using Python*, R. G. McClarren, Ed. Academic Press, 2018, pp. 131–144.

[24] E. Coleman and M. Sosonkina, "Self-stabilizing fine-grained parallel incomplete LU factorization," *Sustain. Comput. Inform. Syst.*, vol. 19, pp. 291–304, Sep. 2018.

[25] J. Thunberg, J. Markdahl, and J. Gonçalves, "Dynamic controllers for column synchronization of rotation matrices: A QR-factorization approach," *Automatica*, vol. 93, pp. 20–25, Jul. 2018.

[26] C. Martins, P. Tomas, and R. Chaves, "Parallelization of the LU Decomposition on Heterogeneous Systems," p. 10.

[27] M. Anderson, G. Ballard, J. Demmel, and K. Keutzer, "Communication-Avoiding QR Decomposition for GPUs," in *2011 IEEE International Parallel & Distributed Processing Symposium*, Anchorage, AK, USA, 2011, pp. 48–58.

[28] "cuSOLVER," *NVIDIA Developer*, 12-Jan-2015. [Online]. Available: https://developer.nvidia.com/cusolver. [Accessed: 21-Feb-2019].

[29] "Experiment," *Wikipedia*. 18-Nov-2018.

[30] "Benefits and Limitations of Experimental Research - Center for Innovation in Research and Teaching." [Online]. Available: https://cirt.gcu.edu/research/developmentresources/research_ready/experimental/benefits_limits. [Accessed: 20-Nov-2018].

[31] "LAPACK — Linear Algebra PACKage." [Online]. Available: http://www.netlib.org/lapack/. [Accessed: 31-Dec-2018].

[32] "OpenMP," *Wikipedia*. 15-Nov-2018.

[33] J. J. Costa, T. Cortes, X. Martorell, E. Ayguade, and J. Labarta, "Running OpenMP applications efficiently on an everything-shared SDSM," *J. Parallel Distrib. Comput.*, vol. 66, no. 5, pp. 647–658, May 2006.

[34] MAD\jphoefli, “Cluster OpenMP* for Intel® Compilers,” 19:52:26 UTC. [Online]. Available: https://software.intel.com/en-us/articles/cluster-openmp-for-intel-compilers. [Accessed: 20-Nov-2018].

[35] “CUDA,” *Wikipedia*. 16-Nov-2018.

[36] R. Schreiber, “Block Algorithms for Parallel Machines,” in *Numerical Algorithms for Modern Parallel Computer Architectures*, 1988, pp. 197–207.

[37] R. Iakymchuk, E. S. Quintana-Ort, and E. Laure, “Towards Reproducible Blocked LU Factorization,” *ResearchGate*, May-2017. [Online]. Available: https://www.researchgate.net/publication/318123432_Towards_Reproducible_Blocked_LU_Factorization. [Accessed: 07-Apr-2019].

[38] J. R. Bischof, “A Block QR Factorization Algorithm Using Restricted Pivoting,” in *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*, New York, NY, USA, 1989, pp. 248–256.

[39] “Householder QR Factorization With Randomization for Column Pivoting (HQRRP),” *SIAM Journal on Scientific Computing 39(2):C96-C115*, Jan-2017. [Online]. Available: https://www.researchgate.net/publication/316056641_Householder_QR_Factorization_With_Randomization_for_Column_Pivoting_HQRRP. [Accessed: 07-Apr-2019].

[40] Introduction to Parallel Programming in OpenMP, *Parallel LU Factorization*. .

[41] Introduction to Parallel Programming in OpenMP, *Understanding LU Factorization*. .

[42] Toby Driscoll, *MATH426: Householder QR*. .

APPENDIX A – RUNTIME RESULTS

LU FACTORIZATION RUNTIMES

GPU only LU matrix factorization implementation runtimes are shown below in Table A.1.

CUDA My Implementation											
Matrix-Size	Copy-to-GPU	Initialize-Matrix	Allocate-Query-Space	A1 LU factorization	perform U2, L3 trsm	perform L3xU1 gemm	subtract matrix A4 - L3xU1	decompose matrix A4 - L3xU1	Copy-To-CPU	Total Time	
32 x 32	280.40	0.0253120	0.01	0.40	0.02	0.01	0.01	0.18	0.06	281.12	ms
64 x 64	280.70	0.0295680	0.01	0.63	0.08	0.02	0.01	0.40	0.10	281.98	ms
128 x 128	280.07	0.0043840	0.01	1.03	0.28	0.05	0.01	0.86	0.21	282.52	ms
256 x 256	282.47	0.0047040	0.01	1.75	1.16	0.21	0.01	1.86	0.65	288.12	ms
512 x 512	292.52	0.0046400	0.01	4.70	4.97	1.79	0.01	4.29	2.35	310.64	ms
1024 x 1024	297.15	0.0048000	0.02	12.39	27.58	12.82	0.01	14.72	7.00	371.69	ms
2048 x 2048	326.10	0.0094080	0.02	51.52	140.05	82.62	0.01	45.40	26.20	671.93	ms
3072 x 3072	380.59	0.0021440	0.03	130.66	393.06	276.64	0.01	124.70	55.83	1,361.52	ms
4096 x 4096	431.61	0.0061760	0.03	264.55	975.52	646.21	0.01	263.04	85.25	2,666.23	ms
5120 x 5120	512.94	0.0064320	0.03	488.13	1,773.22	1,259.72	0.01	488.79	129.23	4,652.08	ms
6144 x 6144	612.91	0.0051520	0.03	819.82	2,906.25	2,173.02	0.01	824.45	185.22	7,521.72	ms
7168 x 7168	723.28	0.0048960	0.03	1,274.88	4,458.92	3,457.95	0.01	1,276.55	248.76	11,440.38	ms
8192 x 8192	861.87	0.0037760	0.04	1,877.61	6,603.84	5,173.85	0.01	1,884.17	322.02	16,723.41	ms
9216 x 9216	999.82	0.0043200	0.04	2,689.76	8,969.38	7,334.23	0.01	2,685.49	412.31	23,091.04	ms
10240 x 10240	1,167.59	0.0059200	0.04	3,682.59	12,114.58	10,060.73	0.01	3,676.58	505.63	31,207.76	ms

Table A.1 GPU only LU matrix factorization implementation runtimes

NVIDIA cuSOLVER LU factorization implementation runtimes are shown in Table A.2, OpenMP LU factorization implementation runtimes are shown in Table A.3 and LAPACK LU factorization implementation runtimes are shown in Table A.4 below.

cuSOLVER							
Matrix-Size	Copy-to-GPU	Initialize-Matrix	Allocate-Query-Space	LU Factorization	Copy-To-CPU	Total Time	
32 x 32	156.61	0.0243520	0.01	0.43	0.08	157.15	ms
64 x 64	157.45	0.0032320	0.01	0.78	0.15	158.40	ms
128 x 128	157.53	0.0038080	0.01	1.57	0.49	159.61	ms
256 x 256	159.15	0.0037120	0.01	4.54	1.74	165.45	ms
512 x 512	159.51	0.0048320	0.01	12.37	5.26	177.16	ms
1024 x 1024	170.97	0.0045440	0.01	54.83	16.20	242.01	ms
2048 x 2048	216.48	0.0040320	0.01	265.00	70.08	551.58	ms
3072 x 3072	291.45	0.0042240	0.01	820.87	143.39	1,255.73	ms
4096 x 4096	392.27	0.0043520	0.01	1,882.90	255.96	2,531.16	ms
5120 x 5120	528.15	0.0059840	0.02	3,720.74	378.04	4,626.95	ms
6144 x 6144	695.07	0.0051840	0.03	6,360.85	544.23	7,600.19	ms
7168 x 7168	886.16	0.0062080	0.04	9,942.04	729.92	11,558.07	ms
8192 x 8192	1,243.92	0.0082560	0.05	14,627.50	1519.63	17,391.10	ms

Table A.2 NVIDIA cuSOLVER LU factorization implementation runtimes

OpenMP						
Matrix-Size			Initialize-Matrix	LU Factorization	Total Time	
32	x	32	0.09	0.75	0.85	ms
64	x	64	0.34	1.74	2.08	ms
128	x	128	1.34	5.17	6.51	ms
256	x	256	5.32	25.31	30.63	ms
512	x	512	8.87	130.60	139.47	ms
1024	x	1024	31.42	1,351.15	1,382.57	ms
2048	x	2048	112.75	23,781.03	23,893.78	ms
3072	x	3072	194.13	84,460.04	84,654.17	ms
4096	x	4096	507.26	245,559.35	246,066.61	ms
5120	x	5120	672.91	419,831.67	420,504.58	ms
6144	x	6144	724.98	792,332.24	793,057.21	ms
7168	x	7168	979.00	1,206,139.18	1,207,118.18	ms
8192	x	8192	1,148.53	1,913,740.13	1,914,888.66	ms

Table A.3 OpenMP LU factorization implementation runtimes

LAPACK						
Matrix-Size			Initialize-Matrix	LU Factorization	Total Time	
32	x	32	0.05	0.20	0.25	ms
64	x	64	0.21	0.46	0.67	ms
128	x	128	1.26	2.65	3.91	ms
256	x	256	6.02	6.12	12.15	ms
512	x	512	8.60	34.22	42.82	ms
1024	x	1024	25.05	200.54	225.58	ms
2048	x	2048	83.85	1,515.55	1,599.40	ms
3072	x	3072	188.51	5,364.36	5,552.87	ms
4096	x	4096	297.44	13,091.10	13,388.54	ms
5120	x	5120	466.45	25,753.20	26,219.65	ms
6144	x	6144	653.47	44,840.50	45,493.97	ms
7168	x	7168	905.60	71,332.30	72,237.90	ms
8192	x	8192	1,173.34	106,207.90	107,381.24	ms

Table A.4 LAPACK LU factorization implementation runtimes

QR FACTORIZATION RUNTIMES

GPU only QR matrix factorization implementation runtimes are shown below in Table A.5. Unlike GPU only LU factorization implementation, several panel block sizes have used in this GPU only QR factorization implementation to derive the QR factorization runtime of a one particular matrix size. Records for the Table A.5 is derived using Table A.6 to Table A.11 and the fastest runtime of the matrix size will be represented as the records in the Table A.5 and as the final runtime of the QR factorization implementation runtimes.

CUDA My Implementation								
Matrix-Size		Initialize Matrix	compute R	Derive Upper Triangular R	Compute Q	Total Time for QR factorization		Block Size
32 x	32	0.021935	0.022888	0.004053	2.898932	2.95	ms	32x8
64 x	64	0.041962	0.209093	0.008821	17.392874	17.65	ms	64x4
128 x	128	0.022888	0.020027	0.003099	200.181961	200.23	ms	128x32
256 x	256	0.036001	0.156164	0.014067	2218.424082	2,218.63	ms	256x16
512 x	512	0.051022	0.563860	0.006914	38917.788982	38,918.41	ms	512x4
1024 x	1024	0.136852	1.153946	0.007153	687277.280092	687,278.58	ms	1024x4
2048 x	2048					N/A	ms	
3072 x	3072					N/A	ms	
4096 x	4096					N/A	ms	
5120 x	5120					N/A	ms	
6144 x	6144					N/A	ms	
7168 x	7168					N/A	ms	

Table A.5 GPU only QR factorization implementation runtimes

In the Table A.6 to Table A.11, the green colored record is the fastest matrix factorization for that particular matrix size and the gray color record is discarded due to the improper matrix size is shown in the result with respect to the selected block panel size.

CUDA My Implementation						32x32	
Block-Size	Initialize Matrix	compute R	Derive Upper Triangular R	Compute Q	Total Time for QR factorization		
8x4	0.02194	0.15998	0.00310	15.96594	16.15	ms	
16x4	0.02503	0.05317	0.00382	3.50213	3.58	ms	28x28
16x8	0.02503	0.04292	0.00405	6.49190	6.56	ms	
32x4	0.04602	0.11611	0.01001	4.62699	4.80	ms	
32x8	0.02194	0.02289	0.00405	2.89893	2.95	ms	
32x16	0.02694	0.01597	0.00382	3.17788	3.22	ms	

Table A.6 GPU only QR factorization for 32x32 matrix

CUDA My Implementation						64x64	
Block-Size	Initialize Matrix	compute R	Derive Upper Triangular R	Compute Q	Total Time for QR factorization		
8x4	0.02194	0.63896	0.00310	177.93703	178.60	ms	
16x4	0.02217	0.23508	0.00405	68.88604	69.15	ms	
16x8	0.02217	0.15807	0.00286	95.06202	95.25	ms	
32x4	0.02313	0.10514	0.00286	24.66202	24.79	ms	60x60
32x8	0.02217	0.05102	0.00286	20.16115	20.24	ms	56x56
32x16	0.02313	0.04697	0.00286	98.22893	98.30	ms	
64x4	0.041962	0.209093	0.008821	17.392874	17.65	ms	
64x8	0.038147	0.072002	0.006914	35.015106	35.13	ms	
64x16	0.071049	0.089169	0.015974	20.60008	20.78	ms	
64x32	0.025034	0.010014	0.003099	N/A	N/A	ms	

Table A.7 GPU only QR factorization for 64x64 matrix

CUDA My Implementation						128x128	
Block-Size	Initialize Matrix	compute R	Derive Upper Triangular R	Compute Q	Total Time for QR factorization		
8x4	0.02694	2.33316	0.00310	2,530.56884	2,532.93	ms	
16x4	0.02384	0.79393	0.00286	870.48101	871.30	ms	124x124
16x8	0.02384	0.61989	0.00382	1,264.92786	1,265.58	ms	
32x4	0.02384	0.32997	0.00310	346.39382	346.75	ms	116x116
32x8	0.02384	0.22388	0.00310	470.63899	470.89	ms	
32x16	0.02289	0.15807	0.00310	831.46095	831.65	ms	
64x4	0.023127	0.2141	0.003099	247.999907	248.24	ms	124x124
64x8	0.022888	0.105858	0.004053	297.724962	297.86	ms	120x120
64x16	0.02408	0.056028	0.004053	253.201962	253.29	ms	112x112
64x32	0.022888	0.030041	0.003099	N/A	N/A	ms	
128x4	0.025988	0.145197	0.004053	208.918095	209.09	ms	
128x8	0.025988	0.072956	0.004053	210.35099	210.45	ms	
128x16	0.025988	0.037909	0.003815	203.536987	203.60	ms	
128x32	0.022888	0.020027	0.003099	200.181961	200.23	ms	
128x64	0.025034	N/A	N/A	N/A	N/A	ms	

Table A.8 GPU only QR factorization for 128x128 matrix

Block-Size	Initialize Matrix	compute R	Derive Upper Triangular R	Compute Q	Total Time for QR factorization		
8x4	0.02503	93.09006	0.00906	69,751.92	69,845.05	ms	
16x4	0.06485	16.16812	0.03099	24,107.09	24,123.35	ms	
16x8	0.03099	3.51787	0.00882	35,352.23	35,355.79	ms	
32x4	0.03099	2.15507	0.01001	10,903.41	10,905.60	ms	
32x8	0.02980	1.16921	0.00906	14,649.02	14,650.23	ms	248x248
32x16	0.06509	2.67100	0.02885	19,461.76	19,464.52	ms	
64x4	0.037909	1.5769	0.014067	6,651.15	6,652.78	ms	244x244
64x8	0.030041	0.517845	0.010014	6,285.53	6,286.09	ms	232x232
64x16	0.030994	0.334978	0.00906	7,296.48	7,296.85	ms	
64x32	0.030994	0.163078	0.010014	N/A	N/A	ms	
128x4	0.025034	0.439167	0.00596	4,469.52	4,469.99	ms	252x252
128x8	0.062943	0.958204	0.030041	4,033.05	4,034.10	ms	248x248
128x16	0.027895	0.106096	0.00596	3,786.72	3,786.86	ms	240x240
128x32	0.061989	0.226974	0.032187	3,169.65	3,169.97	ms	224x224
128x64	0.028133	N/A	N/A	N/A	N/A	ms	
256x4	0.056028	1.005173	0.008106	2,226.37	2,227.44	ms	
256x8	0.025988	0.149012	0.006199	2,221.76	2,221.94	ms	
256x16	0.036001	0.156164	0.014067	2,218.42	2,218.63	ms	
256x32	0.025034	N/A	N/A	N/A	N/A	ms	
256x64	0.025988	N/A	N/A	N/A	N/A	ms	
256x128	0.026941	N/A	N/A	N/A	N/A	ms	

Table A.9 GPU only QR factorization for 256x256 matrix

CUDA My Implementation						512x512	
Block-Size	Initialize Matrix	compute R	Derive Upper Triangular R	Compute Q	Total Time for QR factorization		
8x4	Taking too much time to execute					ms	
16x4						ms	
16x8						ms	
32x4						ms	
32x8	0.05508	67.49296	0.00906	436,704.62	436,772.18	ms	
32x16	0.05102	13.15880	0.45610	639,286.55	639,300.21	ms	
64x4	0.048161	9.462833	0.008106	130,611.33	130,620.85	ms	484x484
64x8	0.052214	1.876116	0.007868	202,622.39	202,624.33	ms	
64x16	0.051975	0.937939	0.007868	180,001.55	180,002.54	ms	496x496
64x32	0.050068	0.395775	0.007153	N/A	N/A	ms	
128x4	0.049829	1.577854	0.006914	85,914.92	85,916.55	ms	500x500
128x8	0.051022	0.849962	0.008106	79,064.60	79,065.51	ms	488x488
128x16	0.045061	0.329018	0.006914	59,323.76	59,324.14	ms	464x464
128x32	0.051975	0.289917	0.008821	121,500.22	121,500.57	ms	
128x64	0.051022	N/A	N/A	N/A	N/A	ms	
256x4	0.051975	1.04785	0.007868	53,858.23	53,859.34	ms	508x508
256x8	0.096083	1.889944	0.032902	51,965.18	51,967.20	ms	504x504
256x16	0.051975	0.290155	0.00906	47,678.35	47,678.70	ms	496x496
256x32	0.048161	N/A	N/A	N/A	N/A	ms	
256x64	0.043869	N/A	N/A	N/A	N/A	ms	
256x128	0.052929	N/A	N/A	N/A	N/A	ms	
512x4	0.051022	0.56386	0.006914	38917.78898	38,918.41	ms	
512x8	0.052214	0.283003	0.006914	38931.33593	38,931.68	ms	
512x16	0.050783	N/A	N/A	N/A	N/A	ms	
512x32	0.051975	N/A	N/A	N/A	N/A	ms	
512x64	0.056028	N/A	N/A	N/A	N/A	ms	
512x128	0.053167	N/A	N/A	N/A	N/A	ms	
512x256	0.054836	N/A	N/A	N/A	N/A	ms	

Table A.10 GPU only QR factorization for 512x512 matrix

CUDA My Implementation						1024x1024	
Block-Size	Initialize Matrix	compute R	Derive Upper Triangular R	Compute Q	Total Time for QR factorization		
8x4	Taking too much time to execute					ms	
16x4						ms	
16x8						ms	
32x4						ms	
32x8						ms	
32x16						ms	
64x4						ms	
64x8						ms	
64x16						ms	
64x32						ms	
128x4						ms	
128x8						ms	
128x16						ms	
128x32						ms	
128x64						ms	
256x4						ms	
256x8						ms	
256x16						ms	976x976
256x32	0.113964	N/A	N/A	N/A	N/A	ms	
256x64	0.134945	N/A	N/A	N/A	N/A	ms	
256x128	0.134945	N/A	N/A	N/A	N/A	ms	
512x4	0.135183	1.694918	0.00596	1,010,080.96	1,010,082.80	ms	
512x8	0.137091	0.874996	0.006914	1,019,371.22	1,019,372.24	ms	
512x16	0.141859	N/A	N/A	N/A	N/A	ms	
512x32	0.128984	N/A	N/A	N/A	N/A	ms	
512x64	0.123978	N/A	N/A	N/A	N/A	ms	
512x128	0.111818	N/A	N/A	N/A	N/A	ms	
512x256	0.135899	N/A	N/A	N/A	N/A	ms	
1024x4	0.136852	1.153946	0.007153	687277.2801	687278.578	ms	
1024x8	0.14019	N/A	N/A	N/A	N/A	ms	
1024x16	0.138998	N/A	N/A	N/A	N/A	ms	
1024x32	0.135899	N/A	N/A	N/A	N/A	ms	
1024x64	0.173092	N/A	N/A	N/A	N/A	ms	
1024x128	0.138044	N/A	N/A	N/A	N/A	ms	

Table A.11 GPU only QR factorization for 1024x1024 matrix

NVIDIA cuSOLVER QR factorization implementation runtimes are shown in Table A.12, OpenMP QR factorization implementation runtimes are shown in Table A.13 and LAPACK QR factorization implementation runtimes are shown in Table A.14 below.

CUDA									
Matrix-Size			Copy-to-GPU	Initialize-Matrix	Allocate-Query-Space	QR Factorization	Copy-To-CPU	Total Time	
32	x	32	292.01	0.0249600	0.01	0.29	0.04	292.37	ms
64	x	64	291.26	0.0051840	0.01	1.38	0.08	292.74	ms
128	x	128	294.94	0.0052800	0.18	5.00	0.26	300.39	ms
256	x	256	296.91	0.0046720	0.18	27.04	6.94	331.07	ms
512	x	512	298.31	0.0056960	0.18	97.95	7.11	403.56	ms
1024	x	1024	303.83	0.0061120	0.20	568.91	10.45	883.40	ms
2048	x	2048	332.04	0.0077200	0.24	3,865.96	53.90	4252.15	ms
3072	x	3072	406.73	0.0047360	0.24	12,695.76	103.03	13,205.76	ms
4096	x	4096	471.79	0.0051520	0.24	29,259.33	171.54	29,902.91	ms
5120	x	5120	553.65	0.0073200	0.25	58,105.42	259.43	58,918.76	ms
6144	x	6144	659.70	0.0076720	0.25	99,073.90	366.96	100,100.82	ms
7168	x	7168	796.14	0.0079600	0.26	157,671.11	494.09	158,961.61	ms
8192	x	8192	947.53	0.0079400	0.26	232,099.32	647.93	233,695.05	ms

Table A.12 NVIDIA cuSOLVER QR factorization implementation runtimes

OpenMP						
Matrix-Size			Initialize variables	QR Factorization	Total Time	
32	x	32	0.63	1.16	1.79	ms
64	x	64	0.66	3.62	4.28	ms
128	x	128	0.70	12.68	13.38	ms
256	x	256	0.73	47.43	48.15	ms
512	x	512	0.66	275.38	276.04	ms
1024	x	1024	0.72	1,992.26	1,992.98	ms
2048	x	2048	0.61	16,576.14	16,576.75	ms
3072	x	3072	0.61	53,316.76	53,317.38	ms
4096	x	4096	0.92	116,193.35	116,194.27	ms
5120	x	5120	0.69	216,314.74	216,315.42	ms
6144	x	6144	0.84	361,180.68	361,181.52	ms
7168	x	7168	0.74	561,908.44	561,909.18	ms

Table A.13 OpenMP QR factorization implementation runtimes

LAPACK						
Matrix-Size			Initialize-Matrix	QR Factorization	Total Time	
32	x	32	0.09	0.35	0.43	ms
64	x	64	0.25	1.49	1.74	ms
128	x	128	0.89	6.59	7.48	ms
256	x	256	5.55	26.48	32.03	ms
512	x	512	9.02	159.03	168.06	ms
1024	x	1024	27.19	1,318.37	1,345.56	ms
2048	x	2048	84.82	13,381.91	13,466.73	ms
3072	x	3072	173.83	48,796.00	48,969.83	ms
4096	x	4096	296.89	119,961.20	120,258.09	ms
5120	x	5120	465.74	238,312.00	238,777.74	ms
6144	x	6144	658.63	413,188.00	413,846.63	ms
7168	x	7168	900.07	662,453.00	663,353.07	ms

Table A.14 LAPACK QR factorization implementation runtimes

APPENDIX B – RESULTS ACCURACY

GPU ONLY LU FACTORIZATION IMPLEMENTATION

8x8 Matrix

The input matrix and the output matrices for the 8x8 LU matrix factorization is shown below respectively in Figure B.1 and Figure B.2.

```
d_A1 Matrix
1.000    9.000    17.000   25.000
2.000    10.000   18.000   26.000
3.000    11.000   19.000   27.000
4.000    12.000   20.000   28.000
d_A2 Matrix
33.000   41.000   49.000   57.000
34.000   42.000   50.000   58.000
35.000   43.000   51.000   59.000
36.000   44.000   52.000   60.000
d_A3 Matrix
5.000    13.000   21.000   29.000
6.000    14.000   22.000   30.000
7.000    15.000   23.000   31.000
8.000    16.000   24.000   32.000
d_A4 Matrix
37.000   45.000   53.000   61.000
38.000   46.000   54.000   62.000
39.000   47.000   55.000   63.000
40.000   48.000   56.000   64.000
```

Figure B.1 Initiated 8x8 input matrix

```
d_L1 Matrix
1.000    0.000    0.000    0.000
2.000    1.000    0.000    0.000
3.000    2.000    1.000    0.000
4.000    3.000    0.000    1.000
d_U1 Matrix
1.000    9.000    17.000   25.000
0.000    -8.000   -16.000  -24.000
0.000    0.000    0.000    0.000
0.000    0.000    0.000    0.000
d_L3 Matrix
5.000    4.000    0.000    0.000
6.000    5.000    0.000    0.000
7.000    6.000    0.000    0.000
8.000    7.000    0.000    0.000
```

```
d_U2 Matrix
33.000   41.000   49.000   57.000
-32.000  -40.000  -48.000  -56.000
0.000    0.000    0.000    0.000
0.000    0.000    0.000    0.000
d_L4 Matrix
1.000    0.000    0.000    0.000
0.000    1.000    0.000    0.000
0.000    0.000    1.000    0.000
0.000    0.000    0.000    1.000
d_U4 Matrix
0.000    0.000    0.000    0.000
0.000    0.000    0.000    0.000
0.000    0.000    0.000    0.000
0.000    0.000    0.000    0.000
```

Figure B.2 Output matrices for 8x8 LU factorization

Obtained the output matrices are multiplied separately and those results are shown in Figure B.3 below.

Input matrix A:

1.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
2.000	1.000	0.000	0.000	0.000	0.000	0.000	0.000
3.000	2.000	1.000	0.000	0.000	0.000	0.000	0.000
4.000	3.000	0.000	1.000	0.000	0.000	0.000	0.000
5.000	4.000	0.000	0.000	1.000	0.000	0.000	0.000
6.000	5.000	0.000	0.000	0.000	0.000	0.000	0.000
7.000	6.000	0.000	0.000	0.000	0.000	0.000	1.000
8.000	7.000	0.000	0.000	0.000	1.000	0.000	0.000

Input matrix B:

1.000	9.000	17.000	25.000	33.000	41.000	49.000	57.000
0.000	-8.000	-16.000	-24.000	-32.000	-40.000	-48.000	-56.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

Matrix product A*B

1.000	9.000	17.000	25.000	33.000	41.000	49.000	57.000
2.000	10.000	18.000	26.000	34.000	42.000	50.000	58.000
3.000	11.000	19.000	27.000	35.000	43.000	51.000	59.000
4.000	12.000	20.000	28.000	36.000	44.000	52.000	60.000
5.000	13.000	21.000	29.000	37.000	45.000	53.000	61.000
6.000	14.000	22.000	30.000	38.000	46.000	54.000	62.000
7.000	15.000	23.000	31.000	39.000	47.000	55.000	63.000
8.000	16.000	24.000	32.000	40.000	48.000	56.000	64.000

Figure B.3 Third party application Matrix Multiplication

Compare the accuracy of the computation is derived using Equation 24 and depicted below in Figure B.4.

Input matrix A:

1.000	9.000	17.000	25.000	33.000	41.000	49.000	57.000
2.000	10.000	18.000	26.000	34.000	42.000	50.000	58.000
3.000	11.000	19.000	27.000	35.000	43.000	51.000	59.000
4.000	12.000	20.000	28.000	36.000	44.000	52.000	60.000
5.000	13.000	21.000	29.000	37.000	45.000	53.000	61.000
6.000	14.000	22.000	30.000	38.000	46.000	54.000	62.000
7.000	15.000	23.000	31.000	39.000	47.000	55.000	63.000
8.000	16.000	24.000	32.000	40.000	48.000	56.000	64.000

Input matrix B:

1.000	9.000	17.000	25.000	33.000	41.000	49.000	57.000
2.000	10.000	18.000	26.000	34.000	42.000	50.000	58.000
3.000	11.000	19.000	27.000	35.000	43.000	51.000	59.000
4.000	12.000	20.000	28.000	36.000	44.000	52.000	60.000
5.000	13.000	21.000	29.000	37.000	45.000	53.000	61.000
6.000	14.000	22.000	30.000	38.000	46.000	54.000	62.000
7.000	15.000	23.000	31.000	39.000	47.000	55.000	63.000
8.000	16.000	24.000	32.000	40.000	48.000	56.000	64.000

Matrix difference A + B

0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

Figure B.4 Compare the accuracy of 8x8 matrix

16x16 Matrix

The input matrix and the output matrices for the 16x16 LU matrix factorization is shown below respectively in Figure B.5 and Figure B.6.

```
d_A1 Matrix
1.000   17.000  33.000  49.000  65.000  81.000  97.000  113.000
2.000   18.000  34.000  50.000  66.000  82.000  98.000  114.000
3.000   19.000  35.000  51.000  67.000  83.000  99.000  115.000
4.000   20.000  36.000  52.000  68.000  84.000  100.000 116.000
5.000   21.000  37.000  53.000  69.000  85.000  101.000 117.000
6.000   22.000  38.000  54.000  70.000  86.000  102.000 118.000
7.000   23.000  39.000  55.000  71.000  87.000  103.000 119.000
8.000   24.000  40.000  56.000  72.000  88.000  104.000 120.000
d_A2 Matrix
129.000 145.000 161.000 177.000 193.000 209.000 225.000 241.000
130.000 146.000 162.000 178.000 194.000 210.000 226.000 242.000
131.000 147.000 163.000 179.000 195.000 211.000 227.000 243.000
132.000 148.000 164.000 180.000 196.000 212.000 228.000 244.000
133.000 149.000 165.000 181.000 197.000 213.000 229.000 245.000
134.000 150.000 166.000 182.000 198.000 214.000 230.000 246.000
135.000 151.000 167.000 183.000 199.000 215.000 231.000 247.000
136.000 152.000 168.000 184.000 200.000 216.000 232.000 248.000
d_A3 Matrix
9.000   25.000  41.000  57.000  73.000  89.000  105.000 121.000
10.000  26.000  42.000  58.000  74.000  90.000  106.000 122.000
11.000  27.000  43.000  59.000  75.000  91.000  107.000 123.000
12.000  28.000  44.000  60.000  76.000  92.000  108.000 124.000
13.000  29.000  45.000  61.000  77.000  93.000  109.000 125.000
14.000  30.000  46.000  62.000  78.000  94.000  110.000 126.000
15.000  31.000  47.000  63.000  79.000  95.000  111.000 127.000
16.000  32.000  48.000  64.000  80.000  96.000  112.000 128.000
d_A4 Matrix
137.000 153.000 169.000 185.000 201.000 217.000 233.000 249.000
138.000 154.000 170.000 186.000 202.000 218.000 234.000 250.000
139.000 155.000 171.000 187.000 203.000 219.000 235.000 251.000
140.000 156.000 172.000 188.000 204.000 220.000 236.000 252.000
141.000 157.000 173.000 189.000 205.000 221.000 237.000 253.000
142.000 158.000 174.000 190.000 206.000 222.000 238.000 254.000
143.000 159.000 175.000 191.000 207.000 223.000 239.000 255.000
144.000 160.000 176.000 192.000 208.000 224.000 240.000 256.000
```

Figure B.5 Initiated 16x16 input matrix

```
d_L1 Matrix
1.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
2.000   1.000   0.000   0.000   0.000   0.000   0.000   0.000
3.000   2.000   1.000   0.000   0.000   0.000   0.000   0.000
4.000   3.000   0.000   1.000   0.000   0.000   0.000   0.000
5.000   4.000   0.000   0.000   1.000   0.000   0.000   0.000
6.000   5.000   0.000   0.000   0.000   1.000   0.000   0.000
7.000   6.000   0.000   0.000   0.000   0.000   1.000   0.000
8.000   7.000   0.000   0.000   0.000   0.000   0.000   1.000
d_U1 Matrix
1.000   17.000  33.000  49.000  65.000  81.000  97.000  113.000
0.000   -16.000 -32.000 -48.000 -64.000 -80.000 -96.000 -112.000
0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
d_L3 Matrix
9.000   8.000   0.000   0.000   0.000   0.000   0.000   0.000
10.000  9.000   0.000   0.000   0.000   0.000   0.000   0.000
11.000  10.000  0.000   0.000   0.000   0.000   0.000   0.000
12.000  11.000  0.000   0.000   0.000   0.000   0.000   0.000
13.000  12.000  0.000   0.000   0.000   0.000   0.000   0.000
14.000  13.000  0.000   0.000   0.000   0.000   0.000   0.000
15.000  14.000  0.000   0.000   0.000   0.000   0.000   0.000
16.000  15.000  0.000   0.000   0.000   0.000   0.000   0.000
d_U2 Matrix
129.000 145.000 161.000 177.000 193.000 209.000 225.000 241.000
-128.000        -144.000        -160.000        -176.000        -192.000        -208.000        -224.000        -240.000
0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
d_L4 Matrix
1.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
0.000   1.000   0.000   0.000   0.000   0.000   0.000   0.000
0.000   0.000   1.000   0.000   0.000   0.000   0.000   0.000
0.000   0.000   0.000   1.000   0.000   0.000   0.000   0.000
0.000   0.000   0.000   0.000   1.000   0.000   0.000   0.000
0.000   0.000   0.000   0.000   0.000   1.000   0.000   0.000
0.000   0.000   0.000   0.000   0.000   0.000   1.000   0.000
0.000   0.000   0.000   0.000   0.000   0.000   0.000   1.000
d_U4 Matrix
0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
```

Figure B.6 Output matrices for 16x16 LU factorization

Obtained the output matrices are multiplied separately and those results are shown in Figure B.7 below.

1.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
2.000	1.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
3.000	2.000	1.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
4.000	3.000	0.000	1.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
5.000	4.000	0.000	0.000	1.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
6.000	5.000	0.000	0.000	0.000	1.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
7.000	6.000	0.000	0.000	0.000	0.000	1.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
8.000	7.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
9.000	8.000	0.000	0.000	0.000	0.000	0.000	0.000	1.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
10.000	9.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	1.000	0.000	0.000	0.000	0.000	0.000	0.000
11.000	10.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	1.000	0.000	0.000	0.000	0.000	0.000
12.000	11.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	1.000	0.000	0.000	0.000	0.000
13.000	12.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	1.000	0.000	0.000	0.000
14.000	13.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	1.000	0.000	0.000
15.000	14.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	1.000	0.000
16.000	15.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	1.000

1.000	17.000	33.000	49.000	65.000	81.000	97.000	113.000	129.000	145.000	161.000	177.000	193.000	209.000	225.000	241.000
0.000	-16.000	-32.000	-48.000	-64.000	-80.000	-96.000	-112.000	-128.000	-144.000	-160.000	-176.000	-192.000	-208.000	-224.000	-240.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

Matrix product A*B

1.000	17.000	33.000	49.000	65.000	81.000	97.000	113.000	129.000	145.000	161.000	177.000	193.000	209.000	225.000	241.000
2.000	18.000	34.000	50.000	66.000	82.000	98.000	114.000	130.000	146.000	162.000	178.000	194.000	210.000	226.000	242.000
3.000	19.000	35.000	51.000	67.000	83.000	99.000	115.000	131.000	147.000	163.000	179.000	195.000	211.000	227.000	243.000
4.000	20.000	36.000	52.000	68.000	84.000	100.000	116.000	132.000	148.000	164.000	180.000	196.000	212.000	228.000	244.000
5.000	21.000	37.000	53.000	69.000	85.000	101.000	117.000	133.000	149.000	165.000	181.000	197.000	213.000	229.000	245.000
6.000	22.000	38.000	54.000	70.000	86.000	102.000	118.000	134.000	150.000	166.000	182.000	198.000	214.000	230.000	246.000
7.000	23.000	39.000	55.000	71.000	87.000	103.000	119.000	135.000	151.000	167.000	183.000	199.000	215.000	231.000	247.000
8.000	24.000	40.000	56.000	72.000	88.000	104.000	120.000	136.000	152.000	168.000	184.000	200.000	216.000	232.000	248.000
9.000	25.000	41.000	57.000	73.000	89.000	105.000	121.000	137.000	153.000	169.000	185.000	201.000	217.000	233.000	249.000
10.000	26.000	42.000	58.000	74.000	90.000	106.000	122.000	138.000	154.000	170.000	186.000	202.000	218.000	234.000	250.000
11.000	27.000	43.000	59.000	75.000	91.000	107.000	123.000	139.000	155.000	171.000	187.000	203.000	219.000	235.000	251.000
12.000	28.000	44.000	60.000	76.000	92.000	108.000	124.000	140.000	156.000	172.000	188.000	204.000	220.000	236.000	252.000
13.000	29.000	45.000	61.000	77.000	93.000	109.000	125.000	141.000	157.000	173.000	189.000	205.000	221.000	237.000	253.000
14.000	30.000	46.000	62.000	78.000	94.000	110.000	126.000	142.000	158.000	174.000	190.000	206.000	222.000	238.000	254.000
15.000	31.000	47.000	63.000	79.000	95.000	111.000	127.000	143.000	159.000	175.000	191.000	207.000	223.000	239.000	255.000
16.000	32.000	48.000	64.000	80.000	96.000	112.000	128.000	144.000	160.000	176.000	192.000	208.000	224.000	240.000	256.000

Figure B.7 Third party application Matrix Multiplication

GPU ONLY QR FACTORIZATION IMPLEMENTATION

<u>8x8 Matrix</u>

The input matrix and the output matrices for the 8x8 QR matrix factorization is shown below respectively in Figure B.8 and Figure B.9.

```
A Matrix 8 x 8 :
 0.000000  8.000000 16.000000 24.000000 32.000000 40.000000 48.000000 56.000000
 1.000000  9.000000 17.000000 25.000000 33.000000 41.000000 49.000000 57.000000
 2.000000 10.000000 18.000000 26.000000 34.000000 42.000000 50.000000 58.000000
 3.000000 11.000000 19.000000 27.000000 35.000000 43.000000 51.000000 59.000000
 4.000000 12.000000 20.000000 28.000000 36.000000 44.000000 52.000000 60.000000
 5.000000 13.000000 21.000000 29.000000 37.000000 45.000000 53.000000 61.000000
 6.000000 14.000000 22.000000 30.000000 38.000000 46.000000 54.000000 62.000000
 7.000000 15.000000 23.000000 31.000000 39.000000 47.000000 55.000000 63.000000
```

Figure B.8 Initiated 8x8 input matrix

```
R Matrix 8 x 8 :
-11.832160 -30.763613 -49.695068 -68.626526 -87.557976 -106.489433 -125.420883 -144.352341
 0.000000 -12.393543 -24.787088 -37.180634 -49.574188 -61.967728 -74.361259 -86.754814
 0.000000  0.000000  0.000002  0.000003  0.000001  0.000003  0.000004  0.000006
 0.000000  0.000000  0.000000  0.000004  0.000002  0.000000  0.000002  0.000006
 0.000000  0.000000  0.000000  0.000000  0.000008  0.000004  0.000005  0.000007
 0.000000  0.000000  0.000000  0.000000  0.000000  0.000005  0.000005  0.000003
 0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000006  0.000001
 0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000001

 Time to compute the 8x8 Q matrix = 0.514030 ms
QMatrix 8 x 8 :
 0.000000 -0.645497  0.404557  0.012503  0.127657  0.434305 -0.138676 -0.441999
-0.084515 -0.516398 -0.117971 -0.234604 -0.557072 -0.493145  0.305540 -0.101779
-0.169031 -0.387298 -0.630182  0.331342  0.538404 -0.149278  0.047803 -0.007991
-0.253546 -0.258199 -0.253834 -0.298289 -0.217178  0.459627 -0.283143  0.614042
-0.338062 -0.129099  0.454781  0.667695 -0.121725 -0.181704 -0.001872  0.410571
-0.422577  0.000000  0.341077 -0.505184  0.506184 -0.147546  0.372379  0.182280
-0.507093  0.129100 -0.004323 -0.136166 -0.021671 -0.333178 -0.708097 -0.307819
-0.591608  0.258199 -0.194105  0.162704 -0.254600  0.410919  0.406065 -0.347304
```

Figure B.9 Output matrices for 8x8 QR factorization

Obtained the output matrices are multiplied separately and those results are shown in Figure B.10 below.

Input matrix A:

0.000	-0.645	0.405	0.013	0.128	0.434	-0.139	-0.442
-0.085	-0.516	-0.118	-0.235	-0.557	-0.493	0.306	-0.102
-0.169	-0.387	-0.630	0.331	0.538	-0.149	0.048	-0.008
-0.254	-0.258	-0.254	-0.298	-0.217	0.460	-0.283	0.614
-0.338	-0.129	0.455	0.668	-0.122	-0.182	-0.002	0.411
-0.423	0.000	0.341	-0.505	0.506	-0.148	0.372	0.182
-0.507	0.129	-0.004	-0.136	-0.022	-0.333	-0.708	-0.308
-0.592	0.258	-0.194	0.163	-0.255	0.411	0.406	-0.347

Input matrix B:

-11.832	-30.764	-49.695	-68.627	-87.558	-106.489	-125.421	-144.352
0.000	-12.394	-24.787	-37.181	-49.574	-61.968	-74.361	-86.755
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

Matrix product A*B

0.000	8.000	16.000	24.000	32.000	40.000	48.000	56.000
1.000	9.000	17.000	25.000	33.000	41.000	49.000	57.000
2.000	10.000	18.000	26.000	34.000	42.000	50.000	58.000
3.000	11.000	19.000	27.000	35.000	43.000	51.000	59.000
4.000	12.000	20.000	28.000	36.000	44.000	52.000	60.000
5.000	13.000	21.000	29.000	37.000	45.000	53.000	61.000
6.000	14.000	22.000	30.000	38.000	46.000	54.000	62.000
7.000	15.000	23.000	31.000	39.000	47.000	55.000	63.000

Figure B.10 Third party application Matrix Multiplication

Compare the accuracy of the computation is derived using Equation 24 and depicted below in Figure B.11.

Input matrix A:

0.000	8.000	16.000	24.000	32.000	40.000	48.000	58.000
1.000	9.000	17.000	25.000	33.000	41.000	49.000	57.000
2.000	10.000	18.000	26.000	34.000	42.000	50.000	58.000
3.000	11.000	19.000	27.000	35.000	43.000	51.000	59.000
4.000	12.000	20.000	28.000	36.000	44.000	52.000	60.000
5.000	13.000	21.000	29.000	37.000	45.000	53.000	61.000
6.000	14.000	22.000	30.000	38.000	46.000	54.000	62.000
7.000	15.000	23.000	31.000	39.000	47.000	55.000	63.000

Input matrix B:

0.000	8.000	16.000	24.000	32.000	40.000	48.000	58.000
1.000	9.000	17.000	25.000	33.000	41.000	49.000	57.000
2.000	10.000	18.000	26.000	34.000	42.000	50.000	58.000
3.000	11.000	19.000	27.000	35.000	43.000	51.000	59.000
4.000	12.000	20.000	28.000	36.000	44.000	52.000	60.000
5.000	13.000	21.000	29.000	37.000	45.000	53.000	61.000
6.000	14.000	22.000	30.000	38.000	46.000	54.000	62.000
7.000	15.000	23.000	31.000	39.000	47.000	55.000	63.000

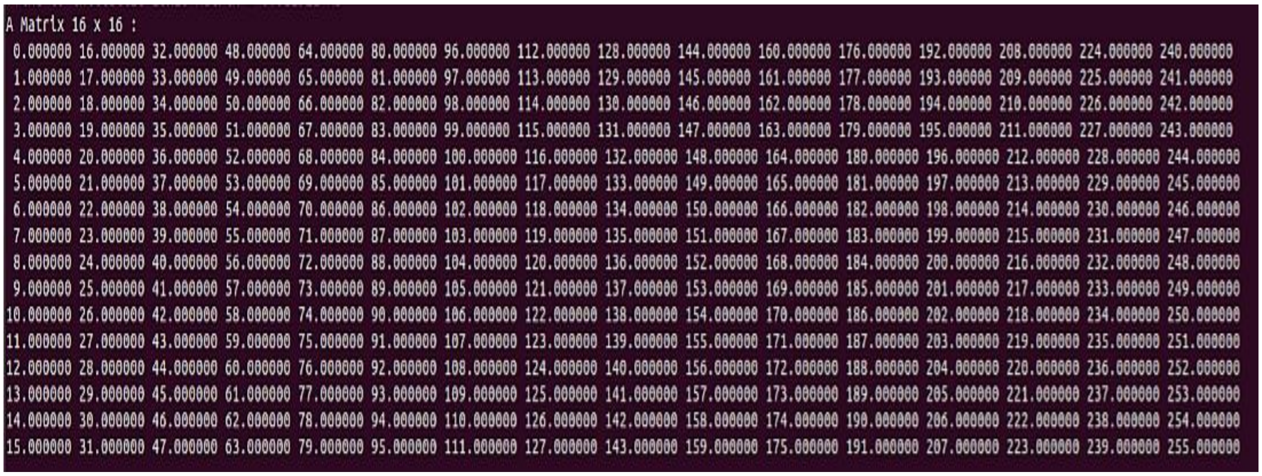
Matrix difference A + B

0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

Figure B.11 Compare the accuracy of 8x8 matrix

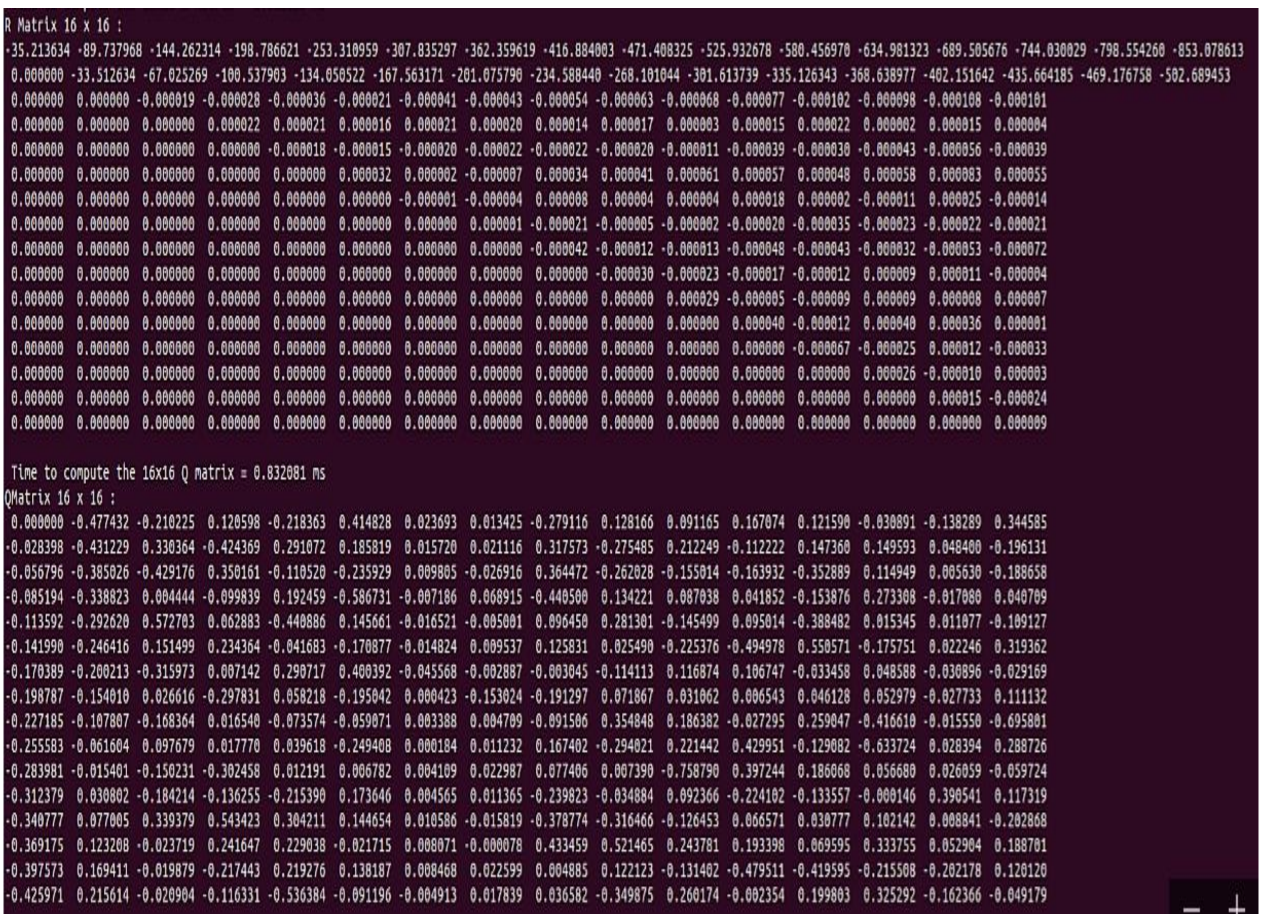
16x16 Matrix

The input matrix and the output matrices for the 16x16 QR matrix factorization is shown below respectively in Figure B.12 and Figure B.13.



```
A Matrix 16 x 16 :
 0.000000 16.000000 32.000000 48.000000 64.000000 80.000000 96.000000 112.000000 128.000000 144.000000 160.000000 176.000000 192.000000 208.000000 224.000000 240.000000
 1.000000 17.000000 33.000000 49.000000 65.000000 81.000000 97.000000 113.000000 129.000000 145.000000 161.000000 177.000000 193.000000 209.000000 225.000000 241.000000
 2.000000 18.000000 34.000000 50.000000 66.000000 82.000000 98.000000 114.000000 130.000000 146.000000 162.000000 178.000000 194.000000 210.000000 226.000000 242.000000
 3.000000 19.000000 35.000000 51.000000 67.000000 83.000000 99.000000 115.000000 131.000000 147.000000 163.000000 179.000000 195.000000 211.000000 227.000000 243.000000
 4.000000 20.000000 36.000000 52.000000 68.000000 84.000000 100.000000 116.000000 132.000000 148.000000 164.000000 180.000000 196.000000 212.000000 228.000000 244.000000
 5.000000 21.000000 37.000000 53.000000 69.000000 85.000000 101.000000 117.000000 133.000000 149.000000 165.000000 181.000000 197.000000 213.000000 229.000000 245.000000
 6.000000 22.000000 38.000000 54.000000 70.000000 86.000000 102.000000 118.000000 134.000000 150.000000 166.000000 182.000000 198.000000 214.000000 230.000000 246.000000
 7.000000 23.000000 39.000000 55.000000 71.000000 87.000000 103.000000 119.000000 135.000000 151.000000 167.000000 183.000000 199.000000 215.000000 231.000000 247.000000
 8.000000 24.000000 40.000000 56.000000 72.000000 88.000000 104.000000 120.000000 136.000000 152.000000 168.000000 184.000000 200.000000 216.000000 232.000000 248.000000
 9.000000 25.000000 41.000000 57.000000 73.000000 89.000000 105.000000 121.000000 137.000000 153.000000 169.000000 185.000000 201.000000 217.000000 233.000000 249.000000
10.000000 26.000000 42.000000 58.000000 74.000000 90.000000 106.000000 122.000000 138.000000 154.000000 170.000000 186.000000 202.000000 218.000000 234.000000 250.000000
11.000000 27.000000 43.000000 59.000000 75.000000 91.000000 107.000000 123.000000 139.000000 155.000000 171.000000 187.000000 203.000000 219.000000 235.000000 251.000000
12.000000 28.000000 44.000000 60.000000 76.000000 92.000000 108.000000 124.000000 140.000000 156.000000 172.000000 188.000000 204.000000 220.000000 236.000000 252.000000
13.000000 29.000000 45.000000 61.000000 77.000000 93.000000 109.000000 125.000000 141.000000 157.000000 173.000000 189.000000 205.000000 221.000000 237.000000 253.000000
14.000000 30.000000 46.000000 62.000000 78.000000 94.000000 110.000000 126.000000 142.000000 158.000000 174.000000 190.000000 206.000000 222.000000 238.000000 254.000000
15.000000 31.000000 47.000000 63.000000 79.000000 95.000000 111.000000 127.000000 143.000000 159.000000 175.000000 191.000000 207.000000 223.000000 239.000000 255.000000
```

Figure B.12 Initiated 16x16 input matrix



```
R Matrix 16 x 16 :
-35.213634 -89.737968 -144.262314 -198.786621 -253.310959 -307.835297 -362.359619 -416.884003 -471.408325 -525.932678 -580.456970 -634.981323 -689.505676 -744.030029 -798.554260 -853.078613
 0.000000 -33.512634 -67.025269 -100.537903 -134.050522 -167.563171 -201.075790 -234.588440 -268.101044 -301.613739 -335.126343 -368.638977 -402.151642 -435.664185 -469.176758 -502.689453
 0.000000  0.000000 -0.000019 -0.000028 -0.000036 -0.000021 -0.000041 -0.000043 -0.000054 -0.000063 -0.000068 -0.000077 -0.000102 -0.000098 -0.000108 -0.000101
 0.000000  0.000000  0.000000  0.000022  0.000021  0.000016  0.000021  0.000020  0.000014  0.000017  0.000003  0.000015  0.000022  0.000002  0.000015  0.000004
 0.000000  0.000000  0.000000  0.000000 -0.000018 -0.000015 -0.000020 -0.000022 -0.000022 -0.000020 -0.000011 -0.000039 -0.000030 -0.000043 -0.000056 -0.000039
 0.000000  0.000000  0.000000  0.000000  0.000000  0.000032  0.000002 -0.000007  0.000034  0.000041  0.000061  0.000057  0.000048  0.000058  0.000083  0.000055
 0.000000  0.000000  0.000000  0.000000  0.000000  0.000000 -0.000001 -0.000004  0.000008  0.000004  0.000004  0.000018  0.000002 -0.000011  0.000025 -0.000014
 0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000001 -0.000021 -0.000005 -0.000002 -0.000020 -0.000035 -0.000023 -0.000022 -0.000021
 0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000 -0.000042 -0.000012 -0.000013 -0.000048 -0.000043 -0.000032 -0.000053 -0.000072
 0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000 -0.000030 -0.000023 -0.000017 -0.000012  0.000009  0.000011 -0.000004
 0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000029 -0.000005 -0.000009  0.000009  0.000008  0.000007
 0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000040 -0.000012  0.000040  0.000036  0.000001
 0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000 -0.000067 -0.000025  0.000012 -0.000033
 0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000026 -0.000010  0.000003
 0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000015 -0.000024
 0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000009

 Time to compute the 16x16 Q matrix = 0.832081 ms
QMatrix 16 x 16 :
 0.000000 -0.477432 -0.210225  0.120598 -0.218363  0.414828  0.023693  0.013425 -0.279116  0.128166  0.091165  0.167074  0.121590 -0.030891 -0.138289  0.344585
-0.028398 -0.431229  0.330364 -0.424369  0.291072  0.185819  0.015720  0.021116  0.317573 -0.275485  0.212249 -0.112222  0.147360  0.149593  0.048400 -0.196131
-0.056796 -0.385026 -0.429176  0.350161 -0.110520 -0.235929  0.009805 -0.026916  0.364472 -0.262028 -0.155014 -0.163932 -0.352889  0.114949  0.005630 -0.188658
-0.085194 -0.338823  0.004444 -0.099839  0.192459 -0.586731 -0.007186  0.068915 -0.440500  0.134221  0.087038  0.041852 -0.153876  0.273308 -0.017080  0.040709
-0.113592 -0.292620  0.572703  0.062883 -0.440886  0.145661 -0.016521 -0.005001  0.096450  0.281301 -0.145499  0.095014 -0.388482  0.015345  0.011077 -0.109127
-0.141990 -0.246416  0.151499  0.234364 -0.041683 -0.170877 -0.014824  0.009537  0.125831  0.025490 -0.225376 -0.494978  0.550571 -0.175751  0.022246  0.319362
-0.170389 -0.200213 -0.315973  0.007142  0.290717  0.400392 -0.045568 -0.002887 -0.003045 -0.114113  0.116874  0.106747 -0.033458  0.048588 -0.030896 -0.029169
-0.198787 -0.154010  0.026616 -0.297831  0.058218 -0.195042  0.000423 -0.153024 -0.191297  0.071867  0.031062  0.006543  0.046128  0.052979 -0.027733  0.111132
-0.227185 -0.107807 -0.168364  0.016540 -0.073574 -0.059071  0.003388  0.004709 -0.091506  0.354848  0.186382 -0.027295  0.259047 -0.416610 -0.015550 -0.695801
-0.255583 -0.061604  0.097679  0.017770  0.039618 -0.249408  0.000184  0.011232  0.167402 -0.294021  0.221442  0.429951 -0.129082 -0.633724  0.028394  0.288726
-0.283981 -0.015401 -0.150231 -0.302458  0.012191  0.006782  0.004109  0.022987  0.077406  0.007390 -0.758790  0.397244  0.186068  0.056680  0.026059 -0.059724
-0.312379  0.030802 -0.184214 -0.136255 -0.215390  0.173646  0.004565  0.011365 -0.239823 -0.034884  0.092366 -0.224102 -0.133557 -0.000146  0.390541  0.117319
-0.340777  0.077005  0.339379  0.543423  0.304211  0.144654  0.010586 -0.015819 -0.378774 -0.316466 -0.126453  0.066571  0.030777  0.102142  0.008841 -0.202868
-0.369175  0.123208 -0.023719  0.241647  0.229038 -0.021715  0.008071 -0.000078  0.433459  0.521465  0.243781  0.193398  0.069595  0.333755  0.052904  0.188701
-0.397573  0.169411 -0.019879 -0.217443  0.219276  0.138187  0.008468  0.022599  0.004885  0.122123 -0.131402 -0.479511 -0.419595 -0.215508 -0.202178  0.120120
-0.425971  0.215614 -0.020904 -0.116331 -0.536384 -0.091196 -0.004913  0.017839  0.036582 -0.349875  0.260174 -0.002354  0.199803  0.325292 -0.162366 -0.049179
```

Figure B.13 Output matrices for 16x16 QR factorization

Obtained the output matrices are multiplied separately and those results are shown in Figure B.14 below.

Input matrix A:

0.000	-0.477	-0.210	0.121	-0.218	0.415	0.024	0.013	-0.279	0.128	0.091	0.167	0.122	-0.031	-0.138	0.345
-0.028	-0.431	0.330	-0.424	0.291	0.186	0.016	0.021	0.318	-0.275	0.212	-0.112	0.147	0.150	0.048	-0.196
-0.057	-0.385	-0.429	0.350	-0.111	-0.236	0.010	-0.027	0.364	-0.262	-0.155	-0.164	-0.353	0.115	0.006	-0.189
-0.085	-0.339	0.004	-0.100	0.192	-0.587	-0.007	0.069	-0.441	0.134	0.087	0.042	-0.154	0.273	-0.017	0.041
-0.114	-0.293	0.573	0.063	-0.441	0.146	-0.017	-0.005	0.096	0.281	-0.145	0.095	-0.388	0.015	0.011	-0.109
-0.142	-0.246	0.151	0.234	-0.042	-0.171	-0.015	0.010	0.126	0.025	-0.225	-0.495	0.551	-0.176	0.022	0.319
-0.170	-0.200	-0.316	0.007	0.291	0.400	-0.046	-0.003	-0.003	-0.114	0.117	0.107	-0.033	0.049	-0.031	-0.029
-0.199	-0.154	0.027	-0.298	0.058	-0.195	0.000	-0.153	-0.191	0.072	0.031	0.007	0.046	0.053	-0.028	0.111
-0.227	-0.108	-0.168	0.017	-0.074	-0.059	0.003	0.005	-0.092	0.355	0.186	-0.027	0.259	-0.417	-0.016	-0.696
-0.256	-0.062	0.098	0.018	0.040	-0.249	0.000	0.011	0.167	-0.294	0.221	0.430	-0.129	-0.634	0.028	0.289
-0.284	-0.015	-0.150	-0.302	0.012	0.007	0.004	0.023	0.077	0.007	-0.759	0.397	0.186	0.057	0.026	-0.060
-0.312	0.031	-0.184	-0.136	-0.215	0.174	0.005	0.011	-0.240	-0.035	0.092	-0.224	-0.134	0.000	0.391	0.117
-0.341	0.077	0.339	0.543	0.304	0.145	0.011	-0.016	-0.379	-0.316	-0.126	0.067	0.031	0.102	0.009	-0.203
-0.369	0.123	-0.024	0.242	0.229	-0.022	0.008	0.000	0.433	0.521	0.244	0.193	0.070	0.334	0.053	0.189
-0.398	0.169	-0.020	-0.217	0.219	0.138	0.008	0.023	0.005	0.122	-0.131	-0.480	-0.420	-0.216	-0.202	0.120
-0.426	0.216	-0.021	-0.116	-0.536	-0.091	-0.005	0.018	0.037	-0.350	0.260	-0.002	0.200	0.325	-0.162	-0.049

-35.214	-89.738	-144.262	-198.787	-253.311	-307.835	-362.360	-416.884	-471.408	-525.933	-580.457	-634.981	-689.506	-744.030	-798.554	-853.079
0.000	-33.513	-67.025	-100.538	-134.051	-167.563	-201.076	-234.588	-268.101	-301.614	-335.126	-368.639	-402.152	-435.664	-469.177	-502.689
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

0.000	16.000	32.000	48.000	64.000	80.000	96.000	112.000	128.000	144.000	160.000	176.000	192.000	208.000	224.000	240.000
1.000	17.000	33.000	49.000	65.000	81.000	97.000	113.000	129.000	145.000	161.000	177.000	193.000	209.000	225.000	241.000
2.000	18.000	34.000	50.000	66.000	82.000	98.000	114.000	130.000	146.000	162.000	178.000	194.000	210.000	226.000	242.000
3.000	19.000	35.000	51.000	67.000	83.000	99.000	115.000	131.000	147.000	163.000	179.000	195.000	211.000	227.000	243.000
4.000	20.000	36.000	52.000	68.000	84.000	100.000	116.000	132.000	148.000	164.000	180.000	196.000	212.000	228.000	244.000
5.000	21.000	37.000	53.000	69.000	85.000	101.000	117.000	133.000	149.000	165.000	180.999	196.999	212.999	228.999	244.999
6.000	22.000	38.000	54.000	70.000	86.000	102.000	118.000	134.000	150.000	166.000	182.000	198.000	214.000	230.000	246.000
7.000	23.000	39.000	55.000	71.000	87.000	103.000	119.000	135.000	151.000	167.000	183.000	199.000	215.000	231.000	247.000
8.000	24.000	40.000	56.000	72.000	88.000	104.000	120.000	136.000	152.000	168.000	184.000	200.000	216.000	232.000	248.000
9.000	25.000	41.000	57.000	73.000	89.000	105.000	121.000	137.000	153.000	169.000	185.000	201.000	217.000	233.000	249.000
10.000	26.000	42.000	58.000	74.000	90.000	106.000	122.000	138.000	154.000	170.000	186.000	202.000	218.000	234.000	250.000
11.000	27.000	43.000	59.000	75.000	91.000	107.000	123.000	139.000	155.000	171.000	187.000	203.000	219.000	235.000	251.000
12.000	28.000	44.000	60.000	76.000	92.000	108.000	124.000	140.000	156.000	172.000	188.000	204.000	220.000	236.000	252.000
13.000	29.000	45.000	61.000	77.000	93.000	109.000	125.000	141.000	157.000	173.000	189.000	205.000	221.000	237.000	253.000
14.000	30.000	46.000	62.000	78.000	94.000	110.000	126.000	142.000	158.000	174.000	190.000	206.000	222.000	238.000	254.000
15.000	31.000	47.000	63.000	79.000	95.000	111.000	127.000	143.000	159.000	175.000	191.000	207.000	223.000	239.000	255.000

Figure B.14 Third party application Matrix Multiplication