PERFORMANCE COMPARISON AND ANALYSIS

OF GAUSS JORDAN ALGORITHM FOR MATRIX

INVERSION ON GPU AND CPU

A

Project Report Submitted to Department of Computer Science

Lamar University

In Partial Fulfillment

of the Requirement for the Degree

Master of Science in Computer Science

By:

Guru Prasad Acharya

Pramit Sinha

Project Advisor: Dr. Lawrence J. Osborne

May 2014

Performance Comparison and Analysis of Gauss Jordan Algorithm for Matrix Inversion on GPU and CPU

A project report submitted in partial fulfillment of the requirements for Master of Science in Computer Science, has been approved and accepted by:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Dr. Lawrence J. Osborne

Project Advisor

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Dr. Stefan Andrei

Committee Member

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Dr. Jiangjiang Liu

Committee Member

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Dr. Stefan Andrei

Chair, Department of Computer Science,

Lamar University – Beaumont, Texas.

Abstract

In recent years, a lot of work has been explored in the field of GPGPU (general purpose computing on Graphics Processing Units). A number of applications have been tweaked to perform at dramatic speeds using parallelizing features of the CUDA architecture. Computing inversion of large matrices, swiftly and precisely, is popular among many applications, such as solving system of linear equations and real time simulations. In this project, we have implemented the Gauss Jordan algorithm for matrix inversion on the CUDA platform. We have exploited the parallelizing capabilities of GPUs and have compared the GPU general purpose implementation with that of CPU implementation.

Acknowledgement

It is our great pleasure to present this report on “Performance Comparison and Analysis of Gauss Jordan Algorithm for Matrix Inversion on GPU and CPU”. We would like to express our deepest gratitude to Dr. Lawrence J. Osborne, our project supervisor, for his valuable guidance, continuous support and encouragement. His availability and support were invaluable.

We would like to thank Dr. Stefan Andrei, Chair of Department of Computer Science and graduate committee member, for his valuable suggestions and encouragement to finish this project.

We are grateful to Dr. Jiangjiang Liu, graduate committee member, for her support and valuable suggestions.

We would also like to thank all the staff of the Department of Computer Science for allowing us to use the labs and resources.

Table of Contents

[List of Tables v](#_Toc387448991)

[List of Figures vi](#_Toc387448992)

[1. Introduction and Motivation 1](#_Toc387448993)

[1.1. Introduction 1](#_Toc387448994)

[1.2. Introduction to CUDA 4](#_Toc387448995)

[2. Related work 5](#_Toc387448996)

[2.1. GPU computing 5](#_Toc387448997)

[2.2. Matrix Inversion 6](#_Toc387448998)

[3. The CUDA architecture 9](#_Toc387448999)

[3.1. Heterogeneous architecture of CUDA 9](#_Toc387449000)

[3.2. Kernels 10](#_Toc387449001)

[3.3. Grids and Blocks 10](#_Toc387449002)

[3.4. The GPU Memory 12](#_Toc387449003)

[3.5. Synchronization of threads 13](#_Toc387449004)

[3.6. Flow of control 14](#_Toc387449005)

[4. Implementation 17](#_Toc387449006)

[4.1. The Gauss Jordan method for matrix inversion 17](#_Toc387449007)

[4.2. Implementation of the parallel Gauss Jordan algorithm 19](#_Toc387449008)

[5. Analysis 25](#_Toc387449009)

[5.1. Preparation 25](#_Toc387449010)

[5.2. Testing and Results 26](#_Toc387449011)

[5.2.1. Random Matrix 27](#_Toc387449012)

[5.2.2. Identity Matrix 33](#_Toc387449013)

[5.2.3. Band Matrix 37](#_Toc387449014)

[5.2.4. Hollow Matrix 42](#_Toc387449015)

[5.2.5. Sparse Matrix 47](#_Toc387449016)

[6. Conclusion and Future Work 52](#_Toc387449017)

[7. References 54](#_Toc387449018)

[APPENDIX A 57](#_Toc387449019)

# List of Tables

[Table 1: Different types of matrices used for benchmarking the algorithm 27](#_Toc387449020)

[Table 2: Computation Time of Random Matrix GPU vs CPU 27](#_Toc387449021)

[Table 3: Computation Time of Identity Matrix GPU vs CPU 33](#_Toc387449022)

[Table 4: Computation Time of Band Matrix GPU vs CPU 38](#_Toc387449023)

[Table 5: Computation Time of Hollow Matrix GPU vs CPU 42](#_Toc387449024)

[Table 6: Computation Time of Sparse Matrix GPU vs CPU 47](#_Toc387449025)

[Table 7: Computation Time for Random Matrices on GPU 57](#_Toc387449026)

[Table 8: Computation Time for Random Matrix on CPU 58](#_Toc387449027)

[Table 9: Computation Time for Identity Matrix on GPU 59](#_Toc387449028)

[Table 10: Computation Time for Identity Matrix on CPU 60](#_Toc387449029)

[Table 11: Computation Time for Band Matrix on GPU 61](#_Toc387449030)

[Table 12: Computation Time for Band Matrix on CPU 62](#_Toc387449031)

[Table 13: Computation Time for Hollow Matrix on GPU 63](#_Toc387449032)

[Table 14: Computation Time for Hollow Matrix on CPU 64](#_Toc387449033)

[Table 15: Computation Time for Sparse Matrix on GPU 65](#_Toc387449034)

[Table 16: Computation Time for Sparse Matrix on CPU 66](#_Toc387449035)

# List of Figures

[Figure 1: The Heterogeneous Architecture 13](#_Toc387449036)

[Figure 2: Grids, Blocks and Threads 15](#_Toc387449037)

[Figure 3: The GPU Memory System 16](#_Toc387449038)

[Figure 4: Flow of control (Inam 2009: 19) 19](#_Toc387449039)

[Figure 5: Matrix P (Matrix A augmented with the Identity Matrix I) 21](#_Toc387449040)

[Figure 6: Equation 2 applied to the first column 22](#_Toc387449041)

[Figure 7: Equation 3 applied to the first column 22](#_Toc387449042)

[Figure 8: The rowExchange kernel 25](#_Toc387449043)

[Figure 9: The fixRows kernel 26](#_Toc387449044)

[Figure 10: The fixColumns kernel 26](#_Toc387449045)

[Figure 11: Pseudo code for the Gauss Jordan algorithm for matrix inversion on GPU (Sharma, et.al. 2013) 27](#_Toc387449046)

[Figure 12: Inversion of random matrix of size 5 – 100 on GPU 33](#_Toc387449047)

[Figure 13: Inversion of random matrix of size 100 – 1000 on GPU 33](#_Toc387449048)

[Figure 14: Inversion of random matrix of size 1000 – 4000 on GPU 34](#_Toc387449049)

[Figure 15: Inversion of random matrix of size 5 – 100 on CPU 34](#_Toc387449050)

[Figure 16: Inversion of random matrix of size 100– 1000 on CPU 35](#_Toc387449051)

[Figure 17: Inversion of random matrix of size 1000 – 4000 on CPU 35](#_Toc387449052)

[Figure 18: Inversion of identity matrix of size 5 – 100 on GPU 38](#_Toc387449053)

[Figure 19: Inversion of identity matrix of size 100 – 1000 on GPU 39](#_Toc387449054)

[Figure 20: Inversion of identity matrix of size 1000 – 4000 on GPU 39](#_Toc387449055)

[Figure 21: Inversion of identity matrix of size 5 – 100 on CPU 40](#_Toc387449056)

[Figure 22: Inversion of identity matrix of size 100 – 1000 on CPU 40](#_Toc387449057)

[Figure 23: Inversion of identity matrix of size 1000 – 4000 on CPU 41](#_Toc387449058)

[Figure 24: Inversion of band matrix of size 5 – 100 on GPU 43](#_Toc387449059)

[Figure 25: Inversion of band matrix of size 100 – 1000 on GPU 44](#_Toc387449060)

[Figure 26: Inversion of band matrix of size 1000 – 4000 on GPU 44](#_Toc387449061)

[Figure 27: Inversion of band matrix of size 5 – 100 on CPU 45](#_Toc387449062)

[Figure 28: Inversion of band matrix of size 100 – 1000 on CPU 45](#_Toc387449063)

[Figure 29: Inversion of band matrix of size 1000 – 4000 on CPU 46](#_Toc387449064)

[Figure 30: Inversion of hollow matrix of size 5 – 100 on GPU 48](#_Toc387449065)

[Figure 31: Inversion of hollow matrix of size 100– 1000 on GPU 48](#_Toc387449066)

[Figure 32: Inversion of hollow matrix of size 1000 – 4000 on GPU 49](#_Toc387449067)

[Figure 33: Inversion of hollow matrix of size 5 – 100 on CPU 49](#_Toc387449068)

[Figure 34: Inversion of hollow matrix of size 100 – 1000 on CPU 50](#_Toc387449069)

[Figure 35: Inversion of hollow matrix of size 1000 – 4000 on CPU 50](#_Toc387449070)

[Figure 36: Inversion of sparse matrix of size 5 – 100 on GPU 53](#_Toc387449071)

[Figure 37: Inversion of sparse matrix of size 100 – 1000 on GPU 53](#_Toc387449072)

[Figure 38: Inversion of sparse matrix of size 1000– 4000 on GPU 54](#_Toc387449073)

[Figure 39: Inversion of sparse matrix of size 5 – 100 on CPU 54](#_Toc387449074)

[Figure 40: Inversion of sparse matrix of size 100 – 1000 on CPU 55](#_Toc387449075)

[Figure 41: Inversion of sparse matrix of size 1000 – 4000 on CPU 55](#_Toc387449076)

# Introduction and Motivation

## Introduction

For a long time, personal computers were built on single core processors and the most effective way to increase computational speed was to increase the clock speed. Limitations to the size of transistors, semiconductor scaling, fabrication of integrated circuits and heating issues were only a few of the several issues with the single core processor. This led to the idea of a multi–core architecture where two or more processors could be placed on a single die which introduced the approach of parallelizing programming techniques.

Single-core processors could execute one instruction at a time where as multi-core processors could execute multiple instructions at a time (one instruction per core). These multi–core processors administered not only better performance but better power and thermal control as each core ran at lower speeds at the expense of lesser dissipation of heat. A task could be broken down into smaller tasks and each of these tasks could be executed in parallel.

This background steered researchers to start thinking about GPU (Graphics Processing Unit) computing. GPUs overture massive parallelism as they dwell gobs of smaller cores, designed for parallel computing, and also offer high memory bandwidth and high computational throughput. The accelerated computing power makes GPUs much faster than CPUs and thus an attractive candidate worth to be investigated for general-purpose parallel computations.

Initially in the mid-1990s, GPUs were used for employing 3D graphics in PC games (Sanders and Kandrot2010, 4-5). In the year 2001, when NVIDIA released the GeForce 3 series, parallel computing received a breakthrough in GPU technology (Sanders and Kandrot2010, 4-5). Since then, GPGPU (General Purpose GPU) has been exploited for faster execution of several algorithms. Today, they are used in the fields of computational chemistry (Richie, Deer 2010), neural networks (Billiconan, Kavinguy 2008), database operations (Bakkum, Skadron 2010), digital signal processing (Dabrowski, et. al 2011), sorting and searching techniques (Satish, et. al 2009) and many more areas. However, a report from Hunan University, China, says that GPU computing is suitable for computations with simple branching logic only, whereas CPU is more suitable for processing complex logic computation (Zhang, et. al 2012).

In this project, we have exploited the parallelizing capabilities of GPUs and have tried comparing the GPU general purpose implementation with that of CPU implementation. Our goal in this project is to check the efficiency of the CUDA Toolkit so as to use it as an advantage to parallelize algorithms for their faster execution.

There are several methods for calculating the inverse of a matrix - such as Strassen’s approach (Strassen 1969), Romani’s approach (Romani 1982), Coppersmith’s and Winograd’s method [11], etc. The Gauss Jordan method for calculating the inverse of a matrix is one of the oldest methods. It has a time complexity of *O (n3)*, where *n* is the size of the matrix. It is easy to implement and can be tuned to run in a way so as to support massive parallelization. We have taken it as an example for calculating the inverse of different kinds of matrices – random, hollow, sparse, band, scalar and identity - of different sizes.

We implemented the Gauss Jordan algorithm using the CUDA platform described in (Sharma, et. al 2013). We also developed a CPU application of the same using C for testing the algorithm on a multi – core CPU. We then examine the computation time for inversing different types of matrices using first the GPU and then the CPU. We used the NVidia GeForce GTX 480 as the GPU and the Intel core i7 960 processor as the CPU system for our testing purpose. Through the obtained performance results we expose the possibilities the CUDA platform can achieve through parallel computing.

The use of multi-core technology in solving general purpose computations has been the domain of several researchers these days. We wanted to develop an algorithm that could exploit the various levels of parallelism offered by the CUDA architecture. We will illustrate the equity between each thread’s resource usage and number of simultaneously active threads. Encouraged by this motive we have implemented the Gauss Jordan algorithm for matrix inversion using GPU computing and have observed its efficiency and resource utilization. We have explored the optimizations needed to be done on the algorithm so as to enable it to produce quick and efficient results for small as well as large sized matrices.

The organization of the rest of this documentation is divided into 6 sections. The rest of this section introduces the CUDA platform. The second section deals with the related work done in the field of GPU computing and matrix inversion. The third section discusses the CUDA architecture. The fourth section describes our implementation of the Gauss Jordan algorithm using the CUDA architecture. The fifth section describes the detailed analysis and experimentation results of our work. Suggestions for future research and conclusions are presented in Section six.

## Introduction to CUDA

CUDA is a parallel computing platform and it stands for Compute Unified Device Architecture. It was invented by NVIDIA in 2007 and is used by developers for general purpose computing on graphics processing units (GPGPUs). GPUs have several cores in them unlike CPUs which allows them to execute many threads concurrently and hence effectively utilize parallelism. The NVIDIA GTX 480 has 448 CUDA cores clocked at 1215 MHz (NVIDIA, GTX 480. Accessed April 10, 2014) while the NVIDIA Tesla K20 accelerator has 2496 cores clocked at 706 MHz (NVIDIA, K20 GPU ACCELERATOR. Accessed April 10, 2014).

The CUDA platform can be utilized using C, C++ and FORTRAN. The benefit of using CUDA over traditional general purpose computing is that it allows a regular programming language like C to perform general purpose computations on the GPU instead of using vertex and pixel shaders.

CUDA implies the usage of functions called kernels which contain the code to be executed on the GPU device. The code that runs on the CPU is known as the host code. This host code invokes the kernel using the concept of tiles (N copies of the kernel code running in parallel using N threads). CUDA also requires proper memory allocation on the device before the kernel is invoked by the CPU.

NVIDIA GPUs that were built after the release of GeForce 8800 GTX are built on the CUDA architecture (Sanders and Kandrot2010, 4-5). More discussions on the CUDA architecture will be dealt in Section 3.

## 2. Related work

## 2.1. GPU computing

In the mid-1990s, PCs used display coprocessors for rendering 2D graphics and text on the display unit. The early 2000s demanded 3D graphics processing as PC based 3D games began to take over the market. At that time, display vendors such as NVIDIA, ATI Technologies and 3dfx Interactive began enumerating 3D processing capabilities in those graphics chips which allowed developers to start building 3D games (Sanders and Kandrot2010, 2-3).

Back then, OpenGL 1.5 and DirectX 7 based display chips allowed programmers to use only fixed graphics functions. The world’s first GPU – GeForce 256, was released on August 31, 1999 by NVIDIA. “It integrated transform, lighting, triangle setup/clipping, and rendering engines into a single chip” (NVIDIA, GeForce 256 1999). This created a breakthrough among programmers as they could now directly program the GPU using the OpenGL pipeline. Vertex and Pixel shaders were introduced after the release of DirectX 8 and OpenGL 2.0 specifications, which made 3D rendering more flexible and allowed developers to perform shadow and lightning computations (Thomas 2009). Programmers then began to perform computations on the GPU using pixel shaders.

In November 2006, NVIDIA released the GeForce 8800 GTX which was the first GPU ever to be built with NVIDIA’s CUDA architecture (Sanders and Kandrot 2010, 6-7). It was an extension of the C language and programmers could perform general-purpose computing without using any kind of graphics pipeline and 3D rendering APIs.

In recent years, a lot of work has been explored in the field of general purpose computing; known as “GPGPU”. A number of applications have been tweaked to perform at dramatic speeds using parallelizing features of the CUDA architecture. Kruger et al. has demonstrated a framework for solving linear algebra problems (Kruger, Westermann 2003). Billconan and Kavinguy utilized GPGPU to boost the performance of a Neural Network solving handwriting recognition problems (Billconan, Kavinguy 2008). Harris et al. presented a cloud dynamics simulation using partial differential equations (Harris, Baxter, Scheuermann, Lastra 2003). Database operations using pixel engines (Govindaraju, Lloyd, Wang, Lin, Manocha, 2004), efficient sorting and searching algorithms (Nadathur, Harris, Michael Garland 2009) and various other applications such as AES encryption (Yamanouchi 2007) have been implemented on GPUs using the CUDA architecture to improve their performance speed.

## 2.2. Matrix Inversion

Until 1969, the Gauss Jordan algorithm was considered to be the fastest algorithm for calculating the inverse of a matrix. Its complexity was calculated to be as *O (n3)*, *n* being the size of the matrix. Later that year, Strassen came up with an algorithm for matrix multiplication that reduced the time complexity to *O (n2.808)* time (Strassen 1969). The year 1978 proved another breakthrough when Pan’s discovery reduced the time complexity to *O (n2.796)* (Pan 1978). The next year, Bini et al. managed to obtain a time complexity of *O (n2.7799)* (Bini, Capovani, Romani, Lotti 1979). Two years later Schönhagere formulated Pan’s algorithm to prove a complexity of *O (n2.522)* time. In 1982, Romani’s discovery reduced the complexity to *O (n2.517)* (Romani, 1982). Coppersmith and Winograd were the first ones to get the complexity below 2.5. Their algorithm had a complexity of *O (2.496)*. Strassen introduced a new algorithm in 1988 (Strassen 1988) and achieved a complexity of *O (2.479)*. In 1990, Coppersmith and Winograd combined Strassen’s recent approach to create an algorithm that had a time complexity of *O (2.376)* which lasted as the fastest algorithm for a long span of 20 years (Coppersmith, Winograd 1990). In 2010, Andrew Stothers came up with a better approach to Coppersmith’s and Winograd’s algorithm and reduced the complexity to *O (n2.373)* (Davie, Stothers 2013). Virginia Williams broke the barrier in 2013 and reduced the complexity to *O (n2.3728642)*.

It is believed that it is possible to improve the complexity further but the exponential cannot be reduced to anything less than two. This is because when two matrices of size *n* are multiplied *(n\* n)* then every element in both the matrices has to be read at least once so as to perform the multiplication to obtain *n2* values.

In the recent past, a lot of developments have been made in the field of parallelization. Using GPGPU, we put forward an algorithm that has a time complexity less than a quadratic function. The extent to which this can be done depends upon the sphere to which parallelization can be achieved which depends upon the type of GPU used. For example, NVidia GeForce GTX 590 may have up to 49152 active threads, divided into sets of warps of 32 threads, but only 1024 threads can be dispatched in parallel (32 SM \* 48 Resident Warps per SM \* 32 Threads per warp) as only 32 SM \* 32 Threads per warp can be run in parallel. Any number above this will lead to queuing of warps waiting for their turn.

# 3. The CUDA architecture

## 3.1. Heterogeneous architecture of CUDA

The CUDA parallel programming model is an extension to C language and utilizes the coprocessor to run thousands of threads in parallel. It combines both serial and parallel executions based upon the heterogeneous style of programming. It involves building a host code that executes on the host memory which invokes functions called kernels that execute in parallel on the device memory. Figure 1 describes the heterogeneous architecture followed while using the CUDA platform.

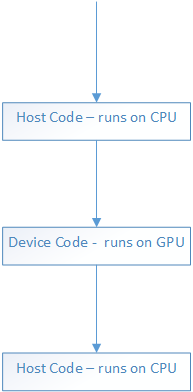


Figure 1: The Heterogeneous Architecture

## 3.2. Kernels

A kernel is a SPMD (Single Program Multiple Data) computation that is executed on the coprocessor by thousands of threads running concurrently. A \_\_global\_\_ qualifier is added to the kernel name which specifies that the kernel is to be executed on the device memory but not the host memory. A \_\_device\_\_ qualifier specifies that the kernel can be called only from the device memory but not the host memory.

## 3.3. Grids and Blocks

The grid can be defined as a one-dimensional, two-dimensional or three dimensional collection of parallel blocks. Each block is further divided into one dimensional or two-dimensional group of threads. Figure 2 shows the arrangement of grids, blocks and threads inside the GPU.

The grids and blocks are declared using the dim3 type. The maximum allowed size for each dimension of blocks is limited to 65,535 while the total number of threads in each block is limited to 1024 in recent GPUs (CUDA Toolkit Documentation, Table 12). The older ones were limited to 512 threads per block. Within a block, threads are organized in warps. Each warp comprises of 32 threads which are scheduled together for execution (CUDA Toolkit Documentation, Table 12).

The threads in a single block can communicate with each other through a shared memory. These threads are executed on the same multiprocessor and hence synchronization of these threads can be achieved easily.

Each block or thread can be identified through its identifier, denoted as id. The blockIdxvariable defines the id of a thread block while the blockDimvariable can be used to access its dimension. The gridDim variable identifies the dimensions of the grid. Each thread can be accessed using the threadIdx variable.

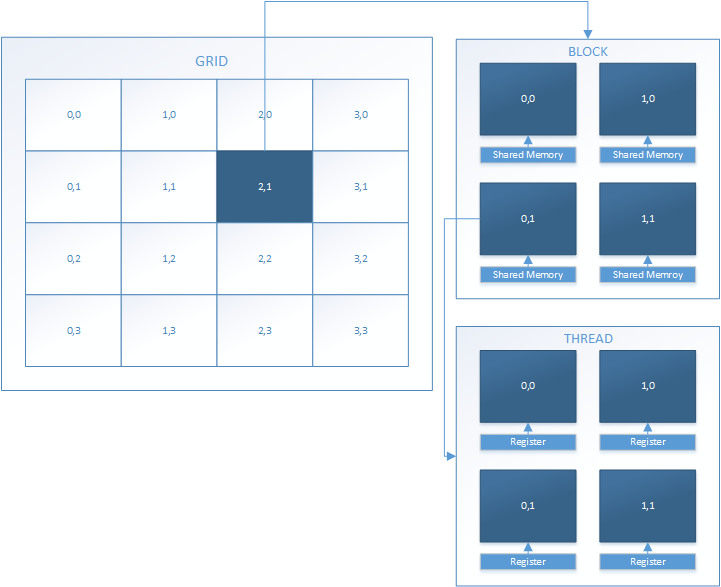


Figure 2: Grids, Blocks and Threads

The programmer controls the number of threads running on the device memory using grids and blocks. The creation, execution and termination of all threads are managed automatically by the GPU and these actions are invisible to the programmer.

## 3.4. The GPU Memory

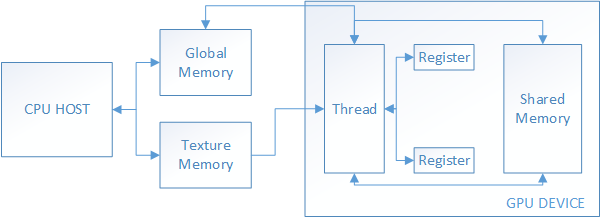


Figure 3: The GPU Memory System

Accessing the global device memory is slow as it does not provide caching. An alternative to this is to use shared memory (also known as PDC – Parallel Data Cache) which is faster than the device memory. The shared memory resides on the GPU and each multiprocessor has its own locally based shared memory which enables better and faster local synchronization of threads. The shared memory is divided into several subparts. Each subpart is assigned to a specific thread block in a multiprocessor and cannot be accessed by other thread blocks. The threads in a block utilize their part of the shared memory to perform all kinds of read and write operations. \_\_shared\_\_ qualifier is used to declare variables in the shared memory whereas \_\_device\_\_ qualifier is used to declare variables in the device memory (Sanders and Kandrot 2010, 75-76).

Apart from shared memories, each multiprocessor has its own read only caches (constant cache and texture cache memories) which are used to speed up read only operations. Each thread also has its own local memory where local variables of the kernel are stored.

## 3.5. Synchronization of threads

The synchronization of threads is important for avoiding race condition situations and nondeterministic results (Sanders and Kandrot 2010, 75-76). This can be achieved using a function called \_\_syncthreads()which acts as a synchronization point for all threads. This function is implemented in the kernel and causes threads to wait until all threads have finished executing up to this point. The intercommunication between threads is achieved through per-block shared memory (PBSM) that is only visible to threads within that thread block. This way of communication between threads is faster as PBSM is implemented using fast SRAM which is similar to first – level cache (Che, et al., 2008). Data can be shared between thread blocks using global memory, which is generally not cached, but the latency is of course much longer (Che, et al., 2008).

## 3.6. Flow of control

There are a few things to consider before we discuss the control flow of the CUDA programming paradigm. The CUDA compiler supports only C language but not object – oriented C++. A heterogeneous interaction is followed between the host code and the kernel code which involves copying data from the host memory to the device memory and then the computed results are copied back to the host memory from the device memory. The kernel functions do not return any results as their return type is always void. Recursions are not permitted in the kernel function as they may lead to memory management issues in the GPU because of thousands of threads executing concurrently. Within the device code it is not possible to allocate or de-allocate memory. All memory allocations must be done in the host code. We now discuss the basic flow of control followed while writing a program based on the CUDA platform.

Since the kernel function executes on the device memory, it is necessary to allocate memory in the device for the kernel to execute. This is achieved using the function cudaMalloc()in the host code*.* If required, the data to be executed can then be copied from the host memory to the device memory using a parameter, cudaMemcpyDeviceToHost in the function, cudaMemcpy()*.* The kernel code is then called using angular brackets (<<< n, n >>>) and arguments which we plan to pass to the runtime system. The content within the angular brackets specify the number of grids and blocks of threads that are going to be used for executing the device code in a parallel fashion. Once the device code finishes its execution, the computed result can then be copied from the device memory to the host memory using the parameter, cudaMemcpyHostToDevice in the function, cudaMemcpy(). The device memory should be freed in the end using cudaFree()(Sanders and Kandrot 2010, 42-43).

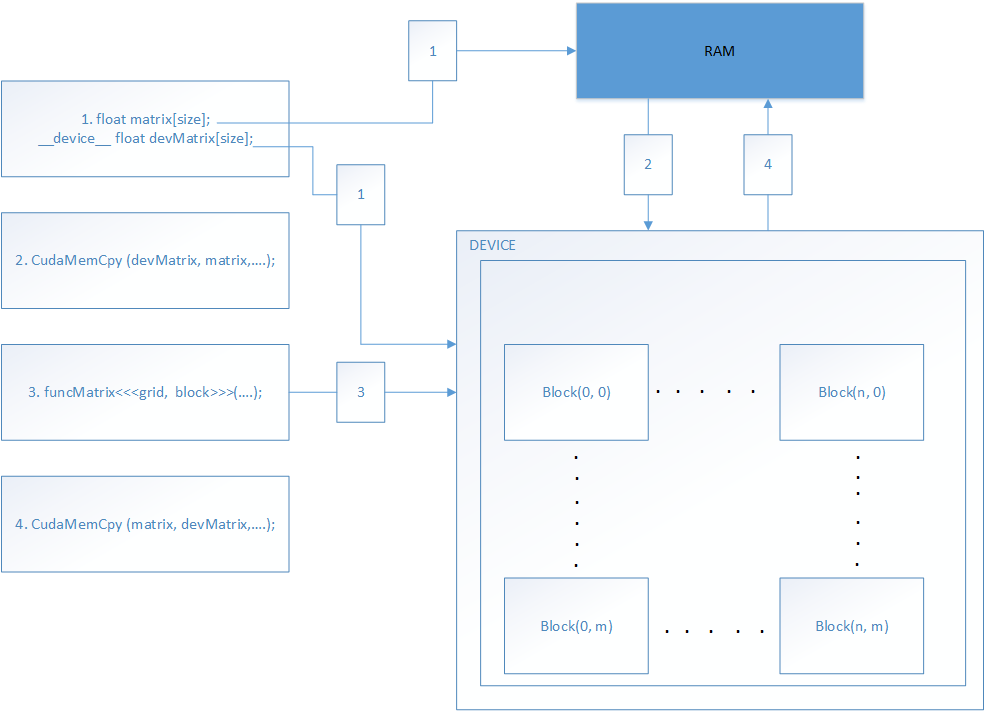


Figure 4: Flow of control (Inam 2009: 19)

Figure 4 describes the basic flow of control followed while writing a CUDA - based code. Step 1 shows two arrays of same size being declared in the host code one of which is allocated in the host memory and the other in the device memory. Step 2 involves copying the data to be executed on the device, from the host memory to the device memory, which is achieved using cudaMemCpy()*.* In Step 3 the kernel function is executed in parallel on the device. Step 4 shows the computed result being copied from the device memory to the host memory using the CUDA API cudaMemcpy()*.*

# Implementation

## The Gauss Jordan method for matrix inversion

The Gauss Jordan method for calculating the inverse of a matrix *A* can be implemented in the following way. The first step involves augmenting an identity matrix to the matrix *A* of size *n*. Figure 5 shows matrix *A* augmented with matrix *I*.

*[P] = [A | I]* (1)

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *A11* | *A12* | *…* | *…* | *A1n* | *1* | *0* | *...* | *…* | *0* |
| *A21* | *A22* | *…* | *…* | *A2n* | *0* | *1* | *…* | *…* | *0* |
| *A31* | *A32* | *…* | *…* |  | *0* | *0* | *…* | *…* | *0* |
| *…* | *…* | *…* | *…* | *…* | *…* | *…* | *…* | *…* | *…* |
| *An1* | *An2* | *…* | *…* | *Ann* | *0* | *0* | *…* | *…* | *1* |

Figure 5: Matrix P (Matrix A augmented with the Identity Matrix I)

The next step is to check if any row of the first half of the matrix *P* has *aii*as zero. If such is the case, then any other row with *aii* not equal to zero is added to it to make its *aii*non-zero.

The next step is to convert the diagonal elements *aii*into one. This can be done by performing the following row operation. Figure 6 shows the transformation of the matrix after applying equation 2 to the first column.

*Ri 🡨 Ri / aii*(2)

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *1* | *A12/A11* | *…* | *…* | *A1n/A11* | *1/A11* | *0* | *...* | *…* | *0* |
| *A21* | *A22* | *…* | *…* | *A2n* | *0* | *1* | *…* | *…* | *0* |
| *A31* | *A32* | *…* | *…* |  | *0* | *0* | *…* | *…* | *0* |
| *…* | *…* | *…* | *…* | *…* | *…* | *…* | *…* | *…* | *…* |
| *An1* | *An2* | *…* | *…* | *Ann* | *0* | *0* | *…* | *…* | *1* |

Figure 6: Equation 2 applied to the first column

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *1* | *A12/A11* | *…* | *…* | *A1n/A11* | *1/A11* | *0* | *...* | *…* | *0* |
| *0* | *A22-A21\*A12/A11* | *…* | *…* | *A2n-A21\*A1n/A11* | *0* | *1* | *…* | *…* | *0* |
| *0* | *A32-A31\*A12/A11* | *…* | *…* | *A3n-A31\*A1n/A11* | *0* | *0* | *…* | *…* | *0* |
| *…* | *…* | *…* | *…* | *…* | *…* | *…* | *…* | *…* | *…* |
| *0* | *An2-An1\*A12/A11* | *…* | *…* | *Ann-An1\*A1n/A11* | *0* | *0* | *…* | *…* | *1* |

Figure 7: Equation 3 applied to the first column

Next step is to convert all the other elements of the *jth*column to zero. This is achieved by performing the following operation. This is done for every row except for the *jth*row. Figure 7 shows the transformation of the matrix after applying equation 3 to the first column.

*Ri 🡨 Ri - Rj x aij* (3)

For each column, equations 2 and 3 are applied to equation 1 sequentially. This converts the left part of the matrix *P* to an identity matrix and the right part of *P* to the inverse of the matrix *A*.

*[P] = [I | A-1]* (4)

It can be observed that equation 2 in the above mentioned original Gauss Jordan algorithm involves processing *2n* elements and equation 3 involves processing *(n – 1)* rows of *2n* elements each; where *n* is the size of the original matrix A. If this algorithm is implemented on the CPU, where the processing is done sequentially, then the complexity of the program becomes equivalent to *n \* (n – 1)\* n*; which equals *O (n3)*.

## 4.2. Implementation of the parallel Gauss Jordan algorithm

We now focus on modernizing the algorithm so that we can capitalize the parallelizing capabilities of the GPU. We first augment the original matrix whose inverse is to be calculated ‘*A*’, with the identity matrix ‘*I*’, to create a new matrix ‘*P*’. We then perform equation 2 in parallel which involves processing the whole row of *P* at once. Each element in the row is divided by the diagonal element of that row. We then perform equation 3 in parallel in which we convert the remaining elements of the *jth* column to zero.

The number of blocks and threads required for performing a parallel computation is decided using the concept of grids, blocks and threads which are explained in much detail in section 3. We know that the number of threads in a block in modern GPUs is limited to 1024. Keeping this in mind, we have created three variables thread, rBlock and cBlockofdim3type. The thread variable decides the number of threads to be executed in each block, rBlock decides the number of blocks required for computing equation 2 and cBlock decides the number of blocks required for computing equation 3.

If the number of columns of the matrix *P* is less than or equal to 1024 then the thread variable will have a value equal to the number of columns. If the number of columns is greater than 1024 then the value of thread variable will be equal to 1024. The idea of implementing the thread variable is to assign a single thread to each element in a row while computing equations 2 and 3, keeping in mind, the maximum number of threads limited to each block (i.e. 1024).

The value of rBlock variable is calculated by dividing the total number of columns of the matrix *P* with the value of thread variable. This operation gives the total number of blocks to be used for computing equation 2.

For computing equation 3, we need to perform operations on *2n \* (n – 1)* elements; where *n* is the size of the original matrix *A*. In order to achieve this, the value of cBlock variable is obtained by dividing the total number of elements in the matrix *P* with the value of thread variable. cBlockgives the total number of blocks that are going to be required for computing equation 3.

We have built three different kernel functions – rowExchange(), fixRows() and fixColumns()for implementing the algorithm in parallel. We now discuss the functionality and present our code for each of these kernel functions.

If any diagonal element, *aii* of the original matrix *A* is zero, then we scan for another row, such that, *aji* is non-zero. The rowExchange()kernel function is then invoked which adds the row containing the *aji* element to the row containing the *aii* element. The rowExchange() kernel function can be implemented using Figure 8.

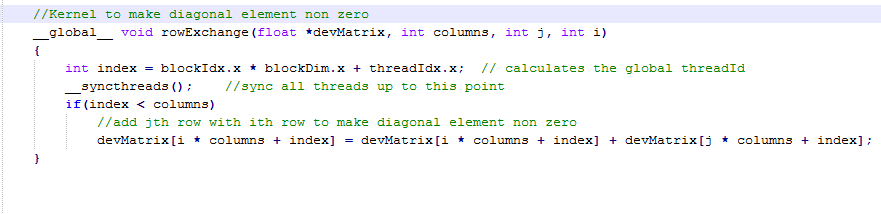


Figure 8: The rowExchange kernel

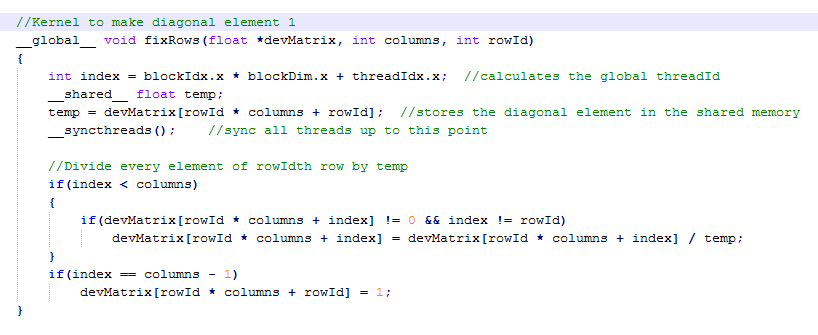
**

Figure 9: The fixRows kernel

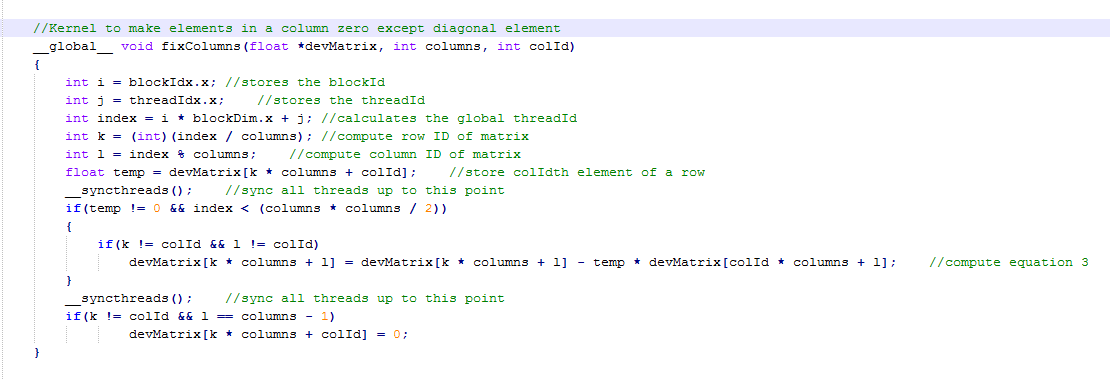


Figure 10: The fixColumns kernel

Next, the fixRows() kernel function is called. This performs the operation required for computing equation 2. Figure 9 explains the implementation of the fixRows() kernel.

The fixColumns() kernel function implements equation 3 in which every other element of the *ith* column is evaluated to zero. The implementation of it is shown in Figure 10.

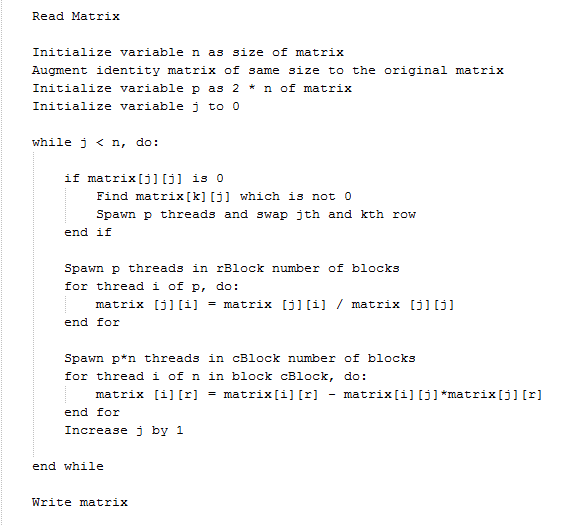


Figure 11: Pseudo code for the Gauss Jordan algorithm for matrix inversion on GPU (Sharma, et.al. 2013)

In order to reduce computations, we have implemented a window system in which we compute only *n + 1* elements for every *jth*row of *P* in equation 2. Following this way we do not have to perform computations on the remaining elements of that row (in the augmented identity matrix *I*) since they are already zero. Similarly, we avoid computations for the elements of the *jth* column in the matrix which are already zero, while implementing equation 3.

The usage of shared memory also speeds up the overall computation time. The temp variable of the fixRows() kernel is stored in the shared memory. Shared memory acts as a cache for the GPU and increases its read and write abilities. We have also implemented pivoting in all the three kernel functions to prevent any division by zero exceptions. Figure 11 describes the final pseudo code of our implementation.

We also implemented a sequential version of the Gauss Jordan algorithm to work on the CPU. We then compare the results of the performance of both versions of the algorithm against each other in the next section.

# 5. Analysis

## 5.1. Preparation

We used CUDA C to program a parallel version of the Gauss Jordan algorithm to be executed on the GPU and a sequential version to be executed on the CPU. We then tested both the codes against different sizes of matrices of different types – random matrix, identity matrix, band matrix, hollow matrix and sparse matrix.

We used the NVidia Tesla K20c as the GPU Accelerator which has a computation capability of 3.5 which means that its major revision number is 3 and its minor revision number is 5 (devices with major number 3 are based on the Kepler architecture) (Gupta, 2013). It has 13 multiprocessors with 192 single precision cores, 64 double-precision cores and 32 special function units per multiprocessor. This means that it has 2496 single-precision cores (192 \* 13), 832 double-precision cores (64 \* 13) and 416 special units (32 \* 13). It is capable of storing 26624 active threads (13 SM \* 64 Resident Warps per SM \* 32 Threads per Warp) (CUDA Toolkit Documentation, Table 12). It has a shared memory size of 48 KB. The total number of registers available per block is 65536. The maximum number of threads per block is limited to 1024 (CUDA Toolkit Documentation, Table 12).

For the CPU, we used the Intel Core i7-3770 Quad-Core Processor @ 3.4 GHz each. It is capable of running eight threads in parallel.

We have used single-precision floating point numbers as our input matrix. We developed another program that uses a seed value and takes the size of the matrix and the type of the matrix as inputs to generate random floating point matrices of the type specified. The output matrix could be finally verified by multiplying the original matrix to the output matrix to get an identity matrix. If the original matrix was not invertible then an entire row of zeroes would show up in the output matrix. This situation arises when the input matrix is linearly dependent (Cherlin, Lecture 5).

## 5.2. Testing and Results

We now present the results of our implementation of the Gauss Jordan algorithm. Table 1 shows the different types of matrices we used for benchmarking the algorithm. A random matrix is a matrix comprising of random numbers as its elements. Identity matrices are those in which the main diagonal elements are set to one while all the other elements are set to zero. A sparse matrix is a matrix in which most of the elements are populated with zeroes. A band matrix is a type of sparse matrix in which the non-zero elements are present on the main diagonal and on zero or more diagonals on the either sides of the main diagonal. A hollow matrix is a matrix that has all its diagonal elements as zeroes.

The experimental results for the implementation of the Gauss – Jordan algorithm on the GPU and CPU using the CUDA architecture are as follows. We generated matrices of different size for each type and performed five runs for every size on both the GPU and the CPU. We then noted the computation time taken for each run and then calculated their median which is the computation time shown in Table 2 - 6. We also calculated the speedup for every five runs by dividing the computation time taken on CPU by computation time taken on GPU.

Table 1: Different types of matrices used for benchmarking the algorithm

|  |  |
| --- | --- |
| No. | Types of matrices |
| 1 | Random Matrix |
| 2 | Identity Matrix |
| 3 | Band Matrix |
| 4 | Hollow Matrix |
| 5 | Sparse Matrix |

### 5.2.1. Random Matrix

Table 2: Computation Time of Random Matrix GPU vs CPU

|  |  |  |  |
| --- | --- | --- | --- |
| **:::Random matrix (GPU vs CPU computation time comparison):::** | | | |
|  | **Computation Time (ms)** |  |  |
| **Matrix size (n x n)** | **GPU** | **CPU** | **Speedup** |
| 5 | 0.12 | 0.01 | 0.11 |
| 10 | 0.22 | 0.02 | 0.07 |
| 15 | 0.28 | 0.07 | 0.25 |
| 20 | 0.35 | 0.15 | 0.43 |
| 25 | 0.33 | 0.55 | 1.67 |
| 30 | 0.47 | 0.77 | 1.62 |
| 35 | 0.56 | 1.03 | 1.82 |
| 40 | 0.61 | 1.39 | 2.30 |
| 45 | 0.67 | 1.02 | 1.51 |
| 50 | 0.75 | 2.44 | 3.26 |
| 55 | 0.81 | 3.08 | 3.81 |
| 60 | 0.87 | 2.01 | 2.32 |
| 65 | 0.95 | 4.82 | 5.07 |
| 70 | 1.02 | 2.67 | 2.61 |
| 75 | 1.08 | 6.56 | 6.10 |
| 80 | 1.17 | 8.75 | 7.47 |
| 85 | 1.23 | 9.33 | 7.61 |
| 90 | 1.32 | 9.22 | 6.98 |
| 95 | 1.39 | 10.46 | 7.53 |
| 100 | 1.46 | 14.07 | 9.62 |
| 200 | 4.44 | 59.52 | 13.41 |
| 300 | 10.63 | 186.66 | 17.56 |
| 400 | 24.02 | 433.84 | 18.06 |
| 500 | 44.22 | 838.42 | 18.96 |
| 600 | 68.09 | 1449.20 | 21.28 |
| 700 | 104.06 | 2296.33 | 22.07 |
| 800 | 150.62 | 3429.27 | 22.77 |
| 900 | 211.07 | 4886.46 | 23.15 |
| 1000 | 285.47 | 6714.02 | 23.52 |
| 1500 | 925.24 | 22758.22 | 24.60 |
| 2000 | 2152.72 | 53845.59 | 25.01 |
| 2500 | 4185.58 | 104868.97 | 25.05 |
| 3000 | 7161.36 | 180783.50 | 25.24 |
| 3500 | 11402.93 | 287660.09 | 25.23 |
| 4000 | 16888.23 | 428133.59 | 25.35 |

Figure 12: Inversion of random matrix of size 5 – 100 on GPU

Figure 13: Inversion of random matrix of size 100 – 1000 on GPU

Figure 14: Inversion of random matrix of size 1000 – 4000 on GPU

Figure 15: Inversion of random matrix of size 5 – 100 on CPU

Figure 16: Inversion of random matrix of size 100– 1000 on CPU

Figure 17: Inversion of random matrix of size 1000 – 4000 on CPU

For random matrices, we observe that the graph in Figure 12 is linear when the size of the matrix is around 100 and below. This is because the maximum number of parallel threads required to perform the matrix inversion for sizes 115 and below is going to be less than 26450; 230 columns \* 115 rows (230 columns, because the original matrix is augmented with an identity matrix of the same size in our algorithm). The maximum number of active threads in our GPU device is limited to 26624. Hence, all the 26450 threads were already loaded in the memory and scheduled to process the 26450 elements which caused no latency.

For matrices of size greater than 115, it can be observed in Figure 13 that the graph takes a quadratic curvature. If the size of the matrix is 116, then the minimum number of threads required to process all the elements will be 26912 (232 columns \* 116 rows). Since the number of active threads limited to our GPU device is 26624, the remaining 288 elements will require another cycle of warps. This causes latencies between dispatching warps when the remaining 288 threads are loaded in the memory. With this analysis, we can say that more the size of the matrix, the more warp cycles will be required and the greater will be the latency. This explains the quadratic nature of the graphs seen in Figure 13 and 14.

In the CPU implementation, we observe in Table 2 that the CPU code performs faster for matrices of size up to 20. Since the CUDA architecture follows the heterogeneous style of programming, the cudaMemcpy() function takes extra time to copy the matrix from the host memory to the device memory and then the results back to the host memory from the device memory. This explains the increase in latency of the GPU code for small sized matrices.

As the size of the matrix goes on increasing, the ratio of the computation time taken by the CPU code to the computation time required by the GPU code continues to increase. This happens because the elements of the matrix are processed sequentially one after the other. Unlike the GPU, which uses the CUDA architecture, there are no parallelization techniques to implement on the CPU.

### 5.2.2. Identity Matrix

Next, follows the results we obtained for identity matrices.

Table 3: Computation Time of Identity Matrix GPU vs CPU

|  |  |  |  |
| --- | --- | --- | --- |
| **:::Identity matrix (GPU vs CPU computation time comparison):::** | | | |
|  | **Computation Time (ms)** |  |  |
| **Matrix size (n x n)** | **GPU** | **CPU** | **Speedup** |
| 5 | 0.11 | 0.01 | 0.12 |
| 10 | 0.22 | 0.03 | 0.13 |
| 15 | 0.28 | 0.03 | 0.11 |
| 20 | 0.34 | 0.15 | 0.44 |
| 25 | 0.41 | 0.55 | 1.33 |
| 30 | 0.47 | 0.75 | 1.58 |
| 35 | 0.54 | 1.05 | 1.93 |
| 40 | 0.59 | 1.40 | 2.38 |
| 45 | 0.69 | 1.84 | 2.67 |
| 50 | 0.74 | 2.41 | 3.24 |
| 55 | 0.84 | 3.10 | 3.70 |
| 60 | 0.88 | 3.99 | 4.55 |
| 65 | 0.96 | 4.83 | 5.05 |
| 70 | 1.01 | 5.97 | 5.94 |
| 75 | 1.06 | 7.28 | 6.85 |
| 80 | 1.13 | 8.72 | 7.70 |
| 85 | 1.21 | 7.45 | 6.16 |
| 90 | 1.15 | 9.77 | 8.49 |
| 95 | 1.33 | 12.99 | 9.78 |
| 100 | 1.41 | 12.78 | 9.09 |
| 200 | 3.90 | 60.05 | 15.40 |
| 300 | 9.14 | 183.40 | 20.06 |
| 400 | 19.79 | 433.91 | 21.93 |
| 500 | 31.42 | 841.60 | 26.79 |
| 600 | 49.73 | 1449.19 | 29.14 |
| 700 | 75.25 | 2298.32 | 30.54 |
| 800 | 108.60 | 3425.28 | 31.54 |
| 900 | 151.19 | 4878.48 | 32.27 |
| 1000 | 203.91 | 6714.30 | 32.93 |
| 1500 | 659.96 | 22759.17 | 34.49 |
| 2000 | 1538.00 | 53881.54 | 35.03 |
| 2500 | 2980.04 | 104942.16 | 35.22 |
| 3000 | 5125.92 | 180889.91 | 35.29 |
| 3500 | 8113.58 | 287670.97 | 35.46 |
| 4000 | 12084.09 | 428087.38 | 35.43 |

Figure 18: Inversion of identity matrix of size 5 – 100 on GPU

Figure 19: Inversion of identity matrix of size 100 – 1000 on GPU

Figure 20: Inversion of identity matrix of size 1000 – 4000 on GPU

Figure 21: Inversion of identity matrix of size 5 – 100 on CPU

Figure 22: Inversion of identity matrix of size 100 – 1000 on CPU

Figure 23: Inversion of identity matrix of size 1000 – 4000 on CPU

When comparing the results of identity matrices with random matrices, we observed that the computation time taken in case of identity matrices is lesser than that of random matrices. In the fixColumns() kernel function of our GPU code, if any element in the *jth* column is zero, we simply discard it. Since all the elements except for the diagonal elements are zero, a lot of computation time is saved in the case of identity matrices.

### 5.2.3. Band Matrix

In band matrices, the elements in the main diagonal are non – zeroes. Also same numbers of diagonals above and below the main diagonal contain non-zeroes while the rest of the elements are zeroes. This explains the computation time of band matrices being slightly greater than identity matrices but much lesser than random matrices.

Table 4: Computation Time of Band Matrix GPU vs CPU

|  |  |  |  |
| --- | --- | --- | --- |
| **:::Band matrix (GPU vs CPU computation time comparison):::** | | | |
|  | **Computation Time (milliseconds)** |  |  |
| **Matrix size (n x n)** | **GPU** | **CPU** | **Speedup** |
| 5 | 0.12 | 0.01 | 0.10 |
| 10 | 0.22 | 0.03 | 0.13 |
| 15 | 0.29 | 0.07 | 0.24 |
| 20 | 0.34 | 0.14 | 0.42 |
| 25 | 0.41 | 0.55 | 1.35 |
| 30 | 0.49 | 0.75 | 1.54 |
| 35 | 0.55 | 0.59 | 1.09 |
| 40 | 0.61 | 1.39 | 2.29 |
| 45 | 0.71 | 1.81 | 2.54 |
| 50 | 0.74 | 2.36 | 3.19 |
| 55 | 0.81 | 3.04 | 3.78 |
| 60 | 0.91 | 3.92 | 4.31 |
| 65 | 0.82 | 4.83 | 5.93 |
| 70 | 1.03 | 4.71 | 4.57 |
| 75 | 1.11 | 7.27 | 6.57 |
| 80 | 1.18 | 5.13 | 4.35 |
| 85 | 1.22 | 9.26 | 7.62 |
| 90 | 1.32 | 12.31 | 9.34 |
| 95 | 1.35 | 10.23 | 7.58 |
| 100 | 1.42 | 11.65 | 8.19 |
| 200 | 4.21 | 53.75 | 12.76 |
| 300 | 10.00 | 249.33 | 24.94 |
| 400 | 21.50 | 602.57 | 28.03 |
| 500 | 34.90 | 1112.94 | 31.89 |
| 600 | 55.29 | 1828.53 | 33.07 |
| 700 | 84.62 | 2767.11 | 32.70 |
| 800 | 121.86 | 3977.41 | 32.64 |
| 900 | 168.90 | 5529.15 | 32.74 |
| 1000 | 226.61 | 7474.10 | 32.98 |
| 1500 | 713.49 | 23959.88 | 33.58 |
| 2000 | 1632.32 | 55420.25 | 33.95 |
| 2500 | 3129.68 | 106875.49 | 34.15 |
| 3000 | 5344.18 | 183176.48 | 34.28 |
| 3500 | 8416.71 | 290247.25 | 34.48 |
| 4000 | 12472.65 | 430832.03 | 34.54 |

Figure 24: Inversion of band matrix of size 5 – 100 on GPU

Figure 25: Inversion of band matrix of size 100 – 1000 on GPU

Figure 26: Inversion of band matrix of size 1000 – 4000 on GPU

Figure 27: Inversion of band matrix of size 5 – 100 on CPU

Figure 28: Inversion of band matrix of size 100 – 1000 on CPU

Figure 29: Inversion of band matrix of size 1000 – 4000 on CPU

### 5.2.4. Hollow Matrix

Hollow matrices are those which have their diagonal elements as zero and the non- diagonal elements as non-zero. Our algorithm calls the rowExchange()kernel function if it finds any zeroes in the main diagonal of the input matrix. This is why the inverse computation of hollow matrices takes more time than other matrices, considered in our case, which do not have diagonal elements as zero.

Table 5: Computation Time of Hollow Matrix GPU vs CPU

|  |  |  |  |
| --- | --- | --- | --- |
| **:::Band matrix (GPU vs CPU computation time comparison):::** | | | |
|  | **Computation Time (milliseconds)** |  |  |
| **Matrix size (n x n)** | **GPU** | **CPU** | **Speedup** |
| 5 | 0.19 | 0.01 | 0.04 |
| 10 | 0.28 | 0.03 | 0.10 |
| 15 | 0.38 | 0.07 | 0.18 |
| 20 | 0.48 | 0.15 | 0.30 |
| 25 | 0.46 | 0.54 | 1.18 |
| 30 | 0.68 | 0.74 | 1.09 |
| 35 | 0.78 | 1.03 | 1.32 |
| 40 | 0.86 | 1.39 | 1.62 |
| 45 | 0.95 | 1.79 | 1.89 |
| 50 | 1.09 | 1.15 | 1.05 |
| 55 | 1.17 | 3.04 | 2.61 |
| 60 | 1.29 | 3.98 | 3.09 |
| 65 | 1.14 | 4.79 | 4.20 |
| 70 | 1.44 | 3.70 | 2.58 |
| 75 | 1.56 | 4.40 | 2.82 |
| 80 | 1.64 | 6.48 | 3.96 |
| 85 | 1.72 | 9.03 | 5.25 |
| 90 | 1.91 | 12.26 | 6.41 |
| 95 | 1.97 | 10.11 | 5.13 |
| 100 | 1.89 | 11.72 | 6.21 |
| 200 | 5.39 | 58.48 | 10.85 |
| 300 | 12.10 | 187.15 | 15.47 |
| 400 | 26.32 | 430.86 | 16.37 |
| 500 | 47.15 | 834.85 | 17.71 |
| 600 | 70.44 | 1437.44 | 20.41 |
| 700 | 107.54 | 2289.63 | 21.29 |
| 800 | 154.60 | 3409.52 | 22.05 |
| 900 | 215.63 | 4859.06 | 22.53 |
| 1000 | 290.56 | 6691.30 | 23.03 |
| 1500 | 932.83 | 22686.44 | 24.32 |
| 2000 | 2161.72 | 53677.53 | 24.83 |
| 2500 | 4196.95 | 104648.30 | 24.93 |
| 3000 | 7176.73 | 180509.25 | 25.15 |
| 3500 | 11421.99 | 287067.25 | 25.13 |
| 4000 | 16907.00 | 427538.88 | 25.29 |

Figure 30: Inversion of hollow matrix of size 5 – 100 on GPU

Figure 31: Inversion of hollow matrix of size 100– 1000 on GPU

Figure 32: Inversion of hollow matrix of size 1000 – 4000 on GPU

Figure 33: Inversion of hollow matrix of size 5 – 100 on CPU

Figure 34: Inversion of hollow matrix of size 100 – 1000 on CPU

Figure 35: Inversion of hollow matrix of size 1000 – 4000 on CPU

### 5.2.5. Sparse Matrix

A sparse matrix is a matrix which is mostly populated with zeroes. Identity matrices and band matrices are types of sparse matrix in which the non-zero elements are concentrated around the main diagonal.

In sparse matrices the zero elements are distributed randomly in the matrix. There might be possible situations when these zero elements might be present in the main diagonal. This will result in calling the rowExchange() kernel function of our algorithm which will consume more computation time.

There might also be a situation when the zero elements are present in a symmetrical pattern in the input matrix. Since our algorithm discards computations on zeroes present in the *jth* column while executing the fixColumn() kernel function, the computation time for such cases will be lesser. This was also evident in the cases of identity matrices and band matrices. However, the position of zeroes in the sparse matrix we generated was random. Hence the time taken for its computation is almost similar to that of random matrix.

Table 6: Computation Time of Sparse Matrix GPU vs CPU

|  |  |  |  |
| --- | --- | --- | --- |
| **:::Sparse matrix (GPU vs CPU computation time comparison):::** | | | |
|  | **Computation Time (milliseconds)** |  |  |
| **Matrix size (n x n)** | **GPU** | **CPU** | **Speedup** |
| 5 | 0.15 | 0.01 | 0.06 |
| 10 | 0.22 | 0.03 | 0.13 |
| 15 | 0.29 | 0.07 | 0.25 |
| 20 | 0.34 | 0.15 | 0.44 |
| 25 | 0.40 | 0.52 | 1.29 |
| 30 | 0.48 | 0.52 | 1.07 |
| 35 | 0.55 | 1.00 | 1.83 |
| 40 | 0.61 | 1.39 | 2.26 |
| 45 | 0.67 | 1.84 | 2.73 |
| 50 | 0.74 | 2.38 | 3.23 |
| 55 | 0.80 | 3.10 | 3.87 |
| 60 | 0.75 | 3.15 | 4.18 |
| 65 | 0.96 | 4.84 | 5.02 |
| 70 | 1.01 | 5.97 | 5.89 |
| 75 | 1.00 | 7.28 | 7.28 |
| 80 | 1.04 | 8.75 | 8.45 |
| 85 | 1.25 | 9.39 | 7.55 |
| 90 | 1.31 | 9.66 | 7.36 |
| 95 | 1.37 | 11.36 | 8.31 |
| 100 | 1.49 | 13.00 | 8.73 |
| 200 | 4.22 | 58.85 | 13.95 |
| 300 | 10.59 | 185.92 | 17.55 |
| 400 | 23.98 | 432.43 | 18.03 |
| 500 | 35.01 | 838.86 | 23.96 |
| 600 | 67.92 | 1444.57 | 21.27 |
| 700 | 103.91 | 2294.40 | 22.08 |
| 800 | 126.38 | 3426.06 | 27.11 |
| 900 | 210.95 | 4881.25 | 23.14 |
| 1000 | 285.21 | 6713.52 | 23.54 |
| 1500 | 924.85 | 22775.67 | 24.63 |
| 2000 | 1773.00 | 53892.11 | 30.40 |
| 2500 | 4182.89 | 104882.45 | 25.07 |
| 3000 | 7159.45 | 181587.41 | 25.36 |
| 3500 | 9324.45 | 288673.69 | 30.96 |
| 4000 | 16883.90 | 427880.00 | 25.34 |

Figure 36: Inversion of sparse matrix of size 5 – 100 on GPU

Figure 37: Inversion of sparse matrix of size 100 – 1000 on GPU

Figure 38: Inversion of sparse matrix of size 1000– 4000 on GPU

Figure 39: Inversion of sparse matrix of size 5 – 100 on CPU

Figure 40: Inversion of sparse matrix of size 100 – 1000 on CPU

Figure 41: Inversion of sparse matrix of size 1000 – 4000 on CPU

# 6. Conclusion and Future Work

Inverting large matrices swiftly and precisely is useful to many applications and products. GPU computing can be used for massive parallelization of tasks and we have implemented a parallel version of the Gauss Jordan algorithm using it. We have tested the algorithm against five different kinds of matrices of different sizes – random matrix, identity matrix, band matrix, hollow matrix and sparse matrix. We also implemented a CPU version of the same algorithm and tested the same sets of matrices on the CPU as we did on the GPU.

We have also calculated the speedup gained by the GPU over CPU for every size of each kind of matrix we tested. Through our results we prove that the overall average speedup of the parallel Gauss Jordan algorithm is much faster than the sequential version. We therefore conclude that GPU computing has a much higher performance price ratio than CPU computing.

Sparse matrices are matrices which are mostly populated with zero elements. We believe that our implementation of the parallel version of the Gauss Jordan algorithm is not suitable for rapid inverse computation of sparse matrices. This is because the zeroes in such matrices are randomly distributed, which leads to calling the rowExchange() kernel function whenever our algorithm encounters main diagonal elements as zeroes, thus, leading to more computation time. The implemented algorithm does produce accurate inverse results for such matrices, but the time consumption for different sized matrices does not produce a linear graph. Considering this situation, we propose to develop a more efficient and faster algorithm for computing the inverse of sparse matrices in the future.

# 7. References

Bakkum P and Skadron K, Accelerating SQL Database Operations on a GPU with CUDA, GPGPU '10: Proceedings of the Third Workshop on General-Purpose Computation on Graphics Processing Units, New York, NY, USA, 2010. ACM, pp. 94-103.

Billiconan and Kavinguy, A Neural Network on GPU, March 13, 2008 CC (ASA2.5) <http://www.codeproject.com/Articles/24361/A-Neural-Network-on-GPU> (Accessed March 12, 2014).

Bini D, Capovani M, Romani F, Lotti G,O(n2.7799) complexity for n x n approximate matrix multiplication, Inf Process Lett, 8 (1979), pp. 234–235.

Che Shuai, Boyer Michael, Meng Jiayuan, Tarjan David, Sheaffer Jermey W, Skadron Kevin, A Performance Study of General-Purpose Applications on Graphics Processors Using CUDA, Journal of Parallel and Distributed Computing, October 2008.

Cherlin, Lecture 5 – Linear Dependence and Independence, <http://www.math.rutgers.edu/~cherlin/Courses/250/Lectures/250L05.html> (Accessed April 14, 2014).

Coppersmith D, Winograd S, Matrix multiplication via arithmetic progressions, J. Symbolic Comput, 9 (1990), pp. 251–280.

CUDA Toolkit Documentation - <http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities>. Accessed: April 11 2014.

Dąbrowski Adam, Pawłowski Pawel, Stankiewicz Mateusz, Misiorek Filip , Quasi-maximum Accuracy Floating-point Computations with GPGPU for Applications in Digital Signal Processing, IEEE Signal Processing Conference SPA 2011, Poznań 2011.

Davie, A.M.; Stothers, A.J. (2013), "Improved bound for complexity of matrix multiplication", Proceedings of the Royal Society of Edinburgh 143A: 351–370.

Govindaraju N. K, Lloyd B, Wang W, Lin M, Manocha D, Fast computation of database operations using graphics processors, in: Proc. of the ACM SIGMOD International Conference on Management of Data, 2004.

Gupta N, What is Compute Capability in CUDA - <http://cuda-programming.blogspot.com/2013/01/what-is-compute-capability-in-cuda.html> (Accessed - April 14, 2014)

Harris M. J, Baxter W. V, Scheuermann T, Lastra A, Simulation of cloud dynamics on graphics hardware, in: Proc. of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, 2003.

Inam Rafia, A\* Algorithm for Multicore Graphics Processors: Figure 4 – An example of processing flow, Chalmers University of Technology, Göteborg 2009.

Krüger J, Westermann R, Linear algebra operators for GPU implementation of numerical algorithms, ACM Transactions on Graphics 22 (3) (2003) 908-916.

Nadathur Satish, Mark Harris, and Michael Garland. Designing Efficient Sorting Algorithms for Manycore GPUs. In Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium, May 2009.

NVIDIA, GeForce 256, <http://www.nvidia.com/page/geforce256.html> (Accessed March 08, 2014).

NVIDIA, GTX 480, <http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-480/specifications> (Accessed April 10, 2014).

NVIDIA, K20 GPU ACCELERATOR, <http://www.nvidia.com/content/PDF/kepler/Tesla-K20-Passive-BD-06455-001-v07.pdf> (Accessed April 10, 2014).

Pan VY. Strassen’s algorithm is not optimal trilinear technique of aggregating, uniting and canceling for constructing fast algorithms for matrix operations. In: Proceedings of the 19thannual symposium on foundations of computer science. Ann Arbo; 1978. p. 166–176.

Richie D, Brown Deer Technology, GPU-Accelerated Computing for Chemistry and Material Simulations Using ATI Stream Technology, June, 2010.

Romani F, Some properties of disjoint sums of tensors related to matrix multiplication, SIAM J Compute, 11:263 – 267, 1982.

Romani F, Some properties of disjoint sums of tensors related to matrix multiplicationSIAM J Comput, 11 (1982), pp. 263–267.

Sanders Jason and Kandrot Edward, CUDA by Example: An Introduction to General-Purpose GPU Programming. Boston: Pearson Education, July, 2010.

Satish Nadathur, Harris Mark and Garland Michael, Designing Efficient Sorting Algorithms for Manycore GPUs, IPDPS: 1-10, 2009.

Sharma G, Agarwala A, Bhattacharya B, A fast parallel Gauss Jordan algorithm for matrix inversion using CUDA, Computers & Structures, Volume 128, Elsevier, November 2013: 31–37.

Strassen V, Gaussian elimination is not optimal, Number Math, 13 (1969), pp. 354–356.

Strassen V, The asymptotic spectrum of tensors, J Für Die Reine und Angewandte Mathematik, 384 (1988), pp. 102–154.

Thomas Anoop, HPC Platform options: Cell BE and GPU: High Performance Computing Group, NeST Software, November, 2009.

Williams V V, Multiplying matrices faster than Coppersmith-Winograd, Proceedings of the 44th symposium on Theory of Computing, ACM, 2011: pp. 887-898.

Yamanouchi T, AES encryption and decryption on the GPU, GPU Gems 3, Addison Wesley, 2007, pp. 785-803.

Zhang Hong, Zhang Da-fang, and Bi Xia-an, Comparison and Analysis of GPGPU and Parallel Computing on Multi-Core CPU, International Journal of Information and Education Technology, vol2, no2, 2012, pp. 185.

# APPENDIX A

In this section, we provide the actual results. For each matrix size, we made 5 different runs and calculated their median to give us the average time taken.

Table 7: Computation Time for Random Matrices on GPU

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **:::Random Matrix (GPU):::** | | | | | |  |
| **Matrix Size** | **Computation Time (ms)** | | | | | **Average Time (ms)** |
| 5 | 0.123 | 0.152 | 0.103 | 0.174 | 0.121 | 0.12 |
| 10 | 0.177 | 0.219 | 0.158 | 0.234 | 0.239 | 0.22 |
| 15 | 0.279 | 0.214 | 0.215 | 0.314 | 0.285 | 0.28 |
| 20 | 0.270 | 0.271 | 0.377 | 0.346 | 0.346 | 0.35 |
| 25 | 0.327 | 0.403 | 0.409 | 0.269 | 0.327 | 0.33 |
| 30 | 0.383 | 0.474 | 0.383 | 0.475 | 0.477 | 0.47 |
| 35 | 0.552 | 0.574 | 0.543 | 0.605 | 0.563 | 0.56 |
| 40 | 0.589 | 0.607 | 0.658 | 0.613 | 0.598 | 0.61 |
| 45 | 0.582 | 0.736 | 0.674 | 0.563 | 0.671 | 0.67 |
| 50 | 0.748 | 0.627 | 0.745 | 0.749 | 0.774 | 0.75 |
| 55 | 0.696 | 0.712 | 0.822 | 0.864 | 0.809 | 0.81 |
| 60 | 0.893 | 0.784 | 0.916 | 0.867 | 0.754 | 0.87 |
| 65 | 0.999 | 0.838 | 0.987 | 0.848 | 0.949 | 0.95 |
| 70 | 1.037 | 1.022 | 0.989 | 1.028 | 1.013 | 1.02 |
| 75 | 0.984 | 1.075 | 1.12 | 1.076 | 1.098 | 1.08 |
| 80 | 1.172 | 1.069 | 1.045 | 1.188 | 1.199 | 1.17 |
| 85 | 1.225 | 1.171 | 1.236 | 1.211 | 1.339 | 1.23 |
| 90 | 1.314 | 1.353 | 1.33 | 1.32 | 1.308 | 1.32 |
| 95 | 1.418 | 1.39 | 1.302 | 1.391 | 1.313 | 1.39 |
| 100 | 1.467 | 1.462 | 1.408 | 1.407 | 1.475 | 1.46 |
| 200 | 4.478 | 4.438 | 4.385 | 4.478 | 4.426 | 4.44 |
| 300 | 10.63 | 10.95 | 10.218 | 10.596 | 10.629 | 10.63 |
| 400 | 23.933 | 23.917 | 24.022 | 24.06 | 24.021 | 24.02 |
| 500 | 44.222 | 44.202 | 44.216 | 44.154 | 44.236 | 44.22 |
| 600 | 68.087 | 68.136 | 68.112 | 68.013 | 68.032 | 68.09 |
| 700 | 104.034 | 104.037 | 104.062 | 104.105 | 104.07 | 104.06 |
| 800 | 150.569 | 150.605 | 150.634 | 150.639 | 150.622 | 150.62 |
| 900 | 211.046 | 211.066 | 211.228 | 211.107 | 211.034 | 211.07 |
| 1000 | 285.461 | 285.366 | 285.48 | 285.512 | 285.468 | 285.47 |
| 1500 | 926.219 | 925.808 | 925.037 | 925.238 | 924.991 | 925.24 |
| 2000 | 2152.544 | 2153.184 | 2152.717 | 2152.9 | 2152.061 | 2152.72 |
| 2500 | 4186.131 | 4185.58 | 4186.316 | 4185.525 | 4183.706 | 4185.58 |
| 3000 | 7160.463 | 7159.539 | 7161.359 | 7162.708 | 7161.488 | 7161.36 |
| 3500 | 11409.52 | 11400.98 | 11406.9 | 11400.32 | 11402.93 | 11402.93 |
| 4000 | 16892.49 | 16890.62 | 16880.88 | 16888.23 | 16883.9 | 16888.23 |

Table 8: Computation Time for Random Matrix on CPU

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **:::Random matrix (CPU):::** | | | | | |  |
| **Matrix Size** | **Computation Time(ms)** | | | | | **Average time(ms)** |
| 5 | 0.008 | 0.013 | 0.014 | 0.012 | 0.014 | 0.01 |
| 10 | 0.028 | 0.015 | 0.014 | 0.029 | 0.014 | 0.02 |
| 15 | 0.031 | 0.07 | 0.074 | 0.07 | 0.071 | 0.07 |
| 20 | 0.064 | 0.149 | 0.149 | 0.065 | 0.153 | 0.15 |
| 25 | 0.545 | 0.545 | 0.546 | 0.116 | 0.117 | 0.55 |
| 30 | 0.807 | 0.51 | 0.8 | 0.768 | 0.74 | 0.77 |
| 35 | 1.002 | 1.084 | 1.025 | 1.023 | 1.054 | 1.03 |
| 40 | 1.444 | 0.714 | 1.41 | 1.394 | 0.714 | 1.39 |
| 45 | 1.016 | 0.632 | 1.849 | 1.84 | 0.902 | 1.02 |
| 50 | 2.443 | 2.435 | 2.419 | 2.412 | 2.521 | 2.44 |
| 55 | 3.094 | 3.072 | 3.082 | 3.12 | 3.082 | 3.08 |
| 60 | 3.877 | 1.805 | 2.011 | 1.476 | 3.899 | 2.01 |
| 65 | 4.848 | 4.351 | 4.855 | 2.191 | 4.816 | 4.82 |
| 70 | 5.993 | 5.982 | 2.653 | 2.667 | 2.344 | 2.67 |
| 75 | 7.296 | 6.562 | 3.165 | 7.264 | 2.888 | 6.56 |
| 80 | 8.754 | 8.742 | 8.759 | 3.759 | 8.751 | 8.75 |
| 85 | 7.04 | 8.951 | 9.355 | 9.325 | 10.012 | 9.33 |
| 90 | 8.606 | 12.306 | 12.288 | 9.218 | 5.222 | 9.22 |
| 95 | 5.806 | 10.461 | 13.881 | 14.491 | 6.042 | 10.46 |
| 100 | 7.066 | 14.193 | 11.859 | 14.72 | 14.071 | 14.07 |
| 200 | 59.521 | 56.367 | 61.622 | 53.972 | 61.634 | 59.52 |
| 300 | 186.838 | 186.657 | 183.203 | 185.586 | 187.546 | 186.66 |
| 400 | 433.842 | 438.363 | 435.767 | 433.52 | 433.63 | 433.84 |
| 500 | 835.093 | 838.388 | 842.22 | 843.419 | 838.416 | 838.42 |
| 600 | 1442.948 | 1449.461 | 1449.203 | 1451.77 | 1447.451 | 1449.20 |
| 700 | 2291.544 | 2296.332 | 2298.143 | 2292.329 | 2297.101 | 2296.33 |
| 800 | 3423.666 | 3442.591 | 3430.455 | 3429.267 | 3423.047 | 3429.27 |
| 900 | 4889.28 | 4886.46 | 4884.929 | 4882.167 | 4887.364 | 4886.46 |
| 1000 | 6711.93 | 6724.764 | 6714.017 | 6713.903 | 6722.626 | 6714.02 |
| 1500 | 22758.217 | 22754.773 | 22767.928 | 22753.727 | 22768.627 | 22758.22 |
| 2000 | 53830.652 | 53854.355 | 53845.59 | 53885.699 | 53831.375 | 53845.59 |
| 2500 | 104919.27 | 104850.25 | 104837.477 | 104956.031 | 104868.969 | 104868.97 |
| 3000 | 180746.52 | 180827.11 | 18030.609 | 180783.5 | 180802.266 | 180783.50 |
| 3500 | 287660.09 | 287949.13 | 287653.219 | 287859.625 | 287528.281 | 287660.09 |
| 4000 | 428346.19 | 427796.13 | 427920.469 | 428630.312 | 428133.594 | 428133.59 |

Table 9: Computation Time for Identity Matrix on GPU

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **:::Identity matrix (GPU):::** | | | | | |  |
| **Matrix Size** | **Computation Time(ms)** | | | | | **Average time(ms)** |
| 5 | 0.109 | 0.104 | 0.152 | 0.177 | 0.102 | 0.11 |
| 10 | 0.221 | 0.152 | 0.215 | 0.223 | 0.215 | 0.22 |
| 15 | 0.283 | 0.207 | 0.277 | 0.299 | 0.206 | 0.28 |
| 20 | 0.259 | 0.424 | 0.259 | 0.35 | 0.343 | 0.34 |
| 25 | 0.412 | 0.407 | 0.444 | 0.418 | 0.385 | 0.41 |
| 30 | 0.472 | 0.472 | 0.47 | 0.473 | 0.496 | 0.47 |
| 35 | 0.54 | 0.565 | 0.56 | 0.542 | 0.533 | 0.54 |
| 40 | 0.588 | 0.478 | 0.605 | 0.481 | 0.604 | 0.59 |
| 45 | 0.716 | 0.684 | 0.689 | 0.534 | 0.702 | 0.69 |
| 50 | 0.787 | 0.744 | 0.593 | 0.741 | 0.77 | 0.74 |
| 55 | 0.797 | 0.865 | 0.8 | 0.845 | 0.84 | 0.84 |
| 60 | 0.909 | 0.713 | 0.878 | 0.906 | 0.871 | 0.88 |
| 65 | 0.983 | 0.793 | 0.955 | 0.9444 | 0.988 | 0.96 |
| 70 | 1.049 | 0.989 | 1.016 | 1.005 | 0.991 | 1.01 |
| 75 | 1.095 | 1.063 | 0.925 | 0.921 | 1.07 | 1.06 |
| 80 | 1.257 | 1.173 | 1.127 | 1.132 | 0.991 | 1.13 |
| 85 | 1.209 | 1.275 | 1.076 | 1.079 | 1.21 | 1.21 |
| 90 | 1.151 | 1.422 | 1.141 | 1.14 | 1.295 | 1.15 |
| 95 | 1.328 | 1.356 | 1.316 | 1.216 | 1.364 | 1.33 |
| 100 | 1.302 | 1.321 | 1.443 | 1.406 | 1.42 | 1.41 |
| 200 | 3.975 | 3.899 | 3.874 | 3.878 | 3.978 | 3.90 |
| 300 | 9.137 | 9.09 | 9.152 | 9.142 | 9.508 | 9.14 |
| 400 | 19.812 | 19.694 | 19.788 | 19.805 | 19.7 | 19.79 |
| 500 | 31.382 | 31.418 | 31.516 | 31.424 | 31.085 | 31.42 |
| 600 | 49.746 | 49.73 | 49.733 | 49.617 | 49.619 | 49.73 |
| 700 | 75.26 | 75.252 | 75.19 | 75.252 | 75.197 | 75.25 |
| 800 | 108.152 | 108.584 | 108.62 | 108.602 | 108.604 | 108.60 |
| 900 | 151.173 | 150.85 | 151.224 | 151.213 | 151.189 | 151.19 |
| 1000 | 204.033 | 203.663 | 203.909 | 203.929 | 203.629 | 203.91 |
| 1500 | 659.957 | 659.81 | 660.067 | 660.025 | 659.958 | 659.96 |
| 2000 | 1537.816 | 1538.292 | 1538.002 | 1537.932 | 1538.368 | 1538.00 |
| 2500 | 2981.808 | 2979.852 | 2980.039 | 2979.627 | 2980.125 | 2980.04 |
| 3000 | 5124.848 | 5127.298 | 5125.916 | 5128.036 | 5125.672 | 5125.92 |
| 3500 | 8113.578 | 8114.768 | 8113.123 | 8112.235 | 8116.2 | 8113.58 |
| 4000 | 12086.64 | 12083.45 | 12084.09 | 12079.6 | 12086.8 | 12084.09 |

Table 10: Computation Time for Identity Matrix on CPU

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **:::Identity matrix (CPU):::** | | | | | |  |
| **Matrix Size** | **Computation Time(ms)** | | | | | **Average time(ms)** |
| 5 | 0.008 | 0.013 | 0.02 | 0.016 | 0.008 | 0.01 |
| 10 | 0.028 | 0.014 | 0.03 | 0.029 | 0.033 | 0.03 |
| 15 | 0.031 | 0.031 | 0.031 | 0.07 | 0.07 | 0.03 |
| 20 | 0.15 | 0.149 | 0.511 | 0.518 | 0.148 | 0.15 |
| 25 | 0.546 | 0.541 | 0.524 | 0.675 | 0.547 | 0.55 |
| 30 | 0.523 | 0.517 | 0.746 | 0.766 | 0.768 | 0.75 |
| 35 | 1.045 | 1.08 | 1.067 | 1.042 | 1.042 | 1.05 |
| 40 | 1.409 | 1.399 | 0.737 | 1.392 | 1.401 | 1.40 |
| 45 | 1.845 | 1.841 | 1.852 | 0.896 | 1.014 | 1.84 |
| 50 | 2.376 | 2.412 | 2.434 | 2.415 | 1.159 | 2.41 |
| 55 | 3.104 | 3.073 | 3.136 | 2.794 | 3.115 | 3.10 |
| 60 | 1.777 | 3.91 | 3.994 | 4.002 | 4.003 | 3.99 |
| 65 | 4.826 | 4.854 | 2.178 | 4.827 | 4.845 | 4.83 |
| 70 | 6.001 | 2.647 | 5.972 | 5.493 | 5.973 | 5.97 |
| 75 | 7.344 | 5.715 | 3.198 | 7.279 | 7.303 | 7.28 |
| 80 | 7.685 | 8.746 | 8.731 | 8.706 | 8.717 | 8.72 |
| 85 | 10.405 | 7.448 | 4.438 | 9.228 | 4.553 | 7.45 |
| 90 | 12.228 | 7.65 | 7.606 | 9.774 | 11.311 | 9.77 |
| 95 | 12.99 | 14.532 | 13.291 | 6.054 | 6.068 | 12.99 |
| 100 | 12.776 | 16.532 | 13.222 | 7.027 | 11.865 | 12.78 |
| 200 | 61.616 | 53.939 | 56.705 | 61.206 | 60.045 | 60.05 |
| 300 | 181.094 | 183.404 | 189.1 | 188.248 | 180.902 | 183.40 |
| 400 | 434.394 | 433.908 | 428.575 | 427.738 | 434.956 | 433.91 |
| 500 | 846.404 | 835.455 | 842.01 | 841.596 | 838.485 | 841.60 |
| 600 | 1446.634 | 1450.494 | 1454.788 | 1449.185 | 1445.234 | 1449.19 |
| 700 | 2298.322 | 2299.158 | 2299.323 | 2293.523 | 2290.81 | 2298.32 |
| 800 | 3425.275 | 3417.699 | 3431.928 | 3424.771 | 3427.101 | 3425.28 |
| 900 | 4885.227 | 4876.252 | 4875.629 | 4878.558 | 4878.484 | 4878.48 |
| 1000 | 6727.064 | 6712.846 | 6712.511 | 6714.991 | 6714.303 | 6714.30 |
| 1500 | 22783.92 | 22761.979 | 22744.896 | 22759.174 | 22736.799 | 22759.17 |
| 2000 | 53955.887 | 53921.844 | 53881.539 | 53876.812 | 53846.547 | 53881.54 |
| 2500 | 104995.695 | 104932.906 | 104841.273 | 105010.469 | 104942.156 | 104942.16 |
| 3000 | 181348.156 | 180791.562 | 180889.906 | 180901.016 | 180765.906 | 180889.91 |
| 3500 | 287924.750 | 287428.062 | 287443 | 287826.188 | 287670.969 | 287670.97 |
| 4000 | 428576.031 | 429001 | 427932.25 | 428087.375 | 428052.312 | 428087.38 |

Table 11: Computation Time for Band Matrix on GPU

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **:::Band matrix (GPU):::** | | | | | |  |
| **Matrix Size** | **Computation Time (ms)** | | | | | **Average time (ms)** |
| 5 | 0.118 | 0.12 | 0.102 | 0.181 | 0.101 | 0.12 |
| 10 | 0.158 | 0.217 | 0.219 | 0.251 | 0.216 | 0.22 |
| 15 | 0.284 | 0.307 | 0.287 | 0.282 | 0.299 | 0.29 |
| 20 | 0.340 | 0.364 | 0.346 | 0.34 | 0.341 | 0.34 |
| 25 | 0.418 | 0.324 | 0.407 | 0.341 | 0.411 | 0.41 |
| 30 | 0.49 | 0.493 | 0.487 | 0.382 | 0.481 | 0.49 |
| 35 | 0.539 | 0.62 | 0.545 | 0.539 | 0.549 | 0.55 |
| 40 | 0.495 | 0.593 | 0.615 | 0.606 | 0.618 | 0.61 |
| 45 | 0.713 | 0.747 | 0.693 | 0.686 | 0.733 | 0.71 |
| 50 | 0.752 | 0.615 | 0.628 | 0.74 | 0.746 | 0.74 |
| 55 | 0.806 | 0.822 | 0.679 | 0.677 | 0.81 | 0.81 |
| 60 | 0.921 | 0.947 | 0.738 | 0.908 | 0.898 | 0.91 |
| 65 | 0.814 | 0.937 | 0.814 | 0.815 | 0.815 | 0.82 |
| 70 | 0.882 | 1.054 | 0.885 | 1.029 | 1.037 | 1.03 |
| 75 | 1.103 | 1.108 | 1.094 | 1.136 | 1.106 | 1.11 |
| 80 | 1.18 | 1.18 | 1.193 | 1.029 | 1.204 | 1.18 |
| 85 | 1.108 | 1.215 | 1.317 | 1.108 | 1.253 | 1.22 |
| 90 | 1.318 | 1.335 | 1.282 | 1.177 | 1.34 | 1.32 |
| 95 | 1.335 | 1.279 | 1.363 | 1.349 | 1.354 | 1.35 |
| 100 | 1.433 | 1.35 | 1.423 | 1.418 | 1.489 | 1.42 |
| 200 | 4.245 | 4.211 | 4.225 | 4.122 | 4.114 | 4.21 |
| 300 | 9.855 | 9.84 | 10.171 | 10.208 | 9.998 | 10.00 |
| 400 | 21.5 | 21.496 | 21.489 | 21.55 | 21.523 | 21.50 |
| 500 | 34.917 | 34.849 | 34.952 | 34.901 | 34.513 | 34.90 |
| 600 | 55.235 | 55.398 | 54.964 | 55.429 | 55.293 | 55.29 |
| 700 | 84.617 | 84.719 | 84.618 | 84.527 | 84.617 | 84.62 |
| 800 | 121.826 | 121.881 | 121.864 | 121.687 | 121.911 | 121.86 |
| 900 | 169.028 | 168.896 | 168.534 | 168.908 | 168.863 | 168.90 |
| 1000 | 226.733 | 226.607 | 226.636 | 226.231 | 226.199 | 226.61 |
| 1500 | 713.865 | 713.453 | 713.493 | 713.577 | 713.312 | 713.49 |
| 2000 | 1632.315 | 1632.476 | 1632.402 | 1631.646 | 1631.644 | 1632.32 |
| 2500 | 3129.728 | 3129.177 | 3129.885 | 3129.676 | 3128.962 | 3129.68 |
| 3000 | 5344.179 | 5342.392 | 5345.222 | 5344.753 | 5343.282 | 5344.18 |
| 3500 | 8415.179 | 8416.466 | 8418.009 | 8416.706 | 8420.61 | 8416.71 |
| 4000 | 12472.1 | 12473.02 | 12475.64 | 12470.5 | 12472.65 | 12472.65 |

Table 12: Computation Time for Band Matrix on CPU

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **:::Band matrix (CPU):::** | | | | | |  |
| **Matrix Size** | **Computation Time (ms)** | | | | | **Average time(ms)** |
| 5 | 0.008 | 0.016 | 0.012 | 0.016 | 0.012 | 0.01 |
| 10 | 0.031 | 0.028 | 0.028 | 0.028 | 0.029 | 0.03 |
| 15 | 0.068 | 0.068 | 0.069 | 0.067 | 0.068 | 0.07 |
| 20 | 0.145 | 0.143 | 0.144 | 0.062 | 0.144 | 0.14 |
| 25 | 0.547 | 0.549 | 0.55 | 0.113 | 0.702 | 0.55 |
| 30 | 0.737 | 0.75 | 0.756 | 0.732 | 0.769 | 0.75 |
| 35 | 1.005 | 0.592 | 0.586 | 1.011 | 0.587 | 0.59 |
| 40 | 1.39 | 1.383 | 1.393 | 1.429 | 1.389 | 1.39 |
| 45 | 0.98 | 1.814 | 0.619 | 1.81 | 1.869 | 1.81 |
| 50 | 2.376 | 1.167 | 0.848 | 2.357 | 2.411 | 2.36 |
| 55 | 3.068 | 1.425 | 3.049 | 3.031 | 3.044 | 3.04 |
| 60 | 3.918 | 3.978 | 3.977 | 2.002 | 1.468 | 3.92 |
| 65 | 4.85 | 4.832 | 4.866 | 4.816 | 4.821 | 4.83 |
| 70 | 5.964 | 5.99 | 4.705 | 2.64 | 2.616 | 4.71 |
| 75 | 7.29 | 7.269 | 7.234 | 7.261 | 7.271 | 7.27 |
| 80 | 7.169 | 3.752 | 5.13 | 3.759 | 7.688 | 5.13 |
| 85 | 10.431 | 4.562 | 6.767 | 9.26 | 10.435 | 9.26 |
| 90 | 5.226 | 12.314 | 12.345 | 12.391 | 12.302 | 12.31 |
| 95 | 9.525 | 10.23 | 10.656 | 6.022 | 12.724 | 10.23 |
| 100 | 7.03 | 11.648 | 12.03 | 12.29 | 7.002 | 11.65 |
| 200 | 53.685 | 53.705 | 53.747 | 56.141 | 58.459 | 53.75 |
| 300 | 249.332 | 249.176 | 247.382 | 250.089 | 250.862 | 249.33 |
| 400 | 600.625 | 597.072 | 603.648 | 602.574 | 608.778 | 602.57 |
| 500 | 1117.305 | 1112.946 | 1112.943 | 1112.051 | 1110.203 | 1112.94 |
| 600 | 1824.23 | 1830.386 | 1828.159 | 1835.768 | 1828.531 | 1828.53 |
| 700 | 2768.398 | 2766.806 | 2767.105 | 2765.2 | 2770.966 | 2767.11 |
| 800 | 3977.674 | 3977.409 | 3976.995 | 3974.495 | 3981.479 | 3977.41 |
| 900 | 5531.133 | 5526.669 | 5525.698 | 5529.147 | 5537.05 | 5529.15 |
| 1000 | 7467.969 | 7477.731 | 7474.104 | 7468.849 | 7483.2 | 7474.10 |
| 1500 | 23995.314 | 23955.258 | 23944.217 | 23963.229 | 23959.875 | 23959.88 |
| 2000 | 55420.254 | 55398.512 | 55421.047 | 55447.113 | 55418.902 | 55420.25 |
| 2500 | 106875.492 | 106856.539 | 106915.5 | 106895.195 | 106838.992 | 106875.49 |
| 3000 | 183286.578 | 183193.219 | 183116.438 | 183151.578 | 183176.484 | 183176.48 |
| 3500 | 290226.219 | 290247.25 | 290145.688 | 290550.875 | 290320.125 | 290247.25 |
| 4000 | 430762.094 | 430831.406 | 430867.281 | 431109.5 | 430832.031 | 430832.03 |

Table 13: Computation Time for Hollow Matrix on GPU

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **:::Hollow matrix (GPU):::** | | | | | |  |
| **Matrix Size** | **Computation Time(ms)** | | | | | **Average time(ms)** |
| 5 | 0.143 | 0.195 | 0.224 | 0.186 | 0.126 | 0.19 |
| 10 | 0.286 | 0.204 | 0.286 | 0.284 | 0.283 | 0.28 |
| 15 | 0.3 | 0.388 | 0.428 | 0.377 | 0.379 | 0.38 |
| 20 | 0.489 | 0.479 | 0.475 | 0.477 | 0.362 | 0.48 |
| 25 | 0.614 | 0.623 | 0.457 | 0.457 | 0.445 | 0.46 |
| 30 | 0.66 | 0.686 | 0.524 | 0.683 | 0.676 | 0.68 |
| 35 | 0.605 | 0.781 | 0.799 | 0.775 | 0.817 | 0.78 |
| 40 | 0.923 | 0.684 | 0.852 | 0.898 | 0.857 | 0.86 |
| 45 | 0.958 | 0.972 | 0.95 | 0.946 | 0.767 | 0.95 |
| 50 | 0.866 | 1.092 | 1.095 | 0.859 | 1.126 | 1.09 |
| 55 | 1.165 | 1.171 | 1.194 | 1.154 | 0.963 | 1.17 |
| 60 | 1.244 | 1.046 | 1.292 | 1.286 | 1.29 | 1.29 |
| 65 | 1.368 | 1.138 | 1.141 | 1.395 | 1.138 | 1.14 |
| 70 | 1.493 | 1.244 | 1.246 | 1.461 | 1.436 | 1.44 |
| 75 | 1.606 | 1.332 | 1.563 | 1.597 | 1.549 | 1.56 |
| 80 | 1.725 | 1.687 | 1.636 | 1.412 | 1.635 | 1.64 |
| 85 | 1.84 | 1.788 | 1.56 | 1.55 | 1.722 | 1.72 |
| 90 | 1.918 | 1.873 | 1.917 | 1.85 | 1.911 | 1.91 |
| 95 | 1.957 | 1.97 | 2.018 | 2.157 | 1.933 | 1.97 |
| 100 | 1.887 | 1.865 | 2.022 | 1.872 | 2.076 | 1.89 |
| 200 | 5.392 | 5.347 | 5.34 | 5.426 | 5.427 | 5.39 |
| 300 | 12.05 | 12.439 | 12.099 | 12.083 | 12.221 | 12.10 |
| 400 | 26.317 | 25.989 | 26.345 | 26.342 | 26.109 | 26.32 |
| 500 | 47.165 | 47.035 | 47.007 | 47.157 | 47.147 | 47.15 |
| 600 | 70.485 | 70.46 | 70.358 | 70.378 | 70.435 | 70.44 |
| 700 | 107.624 | 107.583 | 107.507 | 107.482 | 107.541 | 107.54 |
| 800 | 154.516 | 154.596 | 154.775 | 154.292 | 154.657 | 154.60 |
| 900 | 215.88 | 216.006 | 215.454 | 215.625 | 215.336 | 215.63 |
| 1000 | 290.801 | 290.523 | 290.689 | 290.434 | 290.556 | 290.56 |
| 1500 | 932.677 | 932.833 | 932.825 | 932.694 | 932.836 | 932.83 |
| 2000 | 2161.72 | 2160.833 | 2161.929 | 2161.58 | 2162.31 | 2161.72 |
| 2500 | 4197.205 | 4196.954 | 4195.906 | 4198.099 | 4195.885 | 4196.95 |
| 3000 | 7178.842 | 7176.755 | 7176.73 | 7174.084 | 7175.271 | 7176.73 |
| 3500 | 11417.99 | 11425.44 | 11421.36 | 11425.77 | 11421.99 | 11421.99 |
| 4000 | 16906.48 | 16913.14 | 16907 | 16905.68 | 16910.59 | 16907.00 |

Table 14: Computation Time for Hollow Matrix on CPU

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **:::Hollow matrix (CPU):::** | | | | | |  |
| **Matrix Size** | **Computation Time (ms)** | | | | | **Average time (ms)** |
| 5 | 0.013 | 0.013 | 0.008 | 0.008 | 0.008 | 0.01 |
| 10 | 0.029 | 0.029 | 0.029 | 0.028 | 0.028 | 0.03 |
| 15 | 0.069 | 0.031 | 0.069 | 0.069 | 0.03 | 0.07 |
| 20 | 0.145 | 0.063 | 0.145 | 0.146 | 0.146 | 0.15 |
| 25 | 0.543 | 0.538 | 0.113 | 0.543 | 0.114 | 0.54 |
| 30 | 0.733 | 0.193 | 0.736 | 0.747 | 0.753 | 0.74 |
| 35 | 0.595 | 0.588 | 1.037 | 1.079 | 1.028 | 1.03 |
| 40 | 0.743 | 0.81 | 1.394 | 1.387 | 1.425 | 1.39 |
| 45 | 1.01 | 1.791 | 1.856 | 1.788 | 1.826 | 1.79 |
| 50 | 2.339 | 1.14 | 1.151 | 2.36 | 0.843 | 1.15 |
| 55 | 3.037 | 3.033 | 3.076 | 3.037 | 1.129 | 3.04 |
| 60 | 3.978 | 3.929 | 3.985 | 4.03 | 3.914 | 3.98 |
| 65 | 4.794 | 4.687 | 4.811 | 4.861 | 3.272 | 4.79 |
| 70 | 3.702 | 2.323 | 2.647 | 5.049 | 5.721 | 3.70 |
| 75 | 3.182 | 4.4 | 7.199 | 3.173 | 7.247 | 4.40 |
| 80 | 5.871 | 8.752 | 3.46 | 6.478 | 8.162 | 6.48 |
| 85 | 7.769 | 9.033 | 4.181 | 10.131 | 10.396 | 9.03 |
| 90 | 12.255 | 12.282 | 5.215 | 10.091 | 12.342 | 12.26 |
| 95 | 13.835 | 8.973 | 9.479 | 11.522 | 10.113 | 10.11 |
| 100 | 13.956 | 11.72 | 12.02 | 6.751 | 6.788 | 11.72 |
| 200 | 58.514 | 53.764 | 53.798 | 58.479 | 62.018 | 58.48 |
| 300 | 186.6 | 186.19 | 190.475 | 187.99 | 187.148 | 187.15 |
| 400 | 426.638 | 430.706 | 431.959 | 434.637 | 430.862 | 430.86 |
| 500 | 836.99 | 836.891 | 834.851 | 834.663 | 834.053 | 834.85 |
| 600 | 1443.288 | 1437.147 | 1436.187 | 1446.522 | 1437.442 | 1437.44 |
| 700 | 2290.822 | 2289.347 | 2295.144 | 2285.333 | 2289.629 | 2289.63 |
| 800 | 3409.259 | 3412.356 | 3414.871 | 3404.842 | 3409.519 | 3409.52 |
| 900 | 4861.001 | 4858.11 | 4863.643 | 4859.058 | 4859.058 | 4859.06 |
| 1000 | 6694.654 | 6689.672 | 6692.26 | 6688.152 | 6691.295 | 6691.30 |
| 1500 | 22674.414 | 22922.523 | 22692.193 | 22684.529 | 22686.441 | 22686.44 |
| 2000 | 53640.785 | 53768.594 | 53665.27 | 53703.969 | 53677.531 | 53677.53 |
| 2500 | 104623.758 | 104661.688 | 104547.648 | 104689.023 | 104648.297 | 104648.30 |
| 3000 | 180400.875 | 180565.641 | 180565.641 | 180442.156 | 180509.25 | 180509.25 |
| 3500 | 287145.031 | 287560 | 287067.25 | 287007.875 | 287050.188 | 287067.25 |
| 4000 | 427528.5 | 427536.25 | 427538.875 | 427572.156 | 427568.094 | 427538.88 |

Table 15: Computation Time for Sparse Matrix on GPU

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **:::Sparse matrix (GPU):::** | | | | | |  |
| **Matrix Size** | **Computation Time (ms)** | | | | | **Average time (ms)** |
| 5 | 0.112 | 0.155 | 0.102 | 0.155 | 0.154 | 0.15 |
| 10 | 0.214 | 0.215 | 0.166 | 0.217 | 0.248 | 0.22 |
| 15 | 0.307 | 0.327 | 0.285 | 0.214 | 0.28 | 0.29 |
| 20 | 0.344 | 0.346 | 0.393 | 0.343 | 0.268 | 0.34 |
| 25 | 0.326 | 0.407 | 0.413 | 0.328 | 0.402 | 0.40 |
| 30 | 0.482 | 0.547 | 0.518 | 0.383 | 0.384 | 0.48 |
| 35 | 0.459 | 0.458 | 0.545 | 0.57 | 0.576 | 0.55 |
| 40 | 0.6 | 0.5 | 0.613 | 0.639 | 0.617 | 0.61 |
| 45 | 0.562 | 0.724 | 0.668 | 0.683 | 0.674 | 0.67 |
| 50 | 0.737 | 0.79 | 0.754 | 0.645 | 0.639 | 0.74 |
| 55 | 0.861 | 0.8 | 0.884 | 0.693 | 0.697 | 0.80 |
| 60 | 0.75 | 0.752 | 0.756 | 0.753 | 0.874 | 0.75 |
| 65 | 1.013 | 0.964 | 0.835 | 0.939 | 0.987 | 0.96 |
| 70 | 1.026 | 1.052 | 1.013 | 0.915 | 1.006 | 1.01 |
| 75 | 0.983 | 1.001 | 0.98 | 1.118 | 1.125 | 1.00 |
| 80 | 1.194 | 1.035 | 1.026 | 1.139 | 1.026 | 1.04 |
| 85 | 1.245 | 1.245 | 1.154 | 1.23 | 1.261 | 1.25 |
| 90 | 1.355 | 1.302 | 1.221 | 1.313 | 1.33 | 1.31 |
| 95 | 1.405 | 1.482 | 1.279 | 1.324 | 1.368 | 1.37 |
| 100 | 1.489 | 1.459 | 1.457 | 1.494 | 1.498 | 1.49 |
| 200 | 4.215 | 4.242 | 4.218 | 4.226 | 4.219 | 4.22 |
| 300 | 10.938 | 10.93 | 10.528 | 10.591 | 10.205 | 10.59 |
| 400 | 23.978 | 23.978 | 23.547 | 23.887 | 24.044 | 23.98 |
| 500 | 34.972 | 35.017 | 35.053 | 35.009 | 35.006 | 35.01 |
| 600 | 68.044 | 67.895 | 68.054 | 67.924 | 67.888 | 67.92 |
| 700 | 103.624 | 104.046 | 104.083 | 103.882 | 103.907 | 103.91 |
| 800 | 126.351 | 126.413 | 126.395 | 126.346 | 126.379 | 126.38 |
| 900 | 210.539 | 211.131 | 210.954 | 210.636 | 211.177 | 210.95 |
| 1000 | 285.492 | 285.208 | 285.326 | 285 | 285.129 | 285.21 |
| 1500 | 925.641 | 924.728 | 925.056 | 924.845 | 924.681 | 924.85 |
| 2000 | 1772.997 | 1773.565 | 1773.742 | 1772.728 | 1772.76 | 1773.00 |
| 2500 | 4182.894 | 4182.781 | 4186.575 | 4183.24 | 4181.92 | 4182.89 |
| 3000 | 7162.898 | 7160.984 | 7158.872 | 7158.498 | 7159.449 | 7159.45 |
| 3500 | 9324.453 | 9323.953 | 9323.123 | 9327.125 | 9327.72 | 9324.45 |
| 4000 | 16881.37 | 16881.63 | 16889.61 | 16883.9 | 16888.34 | 16883.90 |

Table 16: Computation Time for Sparse Matrix on CPU

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **:::Sparse matrix (CPU):::** | | | | | |  |
| **Matrix Size** | **Computation Time (ms)** | | | | | **Average time (ms)** |
| 5 | 0.009 | 0.008 | 0.009 | 0.007 | 0.01 | 0.01 |
| 10 | 0.029 | 0.028 | 0.029 | 0.028 | 0.031 | 0.03 |
| 15 | 0.07 | 0.071 | 0.071 | 0.07 | 0.071 | 0.07 |
| 20 | 0.151 | 0.063 | 0.148 | 0.523 | 0.518 | 0.15 |
| 25 | 0.115 | 0.519 | 0.115 | 0.702 | 0.544 | 0.52 |
| 30 | 0.516 | 0.765 | 0.494 | 0.519 | 0.488 | 0.52 |
| 35 | 0.995 | 1.063 | 0.594 | 0.597 | 1.024 | 1.00 |
| 40 | 1.386 | 0.716 | 1.394 | 1.08 | 1.43 | 1.39 |
| 45 | 1.892 | 1.843 | 1.822 | 1.842 | 1.844 | 1.84 |
| 50 | 2.435 | 2.501 | 2.38 | 1.164 | 1.153 | 2.38 |
| 55 | 3.121 | 3.076 | 3.114 | 3.088 | 3.095 | 3.10 |
| 60 | 3.922 | 1.796 | 3.973 | 1.76 | 3.147 | 3.15 |
| 65 | 4.884 | 4.836 | 4.839 | 4.866 | 4.829 | 4.84 |
| 70 | 5.968 | 5.992 | 5.963 | 5.97 | 2.648 | 5.97 |
| 75 | 7.305 | 7.284 | 3.191 | 3.159 | 7.284 | 7.28 |
| 80 | 3.759 | 8.753 | 8.75 | 8.825 | 7.447 | 8.75 |
| 85 | 9.394 | 4.436 | 9.79 | 7.239 | 10.441 | 9.39 |
| 90 | 9.661 | 9.498 | 9.387 | 9.944 | 12.335 | 9.66 |
| 95 | 11.362 | 11.916 | 14.433 | 6.07 | 9.335 | 11.36 |
| 100 | 14.113 | 16.58 | 13 | 11.799 | 11.768 | 13.00 |
| 200 | 59.219 | 58.595 | 56.352 | 59.865 | 58.846 | 58.85 |
| 300 | 185.918 | 181.345 | 189.178 | 186.824 | 183.514 | 185.92 |
| 400 | 433.007 | 430.526 | 432.428 | 435.556 | 428.192 | 432.43 |
| 500 | 839.571 | 838.855 | 836.287 | 840.386 | 835.679 | 838.86 |
| 600 | 1442.722 | 1449.592 | 1442.992 | 1450.762 | 1444.573 | 1444.57 |
| 700 | 2294.402 | 2290.844 | 2294.131 | 2295.058 | 2303.222 | 2294.40 |
| 800 | 3426.072 | 3426.999 | 3424.163 | 3420.661 | 3426.063 | 3426.06 |
| 900 | 4874.795 | 4887.782 | 4881.249 | 4879.497 | 4884.944 | 4881.25 |
| 1000 | 6713.521 | 6717.623 | 6713.424 | 6708.056 | 6719.98 | 6713.52 |
| 1500 | 22758.422 | 22776.896 | 22770.668 | 22776.68 | 22775.672 | 22775.67 |
| 2000 | 53950.168 | 53885.609 | 53892.113 | 53842.148 | 53926.012 | 53892.11 |
| 2500 | 104882.445 | 104908.797 | 104883.688 | 104844.312 | 104878.469 | 104882.45 |
| 3000 | 181925.953 | 181387.484 | 181158.047 | 181587.406 | 182075.484 | 181587.41 |
| 3500 | 288673.688 | 288943.125 | 288825.312 | 287651.188 | 287693.281 | 288673.69 |
| 4000 | 428222.156 | 427880 | 427773.375 | 427779.688 | 428019.969 | 427880.00 |