APPLICATION OF GRAPHICS PROCESSING UNIT FOR PARALLEL PROCESSING IN STRUCTURAL ENGINEERING

By

Vivek K. Patel

13MCLC12

DEPARTMENT OF CIVIL ENGINEERING

INSTITUTE OF TECHNOLOGY

NIRMA UNIVERSITY

AHMEDABAD-382481

May 2015

APPLICATION OF GRAPHICS PROCESSING UNIT FOR PARALLEL PROCESSING IN STRUCTURAL ENGINEERING

Major Project

Submitted in partial fulfillment of the requirements For the degree of

Master of Technology

In

Civil Engineering

(Computer Aided Structural Analysis and Design)

By

Vivek K. Patel

13MCLC12

DEPARTMENT OF CIVIL ENGINEERING

INSTITUTE OF TECHNOLOGY

NIRMA UNIVERSITY

AHMEDABAD-382481

MAY 2015

Declaration

This is to certify that

- The thesis comprises my original work towards the Degree of Master of Technology in Civil Engineering (Computer Aided Structural Analysis And Design) at Nirma University and has not been submitted elsewhere for a degree.
- Due acknowledgement has been made in the text to all other materials used.

Vivek K. Patel

Certificate

This is to certify that the Major Project Report entitled **"Application of Graphics Processing Unit for Parallel Processing in Structural Engineering "** submitted by **Mr.Vivek Patel (Roll No: 13MCLC12)** towards the partial fulfillment of the requirements for the degree of Master of Technology in Civil Engineering (Computer Aided Structural Analysis And Design) of Nirma University is the record of work carried out by him under our supervision and guidance. The work submitted in our opinion reached a level required for being accepted for examination. The results embodied in this major project work to the best of our knowledge have not been submitted to any other University or Institution for award of any degree or diploma.

Dr. P.V.Patel
Guide and Head of Department
Department of Civil Engineering,
Institute of Technology,
Nirma University,
Ahmedabad.

Dr. K. Kotecha	____________________
Director,	Examiner
Institute of Technology,	
Nirma University,	____________________
Ahmedabad.	Date of Examination

Abstract

Multicore machines and hyper-threading technology have enabled scientists and engineers to speed up computationally intensive applications. However, the use of these advanced computing technology requires parallel programming techniques. Solution of linear equation is a computational intensive process in analysis of structural system. With increase in size of problems more linear equations need to be solved which increases execution time of structural analysis dramatically. To overcome this problem parallel programming can be implemented in structural engineering applications.

Objective of this project is to use the concept of parallel programming in Finite Element Analysis of structure using NVIDIA GPU as Hardware. Parallel programming on GPU is carried out using CUDA C language which is based on platform developed by NVIDIA. Unlike CPU, advantage of using GPU is that its architecture allow us to execute many parallel threads slowly, rather than executing a single thread very quickly.

In a few years, many standard software products will be based on concepts of parallel programming. Thus, the need for parallel programming will extend to all areas of software development. The application area for parallel computing will be much larger than scientific computing, which will be main area of parallel computing for many years.

In present study computationally intensive problems of structure engineering are implemented on Graphics Processing Unit(GPU) using concept of parallel computing. For implementation of parallel program on GPU , computational intensive parts of Finite Element Analysis like Matrix multiplication and solution of linear equation are considered. To measure performance of parallel program with respect to sequential program speed up factor is calculated which is ratio of sequential execution time to parallel execution time.

For parallel implementation of Gaussian Elimination solver, linear equation of system representing equilibrium equations of Finite Element Analysis is used. For generation of equation in form of $[A]\{\mathrm{x}\}=\{\mathrm{B}\}$, Finite Element Analysis of Axially loaded bar using 3 node element is considered. Data generated from Finite Element Analysis are always in form of $[K]\{\mathrm{x}\}=\{\mathrm{F}\}$, which is similar to $[A]\{\mathrm{x}\}=\{\mathrm{B}\}$, where K=stiffness matrix, x=displacement vector and F= Force vector. For Solution of displacement vector x, inversion of K matrix is done using Gaussian Elimination method. Sequential program is developed using C language and parallel program is developed using CUDA C language. To compare performance of program, speed up factor is calculated for different number of equation ranging from 100 to 1000.

For parallel implementation of Half Band solver, Finite Element problem used in Gaussian Elimination method is used but in this case a matrix stored in Half Band Form. Data generated from Finite Element analysis are in form of $[K]\{\mathrm{x}\}=\{\mathrm{F}\}$ is converted in to $[A]\{\mathrm{x}\}=\{\mathrm{B}\}$, where A= Half Band stiffness matrix , x= displacement vector and F=Force vector. For solution of displacement vector x, inversion of Half Band matrix is done using Gaussian Elimination method. Sequential program is developed using c language and parallel program is developed using CUDA c language. To compare performance of program speed up factor is calculated for equation ranging from 100 to 10000.

Literature survey shows that parallelization whole Finite Element method rather than focusing equation solver leads to better performance. For parallel implementation Finite Element method, Finite Element analysis of rectangular beam using CST element is developed. Parallel program is developed using CUDA C language. To compare performance of parallel program speed up factor is calculated for number of elements ranging from 10 to 10240.

Acknowledgement

I would like to express my immense gratitude to my guide Dr. Paresh V. Patel,Head of Civil Engineering Department, Institute of Technology, Nirma University, Ahmedabad for his valuable guidance and continual encouragement throughout my major project work. His constant support and interest in subject equipped me with a great understanding of different aspects of the major project work. His extreme supervision and direction right from beginning motivate me to complete this work.

My sincere thanks to Dr. Sharad P. Purohit, Professor, Civil Engineering Department and Dr. Urmil V. Dave, Professor, Civil Engineering Department for their kind suggestions and motivational words throughout the major project work.

A special thanks to Dr K Kotecha, Hon'ble Director, Institute of Technology, Nirma University, Ahmedabad for providing required resources for my project and healthy research environment.

I would like to thank all my friends for their everlasting support and encouragement in all possible ways throughout the major project work.

Most importantly deepest appreciation and thanks to Almighty and my family for their unending love, affection and personal sacrifices during the whole tenure of my study at Nirma University.

Vivek K. Patel

13MCLC12

Abbreviations

CPU	Central Processing Unit
GPU	Graphics Processing Unit
GPGPU	General-Purpose Computing On Graphics Processing Unit
CUDA	Compute Unified Device Architecture
OpenCL	Open Computing Language
ALU	Arithmetic Logic Unit
OpenGL	Open Graphics Library
FEM	Finite Element Method
MPP	Massively parallel processors

Contents

List of Figures

List of Tables

Chapter 1

Introduction to parallel computing

1.1 General

In the past two decades, the development of algorithms for structural engineering applications has received a boost due to the advent of parallel computers. Considerable research is being done in order to rewrite algorithm originally designed to run on sequential machines as well as to develop new methods that take advantage of parallelism offered by the multicore processing computers.

In this project comparison of different parallel algorithms are studied to find out best method for parallel computation. More Focus is made on solving finite element problems rather than concentrating on equation solver, which will lead to higher performance.

1.2 Introduction

Parallel computing is a form of computation in which many calculations are carried out simultaneously. Operating on Principal that large Problems can often be divided into smaller ones, which then solved in parallel. There are several forms of parallel computing: bit level, instruction level, Data and task parallelism[3]

Parallelism has been employed for many years, mainly in high performance computing but interest in it has grown lately due to physical constraints preventing frequency scaling. As Power consumption and heat generation by computer became main concern in recent years parallel computing has become dominant in computer architecture mainly in form of multi-core processor.[3]

Figure 1.1: IBM Blue Gene

Figure 1.1 shows a picture of IBN Blue Gene super computer. Blue Gene supercomputer use large number of low frequency processors to achieve better performance to energy ratio. Blue Gene is Fastest computer in world from November,2004 to November 2007 as per Top 500 organisation.[4]

Parallel computers can be roughly classified according to the level at which the hardware supports parallelism, With multi-core and multiprocessor computers having multiple processing element within a single machine ,while Clusters, MPPs and grids use multiple computes to work on the same task. Specialized parallel computer architectures are sometimes used alongside traditional processors, for accelerating specific task.[3]

Parallel computer programs are more difficult to write than sequential ones, because parallelization introduce several new classes of potential software bugs. Communication and synchronization between the different subtasks are typically some of the greatest obstacles to getting good parallel performance.[3]

1.3 Background of Parallel Processing

Traditionally, computer software has been written for serial computation. To solve a problem, an algorithm is constructed and implemented as a serial stream of instructions. These instructions are executed on a central processing unit(CPU) on one computer. Only one instruction may execute at a time, After one instruction is finished next instruction is executed.[3]

Parallel computing on the other hand uses multiple processing elements simultaneously to solve a problem. This is accomplished by breaking the problem into independent parts so that each processing element can execute its part of algorithm simultaneously with the others. The Processing elements can be diverse and include resources such as a single computer with multiple processors, several networked computers, specialized hardware or any combination of the above.[3]

Frequency scaling was the dominant reason for improvements in computer performance from the mid-1980s until 2004. The runtime of a program is equal to the number of instructions multiplied by the average time per instruction. Maintaining everything else constant, increasing the clock frequency decreases the average time it takes to execute an instruction. Thus an increase in frequency decreases runtime for all compute bound programs.[3]

However, Power consumption by a chip is given by the equation $P=C\times V^2\times F$, Where P is power, C is the capacitance being switched per clock cycle which is proportional to the number of transistors, V is on Voltage and F is the processor frequency. Increases in frequency increase the amount of power used in processor. Increasing processor power consumption led to Intels May 2004 cancellation of its Tejas and Jayhawk processors, which is generally cited as the end of frequency scaling.[3]

Moores Law is empirical observation that transistor density in a microprocessor doubles every 18 to 24 months. Despite power consumption issue and repeated predictions of its end, Moores law is still in effect. With the end of frequency scaling, these additional transistor can be used to add extra hardware for parallel computing.[3]

1.3.1 Amdahls law

Optimally, the speed up from parallelization would be linear means doubling the number of processing elements should halve the runtime and doubling it a second time should again halve the runtime.[1]

However, very few parallel algorithms achieve optimal speed-up. Most of them have a near liner speed up for small number of processing elements, which flattens out into a constant value for large number of processing elements.[1]

The potential speed up of an algorithm on a parallel computing platform is given by Amdahls law, originally formulated by Gene Amdahl in the 1960s. It states that a small portion of the program which cannot be parallelized will limit the overall speed up available from parallelization.[1]

A program solving a large mathematical or engineering problem will typically consist of several parallelizable parts and several non-parallelizable (sequential) parts. If α is the fraction of running time a program spends on non-parallelizable parts, then:

$$\lim_{P\to\infty} \frac{1}{\frac{1-\alpha}{\alpha} + \alpha} = \frac{1}{\alpha} \tag{1.1}$$

$\frac{1}{\alpha}$ is the maximum speed up with parallelization of the program, with P being the number of processors used. If the sequential portion of program accounts for 10% of the runtime (α= 0.1) , we can get no more than a 10× speed-up, regardless of how many processors are added. This puts an upper limit on the usefulness of adding more parallel execution units. When a task cannot be partitioned because of sequential constrains, the application of more effort has no effect on the schedule.[1]

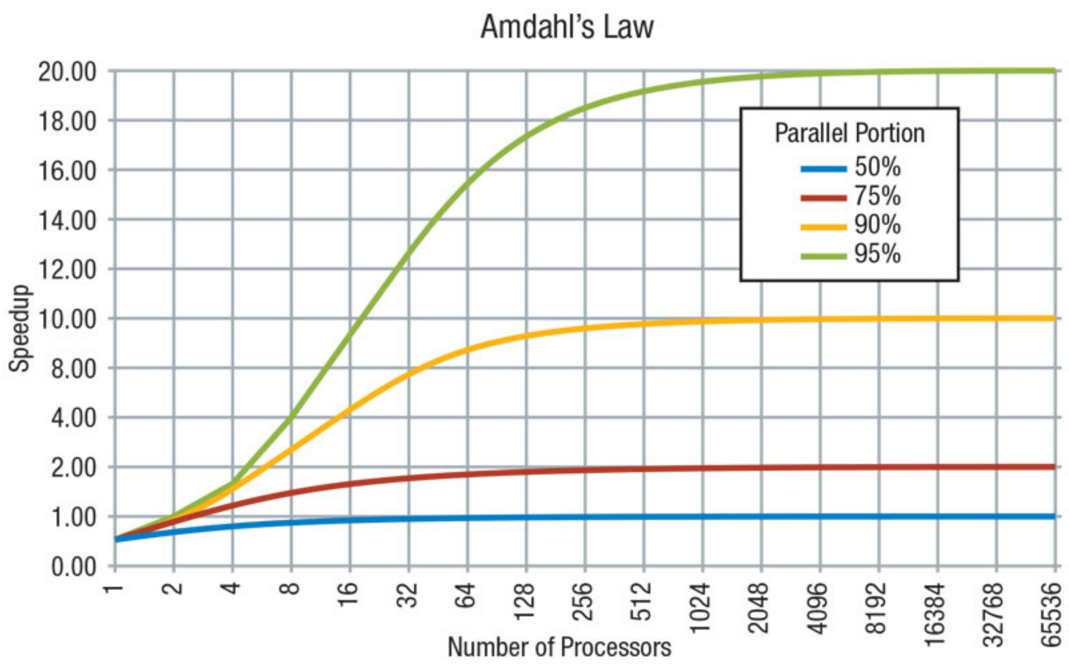


Figure 1.2: Amdahl's law

Figure 1.2 state a graphical representation of Amdahls law. The speed-up of a program from parallelization is limited by how much of the program can be parallelized. For example, if 90% of the program can be parallelized, the theoretical maximum speed-up using parallel computing would be 10× no matter how many processor are used.

Amdahls law assume that the running time of the sequential portion of the program is independent of the number of processor.

1.4 Types of Parallelization

1.4.1 Bit-Level parallelism

From the advent of very large scale integration computer-chip fabrication technology in the 1970s until about 1986, speed up in computer architecture was driven by doubling computer word size the amount of information the processor can manipulate per cycle. Increasing the word size reduces the number of instructions the processor must execute to perform an operation on variable whose sizes are greater than the length of the word.[5]

For example, where an 8-bit processor has to add two 16-bit integers, the processor must first add the 8 lower-order bits from each integration using the standard addition instruction, then add the 8 higher order bits using an add with carry instruction and the carry bit from the lower order addition. Thus, an 8-bit processor requires two instruction to complete a single operation, where a 16-bit processor would be able to complete the operation with a single instruction.[5]

Historically, 4-bit microprocessor were replaced with 8-bit, then 32-bit microprocessors. This trend generally came to an end with the introduction of 32-bit processors, which has been a standard in general purpose computing for two decades. Not until recently with the advent of X86-64 architectures, 64 bit processors become common place.[5]

1.4.2 Instruction level parallelism

A computer program, is in essence, a stream of instructions executed by a processors. These instructions can be re-ordered and combined into groups which are then executed in parallel without changing the result of the program. This is known as instruction-level parallelism. Advances in instruction-level parallelism dominated computer architecture from the mid 1980 until the mid-1990.[5]

1.4.3 Task parallelism

Task parallelism is the characteristic of a parallel program that entirely different calculations can be performed on either the same or different sets of data. This contrasts with data parallelism, where the same calculation is performed on the same or different sets of data. Task parallelism involves the decomposition of task into subtasks and then allocating each sub-task to processor for execution. The processors would then execute these sub-task simultaneously and often cooperatively. Task parallelism does not usually scale with the size of a problem.[5]

1.5 Hardware

1.5.1 Memory and communication

Main memory in the parallel computer is either shared memory (shared between all processing elements in a single address space), or distributed memory (in which each processing element has its own local address space). Distributed memory refers to the fact that the memory is logically distributed, but often implies that it is physically distributed as well. Distributed shared memory and memory virtualization combine the two approaches, where the processing element has its own local memory and access to the memory on non-local processors. Access to local memory are typically faster than access to non-local memory.[3]

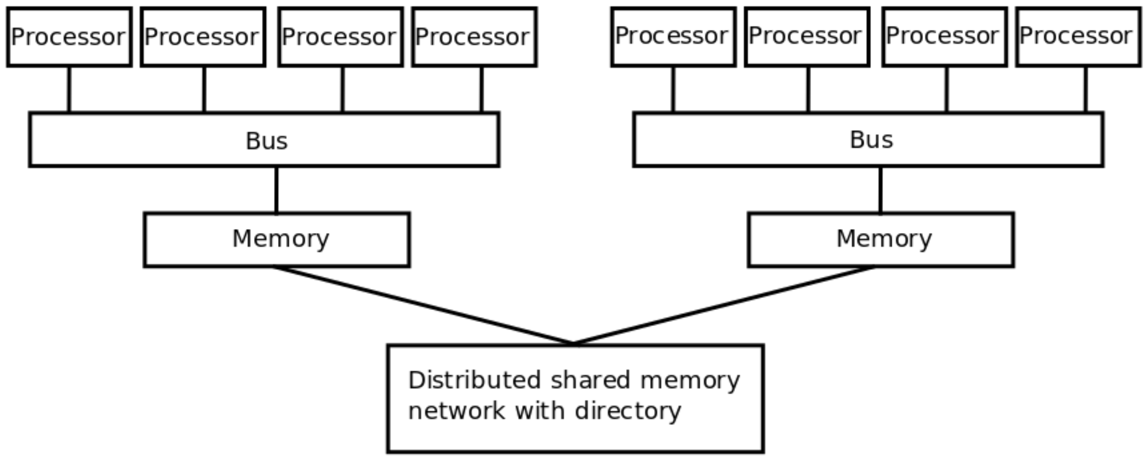


Figure 1.3: Distributed shared memory network

Computer architecture in which each element of main memory can be accessed with equal latency and bandwidth are known as Uniform Memory Access (UMA) systems. Typically, that can be achieved only by a shared memory system, in which the memory is not physically distributed. A system that does not have this property is known as Non-Uniform Memory Access (NUMA) architecture. Distributed memory systems have non-uniform memory access.[3]

Computer system make use of caches-small, fast memories located close to the processor which store temporary copies of memory values (in both physical and logical sense). Parallel computer systems have difficulties with catches that may store the same value in more than one location, with the possibility of incorrect program execution.[3]

Figure 1.3 state the logical view of Non-Uniform Memory Access (NUMA) architecture. Processor in one directory can access the directory's memory with less latency than they can access memory in other directory's memory.[3]

Processor-Processor and Processor-memory communication can be implemented in hardware in several ways, including via shared (multiported or multiplexed) memory, a crossbar switch, a shared bus or an interconnect network of a myriad of topologies including star, ring, tree, hypercube or n-dimensional mesh.[3]

Parallel computers based on interconnect networks need to have some kind of routing to enable the passing of messages between nodes that are not directly connected. The medium used for communication between the processors is likely to be hierarchical in large multiprocessor machines.[3]

1.5.2 Classes of parallel computers

Parallel computers can be roughly classified according to the level at which the hardware supports parallelism. This classification is broadly analogous to the distance between basic computing nodes. These are not mutually exclusive. For example, cluster of symmetric multiprocessors are relatively common.

Multicore computing

A multicore processor is a processor that includes multiple execution units (”cores”) on the same chip. These processors differ from superscalar processors, which can issue multiple instructions per cycle from one instruction stream (thread). In contrast, a multicore processor can issue multiple instructions per cycle from multiple instruction streams. IBM’s Cell microprocessor, designed for use in the Sony PlayStation 3, is another prominent multicore processor.[2]

Each core in a multicore processor can potentially be superscalar as well that is, on every cycle, each core can issue multiple instructions from one instruction stream. Simultaneous multithreading (of which Intel’s HyperThreading is the best known) was an early form of pseudo-multicoreism. A processor capable of simultaneous multithreading has only one execution unit (”core”), but when that execution unit is idling (such as during a cache miss), it uses that execution unit to process a second thread.[2]

Symmetric multiprocessing

A symmetric multiprocessor (SMP) is a computer system with multiple identical processors that share memory and connect via a bus. Bus contention prevents bus architectures from scaling. As a result, SMPs generally do not comprise more than 32 processors. "Because of the small size of the processors and the significant reduction in the requirements for bus bandwidth achieved by large caches, such symmetric multiprocessors are extremely cost-effective, provided that a sufficient amount of memory bandwidth exists."[2]

Distributed computing

A distributed computer (also known as a distributed memory multiprocessor) is a distributed memory computer system in which the processing elements are connected by a network. Distributed computers are highly scalable.[2]

Cluster computing

A cluster is a group of loosely coupled computers that work together closely, so that in some respects they can be regarded as a single computer. Clusters are composed of multiple standalone machines connected by a network. While machines in a cluster do not have to be symmetric, load balancing is more difficult if they are not. The most common type of cluster is the Beowulf cluster, which is a cluster implemented on multiple identical commercial off-the-shelf computers connected with a TCP/IP Ethernet local area network. Beowulf technology was originally developed by Thomas Sterling and Donald Becker is shown in Figure 1.4. The vast majority of the TOP500 supercomputers are clusters.[2]

Figure 1.4: Cluster computing

Massive parallel processing

A massively parallel processor (MPP) is a single computer with many networked processors. MPPs have many of the same characteristics as clusters, but MPPs have specialized interconnect networks (whereas clusters use commodity hardware for networking). MPPs also tend to be larger than clusters, typically having ”far more” than 100 processors.[2]

In a MPP, “ Each CPU contains its own memory and copy of the operating system and application. Each subsystem communicates with the others via a high-speed interconnect.”

Blue Gene shown in Figure1.5 is the fifth fastest supercomputer in the world according to the June 2009 TOP 500 ranking. Blue Gene has massively parallel processor.

Figure 1.5: Blue gene

Grid computing

Grid computing is the most distributed form of parallel computing. It makes use of computers communicating over the Internet to work on a given problem. Because of the low bandwidth and extremely high latency available on the Internet, distributed computing typically deals only with embarrassingly parallel problems.[2]

Most grid computing applications use middleware, software that sits between the operating system and the application to manage network resources and standardize the software interface. The most common distributed computing middleware is the Berkeley Open Infrastructure for Network Computing (BOINC). Often, distributed computing software makes use of ”spare cycles”, performing computations at times when a computer is idling.[2]

Specialized parallel computers

Within parallel computing, there are specialized parallel device that remain niche areas of interest. While not domain-specific, they tend to applicable to only a few classes of parallel problems.[2]

Reconfigurable computing with field-programmable gate arrays

Reconfigurable computing is the use of a field-programmable gate array (FPGA) as a co-processor to a general purpose computer. An FPGA is, in essence, a computer chip that can rewire itself for a given task.[2]

FPGAs can be programmed with hardware description languages such as VHDL or Verilog. However, programming in these languages can be tedious. Several vendors have created C to HDL languages that attempt to emulate the syntax and semantics of the C programming language, with which most programmers are familiar. The best known C to HDL languages are Mitrion-C, Impulse C, DIME-C, and Handel-C. Specific subsets of SystemC based on C++ can also be used for this purpose.[2]

AMD’s decision to open its HyperTransport technology to third-party vendors has become the enabling technology for high-performance reconfigurable computing.

General-purpose computing on graphics processing units (GPGPU)

General-purpose computing on graphics processing units (GPGPU) is a fairly recent trend in computer engineering research. GPUs are co-processors that have been heavily optimized for computer graphics processing. Computer graphics processing is a field dominated by data parallel operations particularly linear algebra matrix operations.[2]

In the early days, GPGPU programs used the normal graphics APIs for executing programs. However, several new programming languages and platforms have been built to do general purpose computation on GPUs with both Nvidia and AMD releasing programming environments with CUDA and Stream SDK respectively.[2]

Other GPU programming languages include BrookGPU, PeakStream, and RapidMind. Nvidia has also released specific products for computation in their Tesla series. The technology consortium Khronos Group has released the OpenCL specification, which is a framework for writing programs that execute across platforms consisting of CPUs and GPUs. AMD, Apple, Intel, NVIDIA and others are supporting OpenCL.[2]

Figure 1.6 shows picture of NVIDIA GPU used in desktop computers and Figure 1.7 shows inside view of GPU. Figure 1.8 is GPU used in laptop computers.

Vector processors

A vector processor is a CPU or computer system that can execute the same instruction on large sets of data. Vector processors have high-level operations that work on linear arrays of numbers or vectors. An example, vector operation is $A = B \times C$, where A, B, and C are each 64-element vectors of 64-bit floating-point numbers. Cray computers became famous for their vector-processing computers in the 1970s and 1980s. However, vector processors both as CPUs and as full computer systems

Figure 1.6: NVIDIA Tesla GPU card

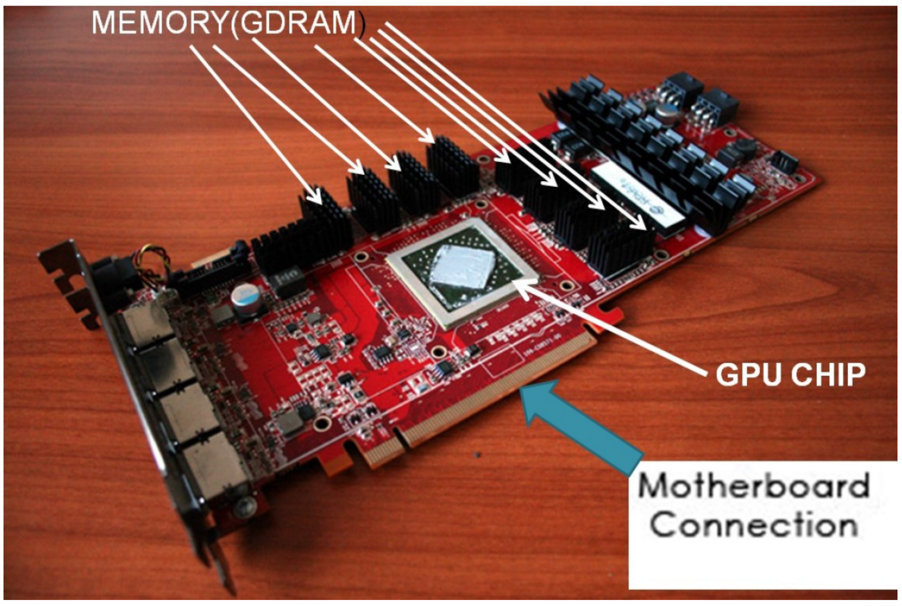


Figure 1.7: Inside View of GPU

Figure 1.8: GPU in Laptop

have generally disappeared. Modern processor instruction sets do include some vector processing instructions, such as with AltiVec and Streaming SIM D Extensions (SSE).[2]

1.6 Objective of study

The major objectives of present study are:

- To understand CUDA C and it's implementation for parallel programming in structural engineering applications.
- To study various Numerical methods available for Parallel Programming.
- To understand effect of parallelization in Structure engineering applications like Finite element analysis problems where massive computation is required.
- Compare performance of different parallel algorithm to solve structure analysis problems.

1.7 Scope of Work

In order to achieve above objective the scope of work for major project is decided as follow.

- Understanding Fundamentals of Parallel Computing.
- Understanding CUDA C and its Specification.
- Study of Various Parallel Processing technique for Structure Engineering applications.
- Development of Computer program for solving equations based on Gaussian Elimination and Half Band Solver.
- Development of parallel computer program for Finite Element Analysis and its implementation on GPU.

1.8 Organization of Report

The study carried out in this major project is related to the application of parallel processing in structural engineering. The content of major project is divided into different chapter as follows.

Chapter 1 include Introduction to parallel computing as well as background of Parallel Processing. It also covers Types of parallelization available and different types of hardware available for Parallel Processing.It includes objective of study and scope of work .

Chapter 2 covers literature review which is divided into four different parts. It includes Paper based on Introduction of parallel computing, Comparison between CUDA C and OpenCL, Algorithm for parallel processing and Application of parallel processing.

Chapter 3 covers different aspects of GPU computing which include brief history of GPU computing, various terminology used in GPU computing , CUDA language specification and CUDA architecture.This chapter also covers list of application acceleration by GPU computing and Software tools available for GPU Programming. At the end application of parallel programming is explained with example of Matrix Multiplication.

Chapter 4 introduce Gaussian Elimination method to solve linear system of equation in form of $[A]$\{x\}=\{B\} using concept of parallel programming on Graphics processing Unit. parallel program is developed using CUDA C language and compared with sequential program developed using C language. Comparison of parallel program and sequential program is done based on execution time and speed up factor.For generation of equations, Finite element analysis of axially loaded bar using 3 node element is adopted.

Chapter 5 introduce Half Band solver based on gaussian elimination method. Parallel program is developed using CUDA C language and compared with sequential program developed using C language. Comparison of parallel program and sequential program is done based on Execution time and speed up factor. For generation of equations, Finite element analysis of axially loaded bar using 3 node element is used and all equations are converted into Half Band form.

Chapter 6 explain parallelization of whole Finite Element program instead of concentrating on equation solver. Finite Element Analysis of Cantilever beam having point load at end is carried out using CST element. Parallel program is developed using CUDA C language and compared with sequential program developed using C language. Comparison of parallel program and sequential program is done based on Execution time and speed up factor.

Chapter 7 contains detailed summary of project, concluding remarks and Future scope of work.

Chapter 2

Literature Survey

2.1 General

In this chapter literature related to various aspects of parallel programming and its applications in structural engineering field are reviewed.

2.2 Introduction of parallel computing

Sotelino[22] presented some of the parallel algorithms that have been developed for parallel computing. More specifically, it was a survey of parallel algorithms applicable for structural engineering. Such algorithms included parallel solvers, techniques for the parallelization of the finite element method. There was good discussion on research in the development of concurrent algorithms for parallel architectures. The discussion in the section of Iterative solver and direct solvers were concerned with the solution of a system of linear algebraic equations. In this work, an attempt was made to provide a thorough survey of the methods that were directly related to structural engineering applications.

2.3 Comparison between CUDA and OpenCL

Karimi et al.[13]presented comparison between CUDA and OpenCL Platform. CUDA and OpenCL offer two different interfaces for programming GPUs. As per this paper, OpenCL is an open standard that can be used to program CPUs, GPUs, and other devices from different vendors, while CUDA is specific to NVIDIA GPUs. Although OpenCL promises a portable language for GPU programming, its generality may entail a performance penalty. In this paper Comparison of performance of CUDA and OpenCL was done using complex, near-identical kernels. In tests, CUDA performed better when transferring data to and from the GPU. CUDA kernel execution was also faster than OpenCL. CUDA seems to be a better choice for application where high performance is important otherwise the choice between CUDA and OpenCL was made based on familiarity with system, available development tools for target hardware.

2.4 Algorithm for parallel Processing

Sharma et al.[10] presented Gauss Jordan algorith for matrix inversion on a CUDA platform to exploit the large scale parallelization feature of a massively multithreaded GPU. The algorithm was tested for various types of matrices (Sparse, band ,identity) and the performance were studied and compared with CPU based parallel methods. Matrix size of 64×64, 128×128, 256×256, 512×512, 1024×1024, 1536×1536 and 2048×2048 were used for measurement of execution time. All matrix sizes were tested for different types of matrix like sparse, identity and band Total speed up factor for 2048 was between 20 to 25. Authors found GPU based parallelization much faster than CPU based parallelization.

Kruzel and Banas[15] presented work on computational aspects of the problem of numerical integration in finite element calculations and considered an openCL implementation of related algorithms. As a platform for testing the implementation they chose the PowerXCell processors. Although the processor was considered old for today standard , they investigated it's performance due to two features : Wide vector units and relatively slow connection of computing cores with main global memory. The performed analysis of parallelization options could also be used for designing numerical integration algorithms for other processors with vector registers. They considered higher order finite element approximations and implemented the standard algorithm of numerical integration for prismatic element.

The performance results presented for finite element numerical integration algorithm running on the powerXCell processor proved that the algorithm could be successfully ported to multi-core processor with manually managed memory hierarchy and vector execution units.

Yang et al.[8] tested Cholesky decomposition on GPU and FPGAs. Cholesky decomposition has been widely utilized for positive symmetric matrix factorization in solving least square problems. Various parallel accelerators including GPUs and FPGAs had been explored to improve performance. In this paper, Cholesky decomposition was implented on both FPGAs and GPUs by designing a dedicated architecture for FPGAs and exploiting massively parallel computation for GPUs. Performance of the cholesky decomposition on GPUs, CPUs, FPGAs and hybrid systems were compared in both single and double precision. Result showed that the FPGA implementation had better efficiency with respect to clock cycles compared with our pure GPU implementation.

Hsieh et al.[21] presented general sparse matrix and parallel computing technologies for finite element solution of large scale structural problems in a PC cluster environment. The general sparse matrix technique was first employed to reduce execution time and storage requirement for solving the simultaneous equilibrium equations in finite element analysis . To further reduce the time required for large scale structural analysis , two parallel processing approaches for sharing computational workloads among collaborating processors were then investigated. One approach adopted a publicly available parallel equation solver, called SPOOLES, to directly solve the sparse finite element equations while other employed a parallel substructure method for the finite element solution. This work focused more on integrating the general sparse matrix technique and the parallel substructure method for large scale finite element solutions. Additionally, Numerical studies had been conducted on several large scale structural analysis using a PC cluster to investigate the effectiveness of the general sparse matrix and parallel computing technologies in reducing time and storage requirement in large scale finite element analysis.

2.5 Application of parallel processing

Patel [17] presented work related to introduction of High performance computing to Structural engineering. Main focus was solution of structural engineering related problem using OpenCL for parallel programming. He used OpenCL as parallel programming platform and C++as programming language.For experiment two different solver was used for solution of equation. First was Gaussian elimination method, in which execution time measured for number of equations from range 100 to 10000. Another method was Half band in which displacement for 2D plane frame was calculated for different matrix size. structure size (No of bay*Number of story) of 50×50, 100×100, 150×150, 200×200, and 250×250 were solved. Program was tested on different CPU and GPU. Speed Up factor for both method and all different CPU and GPU was calculated. For Gaussian Elimination max Speed Up was 1702 for i7-2630QM processor and for 10000 no of equation. For Half Band method max speed up was 3.9 for i7-3450 for 250×250 size of matrix.

Wang et al.[16] presented work on the GPU parallelization of complex three-dimensional software for nonlinear analysis of concrete structures. It focused on coupled thermo-mechanical analysis of complex structures As the modeling of a large structure by means of FEM/DEM may lead to prohibitive computation times, a parallelization strategy was required. In this paper comparative study between the GPU and CPU computation results was presented, and the runtimes and speedups were analyzed. The results showed that dramatic performance improvements gained from GPU parallelization. One example was given to demonstrate GPU implementation with CUDA in which program was made to run on multiple GPU. In main experiment program to calculate displacement due to stress was given, in which program was tested on 3 different GPU and compared with CPU. Comparison was done for execution time , generation of stiffness matrix and also for copy output from GPU to CPU(communication time). Errors in results compared to CPU results were also calculated in percentage and compared. In this study, the CDEM(Coupled Finite/Discrete Element Method) was successfully accelerated by using GPUs. Detailed tests on accuracy, runtime, and speedup were performed on different GPUs. Authors concluded Maximum and minimum speed up observed was 417 and 102.

Dziekonski et al.[6] presented an efficient technique for fast generation of sparse systems of linear equations. The proposed approach employed a graphics processing unit (GPU) for both numerical integration and matrix assembly. The performance results obtained on a test platform consisting of a GPU (1x Tesla C2075-448 core) and a CPU (2x twelve-core Opterons), indicated that the GPU implementation of the matrix generation allows one to achieve speedups by a factor of 81. Speed up factor for numerical integration for tesla vs opteron was 77.88. The obtain performance results indicated that the proposed GPU accelerated implementation allow significant reduction in matrix generation time.

Hajjar and Abel[11] presented work regarding Parallel Processing Of Nonlinear Dynamic Analysis Of Steel Frame Structures Using Domain Decomposition In this paper analysis of three-dimensional framed structure subjected to seismic loading using parallel processing. Non linearity requires frequent updating of stiffness matrix. In this paper, loading was multicomponent, non proportional and time varying and time span of the seismic loading may be on the order of 10 to 100 times the fundamental period of the structure. Therefore thousands of the steps must be run to model properly the nonlinear behavior. Domain decomposition method was adopted in this research for parallel processing. This paper concluded that the performance of Domain Decomposition improve considerably with increase in size of problems.

Kandasamy[14] dealt with a research concept of parallel finite element (FE) simulation for moving boundary and adaptive refinement problems using graphics processing unit (GPU).The main concern in this study was to improve the numerical performance of continuous FE simulation using recent data-parallel computing technology (GPU-CUDA).The computational time for existing simulations was very long using conventional parallel computing technique (MPI). This short-coming could be overcomed using data parallel computing power of CPU and GPU by increasing the overall performance of FE simulation. By adapting the computing power of graphic processors for multi-threaded fine-grain parallelization for FE assembly and solving, overall performance could be significantly improved.

In this paper Numerical Simulation of Tunnel boring process was carried out using CUDA programming. Parallelization of FE simulation was done using domain decomposition technique. The whole model was subdivided into many subdomains and each part solved by different processors as distributed or shared memory. To represent the reality of real time tunnel boring process numerical computation should be fast enough to update the current state of boring process. This type of large scale FEM modeling handles huge domain with many hundred thousand DOFs.

In continuous simulations, each and every time step solution was updated for the next step calculation. When the problem size increased, required solution time in each step dramatically increased. To meet the fast solution, parallel computing techniques were applied. The major part of the research was the application of GPU parallelization in whole part of FE simulation of tunnel boring process.The ultimate goal of this research was to make GPUs system capable to compute custom application with minimum modifications of programs.

Bahcecioglu and Kurc[7] presented work related to decrease the analysis time for nonlinear dynamic analysis of large scale structural models utilizing the GPUs.In the implementation, explicit version of the Newmark family of algorithms was utilized. This type of algorithm enabled the computations to be applied on each finite element, eliminating the need for global matrix assembly. Two different GPU implementations were tested. In the first approach, creation of elemental matrices and computation of the explicit Newmark algorithm were separated into two different kernels. The second approach fused these two kernels at compile time into a single kernel code. Both implementations were developed using CUDA language. Implementation details of both algorithms were discussed in detail noticing optimization differences. Both GPU implementations were tested and compared with a CPU implementation using models with varying sizes.

For testing both implementation various model sizes from 10,000 to 10,00,000 were analyzed and compared to a CPU implementation. For each model size, a two dimensional and a three dimensional model was constructed composed of nonlinear quadrilateral and nonlinear hexahedron elements respectively. Models were constructed in order to analyze the performance characteristics of implementations. Two dimensional models were in square shape and three dimensional models were in cubic shape. All models were loaded dynamically with prescribed displacements that represent east-west component from Treasure Island record of Loma Prieta Earthquake. Two dimensional models were analyzed for 100 time steps and three dimensional models were analyzed for 20 time steps for comparison purposes.

Both implementations resulted in similar performance characteristics that outperformed the CPU implementation in two dimensional models. In three dimensional models CPU implementation had enough computation to suppress communication costs and outperformed GPU implementations.

Qian et al.[26] carried out research of parallel computing for Large scale Finite-Element Model of WheelRail Rolling Contract. For the increasing requirement of calculation scale and computing accuracy, the parallel computing method and parallel computing environment became an effective way to solve this problem. The parallel computing method of contact problem was analyzed firstly. Then the contact algorithms and parallel computing of ABAQUS was introduced and parallel computing environment using MPI in ABAQUS is put forward. On the basis of cluster, some different finite element model was solved by implicit and explicit solution. It is found that mesh size of wheel/rail contact field was refined to 0.75mm in order to ensure accuaracy for engineering. At last, the parallel computing for the contact problem of wheel/rail was discussed using the speedup and efficiency.

Fan et al.[27] presented application of parallel computing in large Eigenvalue problems for engineering structures. A parallel solving system was constructed via integrating these software packages into the finite element parallel computing framework-PANDA. The finite element model of engineering structures was built in pre-processing software-MSC.Patran. Based on interface between PANDA and MSC.Patran, the model information was translated in PANDA to generate stiffness and mass matrices in a parallel way. Utilizing these matrices, a large scale parallel computing of eigenvalues was carried out via calling software packages in PANDA. The numerical results show that PANDA frame was competent for carrying out large scale parallel computing of eigenvalue problems in virtue of supercomputer, the computing scale attains millions degree of freedom and the parallel efficiency was favorable. They gave a brief review on some dominant algorithm and freely available software for numerical solution of large sparse eigenvalue problems. There was also description of whole processes of parallel computing for eigenvalue problems arising from engineering structures. In the analysis example solved, the number of degree of freedom of the finite element model was about 2.3 mllions.

Fu[9] presented parallel finite element method using domain decomposition technique which was implemented on a distributed parallel environment of workstation cluster. The algorithm was presented for solving the conjugate gradient method on a parallel platform with element based domain decomposition. Using the developed code, structural analysis of a dam was solved on workstation cluster and results were presented. The parallel performance was analysed,On the basis of mode synthesis analysis, parallel algorithm of solving large scale structural eigen problem was presented by FU. The numerical results showed that this parallel algorithm was effective for large scale structure eigen problem. Parallel computing for numerical example of structural modal analysis was performed on DELL workstation cluster in school of computer engineering and science, Shanghai University. It was a cluster with 8 processor arranged in 4 dual-processor nodes with 2.4GHz Intel Xeon chips (512KB cache) and 1GB of memory per node. These nodes were connected with 100Mbps Ethernet interconnect.

Leow et al.[28] presented parallel implementation of a direct method for solving linear equations called Gaussian Elimination.The solution of a linear system of equations constitute an important part in the field of linear algebra that is widely used in industries like aerospace, aeronautics, solid mechanics, fluid dynamics, oil research and numerous others. Through evaluations had been performed for variants of implementation that exploit different memory features on an NVIDIA Tesla C1060 GPU. Compared to a serial implementation on an Intel Core i7, the execution time for forward elimination on the GPU was reduce by a factor of 183 when using both global and shared memory systems, and by a factor of 185 when using only global memory. The maximum size of matrix considered for study was 8182 $\times$ 8192.

Reddy et al.[18]described the design and the implementation of parallel routines in

Heterogeneous SCALAPACK library that solve a dense system of linear equations. It was discussed that the efficiency of these parallel routines was due to the most important feature of the library, which was the automation of the difficult optimization task of parallel programming on heterogeneous computing cluster. They showed that the efficiency of these parallel routines was due to the most important feature of library , which was the automation of the difficult optimization task of parallel programming on heterogeneous computing cluster, Other features were the determination of the accurate values of the platform parameters such as speed of the processor and the latencies and bandwidth of the communication links connecting different pairs of processors, the optimal values of the algorithmic parameters such as the total number of processes, the 2D process grid arrangement and the efficient mapping of the processes executing the parallel algorithm to the executing nodes of the heterogeneous computing cluster.

Stefanski et al.[23]evaluated the usability and performance of open computing Language (OpenCL) targeted for implementation of the Finite-Difference Time-Domain (FDTD) method. The simulation speed was compared to implementation based on alternative techniques of parallel processor programming. Moreover, the portability of OpenCL FDTD code between modern computing architectures was assessed. The average speed of OpenCL FDTD simulations on a GPU was about 1.1 times lower than a comparable CUDA based solver for domains with sizes varying from 503 to 4003 cells. Although OpenCL code dedicated to GPU can be executed on multi-core CPUs, a direct porting did not provide satisfactory performance due to an application of architecture specific feature in GPU code. Therefore, the OpenCL kernels of the developed FDTD code were optimized for multi-core CPUs. However, this improved OpenCL FDTD code was still about 1.5 to 2.5 times slower than the FDTD solver developed in the OpenMP parallel programming standard. The study concluded that, despite current performance drawbacks, the future potential of OpenCL was significant due to its flexibility and portability to various architectures.

Wang et al.[25] used Gaussian Elimination and Gauss-Jordan methods because of their extensive use in finite element applications. In most cases, dense, nonsymmetric, real systems were solved but similar methods for banded and complex system were presented.Parallel solution algorithms based on the Gaussian Elimmination and Gauss-Jordan were implemented and compared. These parallel solvers were applied to large, dense or banded systems of equation arising from finite element analysis of 2-D and 3-D electromagnetic field problems. Both real and complex matrices were considered with emphasis on very large systems. The speed up obtained by parallelization on the MPP compared to sequential computers was almost three order of magnitude.

In engineering applications it is often necessary to solve large systems of equations that were either too large or require too much computer resources to be economically feasible on standard computers. For this type of problem a parallel machine was very attractive. The type of systems considered were those arising from the application of the Finite element method(FEM) to engineering applications. The FEM was particularly computationally intensive , yet its various parts were either intrinsically parallel or could be parallelized. By using a parallel processor, considerably faster solution could be achieved or, alternatively , large problems could be solved.

Mani et al.[24] proposed a parallel Gaussian elimination technique for the solution of liner equations. They considered the direct solution of $[A]\{x\} = \{C\}$, where A is a banded matrix with half bandwidth b. They modeled the situation as a acyclic directed graph. In this graph, the nodes represented arithmetic operation applied to the elements of A and the arcs represents the precedence relation that exists among the operations in the solution process. This graph gives clear picture to user in identifying the operations that can be done in parallel. this graph was also useful in scheduling operations to the processors. The absolute minimum completion time and the lower bound on the minimum number f processors required to solve the equations in minimum time can be found from it. Speedup approached a limit using parallel processors , set by the absolute minimum time was also brought out from this graph.

McGinn et al.[20] presented parallel algorithm for gaussian elimination. Elimination in both a shared memory environment, using OpenMP, and in a distributed memory environment, using MPI. Parallel LU and gaussian algorithms for linear systems had been studied extensively and the paper presented the results of examining various load balancing schemes on both platforms. It was noted that the impact on performance occur as one changes the size of Matrix i.e. When increase in Value of Number of equations the MPI displays an improvement in performance as oppose to the openMP program where performance increase seems to diminish. It is possible that as n increases one may find a point where the distributed environment would show a greater increase in performance than the shared platform.

Liu et al.[12] designed and developed a GPU based Bi-Conjugate Gradient STABilized solver that need both generality and stability requirements. It was well suited for all types of banded linear systems and this solver combined with new matrix deposition method with several optimization for inter-GPU and inter-machine communications to achieve good scalability on large scale GPU clusters. Solving a banded linear system efficiently is important to many scientific and engineering applications. Current solvers achieve good scalability only on linear systems that can be partitioned into independent subsystems. They designed a number of GPU and MPI optimization to speed up inter GPU- and intermachine communications and evaluated the solver on Poisson equation and advection diffusion equation as well as several other banded systems. The solver achieved a speedup of more than 21 times running from 6 to 192 GPUs on the XSEDE'S Keeneland supercomputer and because of small communication overload, could scale up to 32 with GPUs on Amazon EC2 with relatively slow ethernet network.

Zhang et al.[29] presented a GPU based parallel Jacobian iterative solver for dense linear equations. Modern GPUs are high performance as many core processors fit for large scale parallel computing. They provided a novel way for accelerating the solution process. First, they introduced background for accelerating linear equations solver together with GPUs and the corresponding parallel platform CUDA. They compared experimental results of CUDA program with traditional programs on CPU. Experiments showed that it obtained speed up of approximately 59 times with single floating point at low precision, 19 times with double at high precision.

2.6 Sumary

In this chapter literature on Application of parallel computing,Algorithms for parallel Programming as well as performance comparison between CUDA and OpenCL are discussed briefly.It gives an overview of the work carried out by various researchers in different fields of structural engineering.

Chapter 3

Introduction to GPU Computing

3.1 General

Multicore machines and hyper-threading technology have enabled scientists, engineers, and financial analysts to speed up computationally intensive applications in a variety of disciplines. Today, another type of hardware promises even higher computational performance: the graphics processing unit (GPU).

Originally used to accelerate graphics, GPUs are increasingly applied to scientific calculations. Unlike a traditional CPU, which includes no more than a handful of cores, a GPU has a massively parallel array of integer and floating-point processors, as well as dedicated, high-speed memory. A typical GPU comprises hundreds of these smaller processors.

3.2 Background of GPU Computing

In recent years, much has been made of the computing industry's widespread shift to parallel computing. Nearly all consumer computers in the year 2015 ship with multicore central processors. From the introduction of dual-core, low-end netbook machines to 8 and 16 core workstation computers, parallel computing no longer limited to exotic supercomputers or mainframes.[19]

Moreover, electronic devices such as mobile phones and portable music players have begun to incorporate parallel computing capabilities in an effort to provide functionality well beyond those of their predecessors.[19]

More and more, software developers will need to cope with a variety of parallel computing platforms and technologies in order to provide novel and rich experiences for an increasingly sophisticated base of users.[19]

3.2.1 Central Processing Units

For 30 years, one of the important methods for the improving the performance of consumer computing devices has been to increase the speed at which the processors clock operated. Starting with the first personal computers of the early 1980s, consumer central processing units (CPUs) ran with internal clocks operating around 1MHz. About 30 years later, most desktop processors have clock speeds between 1GHz and 4GHz, nearly 1,000 times faster than the clock on the original personal computer. Although increasing the CPU clock speed is certainly not the only method by which computing performance has been improved, it has always been a reliable source for improved performance.[19]

In recent years, manufacturers have been forced to look for alternatives to this traditional source of increased computational power. Because of various fundamental limitations in the fabrication of integrated circuits, it is no longer feasible to rely on upward-spiraling processor clock speeds as a means for extracting additional power from existing architectures. Because of power and heat restrictions as well as a rapidly approaching physical limit to transistor size, researchers and manufacturers have begun to look elsewhere.[19]

Outside the world of consumer computing, supercomputers have for decades extracted massive performance gains in similar ways. The performance of a processor used in a supercomputer has climbed astronomically, similar to the improvements in the personal computer CPU. However, in addition to dramatic improvements in the performance of a single processor, supercomputer manufacturers have also extracted massive leaps in performance by steadily increasing the number of processors. It is not uncommon for the fastest supercomputers to have tens or hundreds of thousands of processor cores working in together.[19]

In 2005, faced with an increasingly competitive marketplace and few alternatives, leading CPU manufacturers began offering processors with two computing cores instead of one. Over the following years, they followed this development with the release of three, four, six, and eight-core central processor units. Sometimes referred to as the multicore revolution, this trend has marked a huge shift in the evolution of the consumer computing market.[19]

Today, it is relatively challenging to purchase a desktop computer with a CPU containing but a single computing core. Even low-end, low-power central processors ship with two or more cores per die. Leading CPU manufacturers have already announced plans for 12- and 16-core CPUs, further confirming that parallel computing has arrived for good.[19]

3.3 The Rise of GPU computing

In comparison to the central processors traditional data processing pipeline, performing general-purpose computations on a graphics processing unit (GPU) is a new concept. In fact, the GPU itself is relatively new compared to the computing field at large.

3.3.1 Brief History of GPUs

We have already looked at how central processors evolved in both clock speeds and core count. In the meantime, the state of graphics processing underwent a dramatic revolution. In the late 1980s and early 1990s, the growth in popularity of graphically driven operating systems such as Microsoft Windows helped create a market for a new type of processor. In the early 1990s, users began purchasing 2D display accelerators for their personal computers. These display accelerators offered hardware-assisted bitmap operations to assist in the display and usability of graphical operating systems.[19]

Around the same time, in the world of professional computing, a company by the name of Silicon Graphics spent the 1980s popularizing the use of three-dimensional graphics in a variety of markets, including government and defense applications and scientific and technical visualization, as well as providing the tools to create stunning cinematic effects. In 1992, Silicon Graphics opened the programming interface to its hardware by releasing the OpenGL library. Silicon Graphics intended OpenGL to be used as a standardized, platform-independent method for writing 3D graphics applications. As with parallel processing and CPUs, it would only be a matter of time before the technologies found their way into consumer applications.[19]

From a parallel-computing standpoint, NVIDIAs release of the GeForce 3 series in 2001 represents arguably the most important breakthrough in GPU technology. The GeForce 3 series was the computing industrys first chip to implement Microsofts then-new DirectX 8.0 standard. For the first time, developers had some control over the exact computations that would be performed on their GPUs.[19]

3.3.2 Early GPU computing

The release of GPUs that possessed Computing attracted many researchers to the possibility of using graphics hardware for more than simply OpenGL- or DirectX-based rendering. The general approach in the early days of GPU computing was extraordinarily convoluted. Because standard graphics APIs such as OpenGL and DirectX were still the only way to interact with a GPU, any attempt to perform arbitrary computations on a GPU would still be subject to the constraints of programming within a graphics API.[19]

Essentially, the GPUs of the early 2000s were designed to produce a color for every pixel on the screen using programmable arithmetic units known as pixel shaders. In general, a pixel shader uses its (x,y) position on the screen as well as some additional information to combine various inputs in computing a final color. The additional information could be input colors, texture coordinates, or other attributes that would be passed to the shader when it ran. But because the arithmetic being performed on the input colors and textures was completely controlled by the programmer, researchers observed that these input colors could actually be any data.[19]

Because of the high arithmetic throughput of GPUs, initial results from these experiments promised a bright future for GPU computing. However, the programming model was still far too restrictive for any critical mass of developers to form. As if the limitations werent severe enough, anyone who still wanted to use a GPU to perform general-purpose computations would need to learn OpenGL or DirectX since these remained the only means by which one could interact with a GPU.[19]

3.4 Introduction to CUDA

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model created by NVIDIA and implemented on the graphics processing units (GPUs) that NVIDIA produce. CUDA gives developers direct access to the virtual instruction set and memory of the parallel computational elements in CUDA GPUs.

Using CUDA, the GPUs can be used for general purpose processing, not exclusively graphics. This approach is known as GPGPU(General-Purpose Computing On Graphics Processing Unit). Unlike CPUs, however, GPUs have a parallel throughput architecture that emphasizes executing many concurrent threads slowly, rather than executing a single thread very quickly.

3.5 CUDA Architecture

3.5.1 General

In November 2006, NVIDIA unveiled the industrys first DirectX 10 GPU, the GeForce 8800 GTX. The GeForce 8800 GTX was also the first GPU to be built with NVIDIAs CUDA Architecture. This architecture included several new components designed strictly for GPU computing which make easier many of the limitations that prevented previous graphics processors from being legitimately useful for general-purpose computation.

3.5.2 CUDA Architecture

The CUDA Architecture allow each and every arithmetic logic unit (ALU) on the chip to be arranged in logical order by a program intending to perform general-purpose computations. Because NVIDIA intended this new family of graphics processors to be used for general purpose computing, these ALUs were designed to use an instruction set for general computation rather than specifically for graphics.

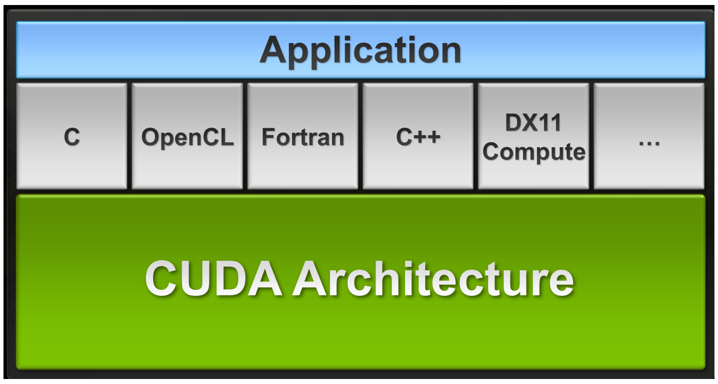


Figure 3.1: CUDA Architecture

Furthermore, the execution units on the GPU were allowed arbitrary read and write access to memory as well as access to a software-managed cache known as shared memory. All of these features of the CUDA Architecture were added in order to create a GPU that would excel at computation in addition to performing well at traditional graphics tasks. CUDA Architecture natively support all computational interfaces as shown in Figure 3.1 (standard languages and APIs)[19]

3.5.3 Use of the CUDA architecture

The effort by NVIDIA to provide consumers with a product for both Computation and graphics could not stop at producing hardware incorporating the CUDA Architecture. Regardless of how many features NVIDIA added to its chips to facilitate computing, there continued to be no way to access these features without using OpenGL or DirectX.

To reach the maximum number of developers possible, NVIDIA took industry standard C and added a relatively small number of keywords in order to harness some of the special features of the CUDA Architecture. A few months after the launch of the GeForce 8800 GTX, NVIDIA made public a compiler for this language, CUDA C. And with that, CUDA C became the first language specifically designed by a GPU

company to facilitate general-purpose computing on GPUs.
In addition to creating a language to write code for the GPU, NVIDIA also provides a specialized hardware driver to exploit the CUDA Architectures massive computational power. Users are no longer required to have any knowledge of the OpenGL or DirectX graphics programming interfaces, nor are they required to force their problem to look like a computer graphics task.

3.6 Execution of Program on GPU

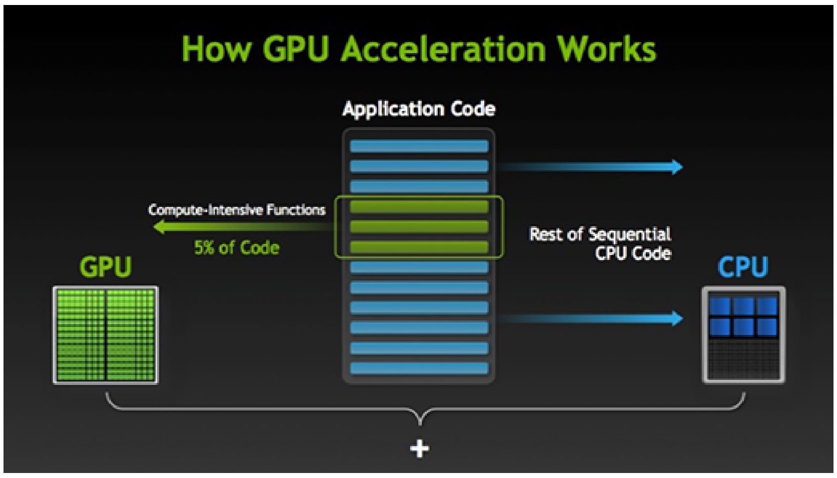

How GPU Acceleration Works
Application Code
Compute-Intensive Functions
5% of Code
GPU
Rest of Sequential CPU Code
CPU
+

Figure 3.2: GPU Acceleration

GPU-accelerated computing offers unprecedented application performance by offloading compute-intensive portions of the application to the GPU, while the remainder of the code still runs on the CPU as per Figure 3.2. From a user's perspective, applications simply run significantly faster.[19]

A simple way to understand the difference between a CPU and GPU is to compare how they process tasks. A CPU consists of a few cores optimized for sequential serial processing while a GPU consists of thousands of smaller, more efficient cores designed for handling multiple tasks simultaneously as per Figure 3.3.

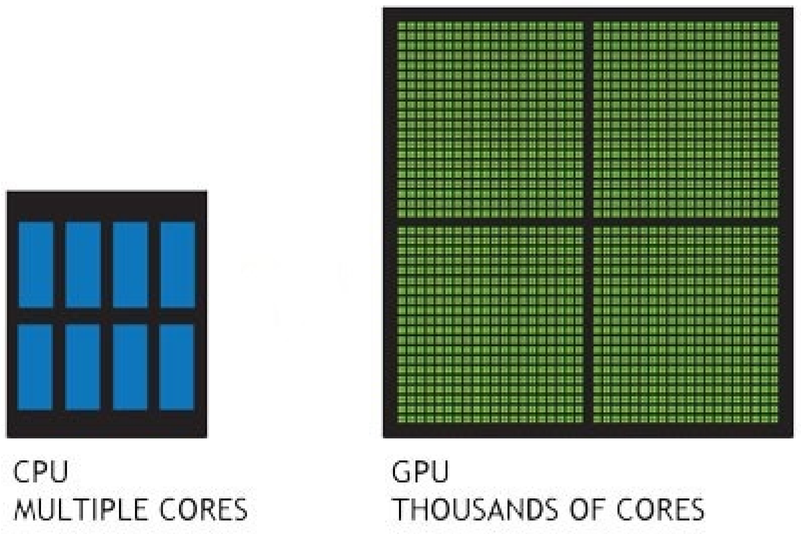


Figure 3.3: CPU Core Vs GPU Core

3.7 Programming Concepts

3.7.1 Heterogeneous computing

Heterogeneous computing refers to systems that use more than one kind of processor incorporating specialized processing capabilities to handle particular tasks. Heterogeneous System Architecture (HSA) systems utilize multiple processor types (typically CPUs and GPUs), GPU processing, apart from its well-known 3D graphics rendering capabilities, can also perform mathematically intensive computations on very large data sets, while CPUs can run the operating system and perform traditional serial tasks.

3.7.2 Kernels

Parallel portion of application execute as a kernel, Entire GPU executes Kernel which means Kernels works as a GPU function.As per Table 3.1 CPU is generally referred as Host and GPU is referred as Device In GPU Computing.If we run CUDA program then Host execute functions and Device Execute kernals.

Table 3.1: function of CPU and GPU in CUDA

Name	Referred as	Work
CPU	Host	Execute Functions
GPU	Device	Execute Kernals

Figure 3.4: Host

Figure 3.5: Device

Figure 3.4 and 3.5 shows picture of Center processing unit and Graphics Processing unit used in Computers. As shown in Table 3.1, CPU is referred as Host and GPU referred as Device in CUDA .

3.7.3 Blocks

If we are running program for vector addition in Parallel on GPU than each Parallel invocation of VectorAdd() kernal is referred to as a Block. Block exist in 2d grid as per Figure 3.6 so Indexing of Blocks is done using blockIdx.x for X direction and blockIdx.Y for Y direction. On the device, each block can execute in parallel.

3.7.4 Threads

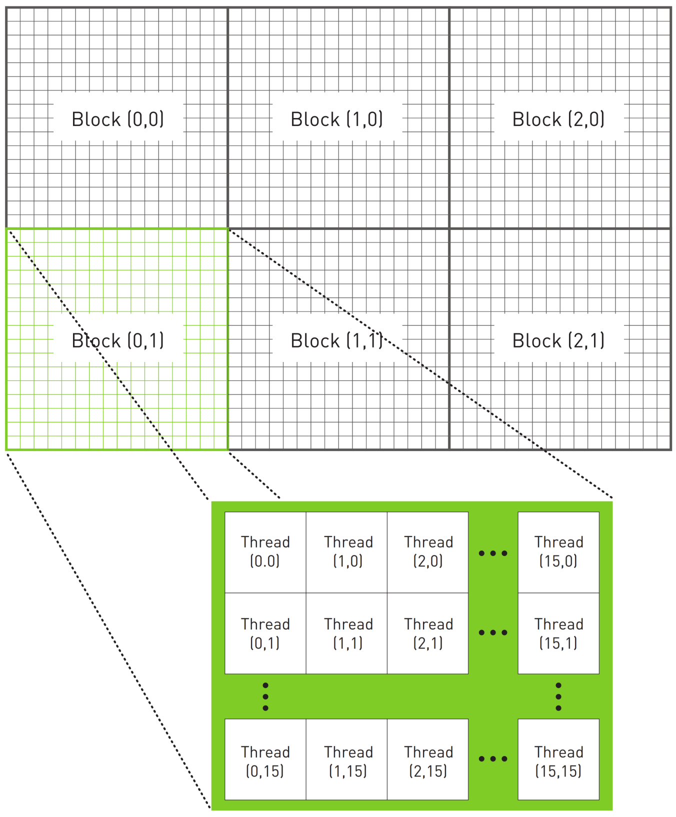


Figure 3.6: CUDA Thread Index

A block can be split into parallel threads. Threads are also exist in 2d/3d Grid. For Indexing of Thread ThreadIdx.x and ThreadIdx.y can be used as shown Figure 3.6 in X and Y direction respectively. On device each thread can execute in parallel. Advantage of Threads over Blocks is that thread share same memory if they are in same block so better computation take place compared to blocks[19] .

3.7.5 Indexing

Indexing of array is one of the most important aspects of GPU computing. for indexing of array in one direction following formula can be used:

$$index = threadId.x + blockId.x * (BlockWidth) \tag{3.1}$$

To find index of Thread as shown in Figure3.7 then index of thread=2+2*4=10.
same way as per Figure 3.8 index of thread=4+2*8=20.
so,as per Figure 3.7 index value starts from 0 to max value 15 which mean program can handle total 16 number of process in parallel for this particular size of grid. we can define Size of Grid using number of blocks and threads. Grid size(4,3) means 4 number of blocks in which each block consist of 3 number of threads. Figure 3.7 and 3.8 are example of (4,4) and (4,8)grid size respectively.

3.7.6 Memory management

Host and device memory are

- separate entities Device pointers point to GPU memory
- May be passed to/from host code
- May not be dereferenced in host code Host pointers point to CPU memory
- May be passed to/from device code

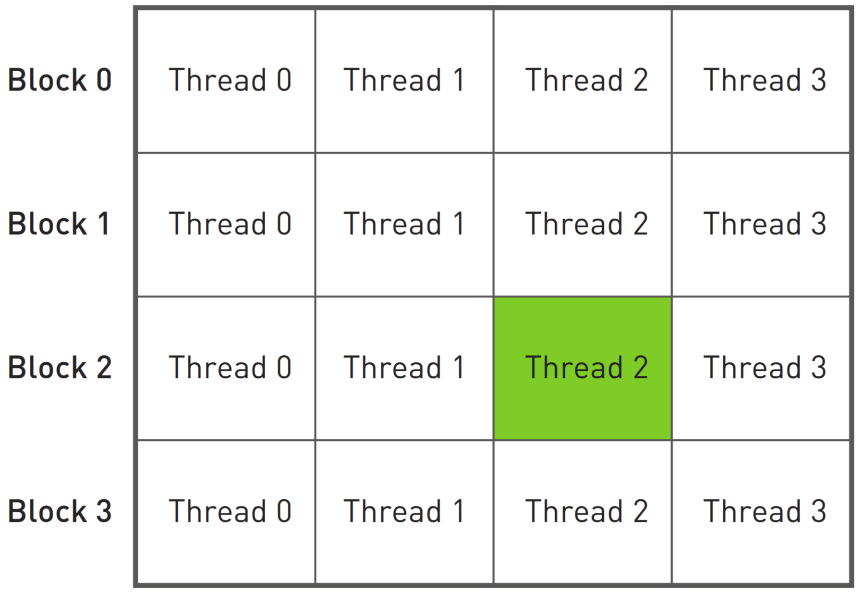


Figure 3.7: CUDA Block Index

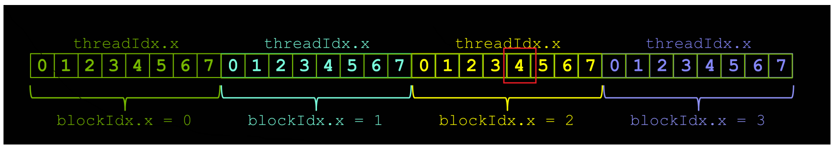


Figure 3.8: Indexing of Array

- May not be dereferenced in device code

Simple CUDA API(Table 3.2) for handling device memory Similar to the C equivalents malloc(), free(), memcpy().

Table 3.2: CUDA C functions for Memory Management

Name of Function	**Descripson**
cudaMalloc()	allocate Space for Variable on GPU
cudaMemcpy()	Copy memory from Host to Device in both ways
cudaFree()	free up space allocated on GPU

3.8 Languages Supported by NVIDIA CUDA

List of Language in which we can do GPU programming by using CUDA libraries are shown in Table 3.3 .

Table 3.3: Languages Supported by NVIDIA CUDA

Name of Language	**Library or API**
Fortran	FORTRAN CUDA
F#	Alea.CUDA
Java	jCUDA
Mathematica	CUDALink
MATLAB	Parallel Computing Toolbox
.NET	CUDA.NET
Python	Numba, NumbaPro, PyCUDA

3.9 Applications Accelerated using CUDA

Since its debut in early 2007, a variety of industries and applications have enjoyed a great deal of success by choosing to build applications in CUDA C. These benefits often include orders-of-magnitude performance improvement over the previous state-of-the-art implementations.

Furthermore, applications running on NVIDIA graphics processors enjoy superior performance per watt than implementations built exclusively on traditional central processing technologies. The following list represent just a few Application in which people have put CUDA C and the CUDA Architecture into successful use.

- MATLAB
- ANSYS Mechanical
- MATHEMATICA
- ABAQUS

- LabView

3.10 Development Environment for CUDA C

The prerequisites to developing code in CUDA C are as follows:

- A CUDA-enabled graphics processor
- An NVIDIA device driver
- A CUDA development toolkit
- A standard C compiler

NVIDIA Cuda Toolkit used for development environment for c/c++ language for building GPU accelerated application.
The CUDA toolkit include a compiler for NVIDIA GPUs, Math libraries and tools for debugging and optimizing the performance of applications.Latest version of CUDA toolkit available is 6.5.
For standard C Compiler on windows Oprating System Visual Studio or Eclipse can be used as per you preferance.NVIDIA Nsight Plug in is available for both compiles.

3.11 Example : Matrix Multiplication

3.11.1 General

To understand effect of parallel programming in computation heavy application example of matrix multiplication is shown in this chapter. For comparison and check speed improvement two different programs are developed. First program is standard c program(sequential) which run on CPU using c language. Second program is developed using CUDA c language which run in Parallel on GPU. Comparison is done based on execution time and speed up factor as shown in table 3.4 .

3.11.2 Algorithm

As shown in Figure 3.9 example of Matrix Multiplication $[A] * [B] = [C]$ is given. So in case of sequential program all elements of $[C]$ matrix is calculated one by one in sequence.

For example If we execute sequential program, It will calculate all elements of $[C]$ matrix in following sequence first C11,second C12,third C13 .... C33. In case of parallel program all elements of $[C]$ matrix are calculated in parallel. In parallel program, All members of $[C]$ matrix will be calculated together at same time in parallel.

Algorithm for Sequential program is simple to understand no explanation required for that but in case of parallel program we need to make kernal for matrix multiplication in such a way that each element of resulting matrix is executed in parallel.

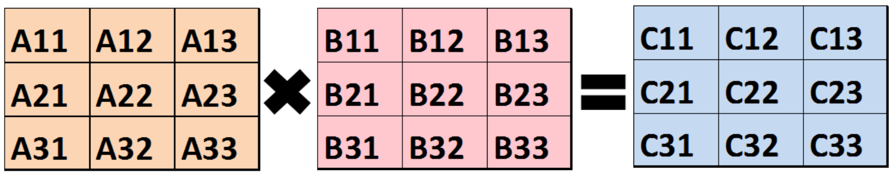


Figure 3.9: Matrix Multiplication of A and B Matrix

3.11.3 Code

Sequential Program using C

Code for Sequential matrix multiplication is given below.

```
//Matrix Multiplication in CPU
#include<stdio.h>
#include<time.h>

//BLOCK_SIZE is size Multiplier for Square matrix
```

```
#define BLOCK_SIZE  1

void  main( )
{
int N, K;
K = 100;
N = K*BLOCK_SIZE;//N=size of square Matrix
float *A, *B, *C;//A*B=C

//for measurement of execution time
clock_t begin, end;
double ExecTime;

//Output file
FILE *out;

//starting of Matrix Multiplication
begin = clock();

out = fopen("output.txt", "w");

printf("Executing Matrix Multiplcation");
printf("\nMatrix size: %d\n", N);

A = new float[N*N];
B = new float[N*N];
C = new float[N*N];

// Initialize matrices on the host
```

```
for (int row = 0; row<N; row++)
{
for (int col= 0; col<N; col++)
{
A[row*N + col] = row*col;
B[row*N + col] = -row*col;
}
}
// Print A and B matrix
fprintf(out, "Matrix A\n");
for (int row = 0; row<N; row++)
{
for (int col = 0; col<N; col++)
{
fprintf(out, "%f\t", A[row*N + col]);
}
fprintf(out,"\n");
}

fprintf(out, "Matrix B\n");
for (int row = 0; row<N; row++)
{
for (int col = 0; col<N; col++)
{
fprintf(out, "%f\t", B[row*N + col]);
}
fprintf(out, "\n");
}
```

```
// Now do the matrix multiplication on the CPU
float sum;
for (int row = 0; row<N; row++){
for (int col = 0; col<N; col++){
sum = 0;
for (int n = 0; n<N; n++){
sum += A[row*N + n] * B[n*N + col];
}
C[row*N + col] = sum;
}
}

fprintf(out, "Matrix C\n");
for (int row = 0; row<N; row++)
{
for (int col = 0; col<N; col++)
{
fprintf(out, "%f\t", C[row*N + col]);
}
fprintf(out, "\n");
}

//end of Matrix Multiplication
end = clock();
ExecTime = end - begin;
   printf("\nExecution Time=%f",ExecTime);
fprintf(out, "\ntime spent=%f",ExecTime);
```

```
}
```

Parallel Program using CUDA C

Code for parallel matrix multiplication is given below. Unlike in sequential code, in parallel code Kernal calculate all products of matrix multiplication in parallel at a same time.

```
  //Program for Matrix multiplication on GPU
#include<cuda.h>
#include<cuda_runtime.h>
#include<stdio.h>
#include<time.h>

//define dimension of block
#define BLOCK_SIZE 1  //Block_size=Number of threads per block

//kernal for Matrix Multiplication
__global__ void gpuMM(float *A, float *B, float *C, int N)
{
// Matrix multiplication for NxN matrices C=A*B
// Each thread computes a single element of C
int row = blockIdx.y*blockDim.y + threadIdx.y;
int col = blockIdx.x*blockDim.x + threadIdx.x;

float sum = 0;

for (int n = 0; n < N; ++n)
sum += A[row*N + n] * B[n*N + col];
```

```
C[row*N + col] = sum;
}

int main( )
{
// where A, B and C are NxN matrices on Device
//hA,hB and hC are matrices on Host(CPU)
//dA,dB and dC are for Device(GPU)

int N, K;  //K=size of Grid in 2d
K = 100;
N = K*BLOCK_SIZE; // N=Size of square matrix for 100*100 size of Matrix N=100

clock_t begin, end; //for mesurement of execution time
double dt;

FILE *out; // output file
out = fopen("output.txt", "w");

//begin time of Matrix Multiplication
begin = clock();

printf("Executing Matrix Multiplcation");
printf("\nMatrix size: %d", N);



// Allocate memory on the host
float *hA, *hB;
```

```
hA = new float[N*N];
hB = new float[N*N];



// Initialize matrices on the host
for (int j = 0; j<N; j++){
for (int i = 0; i<N; i++){
hA[j*N + i] = i*j;
hB[j*N + i] = -i*j;
    }
}

//print Matrix A and B
fprintf(out,"\n A matrix \n");
for (int j = 0; j<N; j++){
for (int i = 0; i<N; i++){
fprintf(out, "%f\t", hA[j*N + i]);

}
fprintf(out, "\n");
}
fprintf(out, "\n B matrix \n");
for (int j = 0; j<N; j++){
for (int i = 0; i<N; i++){
fprintf(out, "%f\t", hB[j*N + i]);

}
fprintf(out, "\n");
}



// Allocate memory on the device
int size = N*N*sizeof(float);
float *dA, *dB, *dC;
cudaMalloc(&dA, size);
cudaMalloc(&dB, size);
cudaMalloc(&dC, size);

//specify size of Block and Thread
dim3 threadBlock(BLOCK_SIZE, BLOCK_SIZE);
dim3 grid(K, K);

// Copy matrices from the host to device
cudaMemcpy(dA, hA, size, cudaMemcpyHostToDevice);
cudaMemcpy(dB, hB, size, cudaMemcpyHostToDevice);

//Execute the matrix multiplication kernel

gpuMM <<< grid, threadBlock >>> (dA, dB, dC, N);



// Allocate memory to store the GPU answer on the host
float *hC;
hC = new float[N*N];

// Now copy the GPU result back to CPU
cudaMemcpy(hC, dC, size, cudaMemcpyDeviceToHost);
```

```
fprintf(out, "\n C matrix \n");
for (int j = 0; j<N; j++){
for (int i = 0; i<N; i++){
fprintf(out, "%f\t", hC[j*N + i]);

}
fprintf(out, "\n");
}
//end of Matrix Multiplication
end = clock();
dt = (end - begin);//total Execution time
printf("Execution time=%f\n",dt);
//fprintf(out, "\ntime spent=%f", dt);
}
```

3.11.4 Results

Execution time for different sizes of Matrix is compared for both program as shown in Table 3.4 which clearly indicate that with increase in size of matrix execution time for sequential program increase drastically.

In case of parallel program with increase in execution time is not much compared to sequential program.Comparison of execution time for sequential program and parallel program is done in Figure 3.10 and 3.11.

As per shown in Figure 3.10 up to matrix size 500 $\times$500 Execution time of Sequential Program is lower than Parallel Program after that size speed Up factor for Matrix Multiplication Program is increasing drastically as shown in Figure3.11.Speed Up factor is ratio of execution time of Sequential Program to Parallel Program.

Table 3.4: Execution Time of Matrix Multiplication using Core i7 CPU and Nvidia Gefore GT 750M GPU

Size of Matrix	**CPU(ms)**	**GPU(ms)**	**Speed Up Factor**
100×100	4	534	0.01
200×200	26	545	0.05
300×300	120	571	0.21
400×400	220	599	0.37
500×500	487	639	0.76
600×600	850	650	1.31
700×700	1300	680	1.91
800×800	1900	700	2.71
900×900	3326	730	4.56
1000×1000	4505	769	5.86
1100×1100	9239	818	11.29
1200×1200	13620	907	15.02
1300×1300	30000	1010	29.7
1400×1400	36684	1118	32.81
1500×1500	48763	1235	39.48

3.12 Summary

This chapter include basic understanding related to GPU programming which include rise of GPU computing to languages for GPU computing. GPU programming concepts and various terminology is also included in this chapter. Example of Matrix multiplication Parallel program is also given with algorithm and results to have clear understanding of parallel computing on GPU.

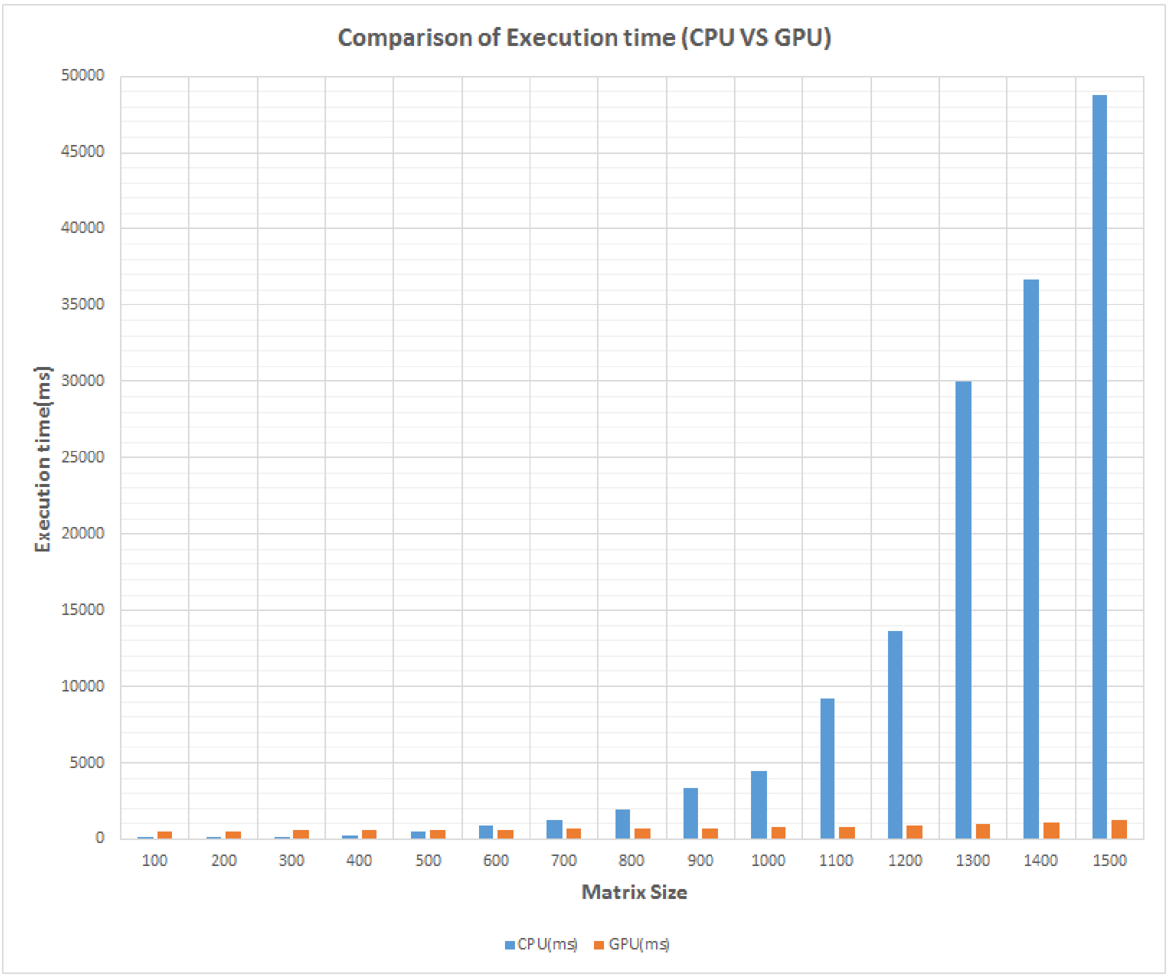


Figure 3.10: Execution Time Comparison Graph1

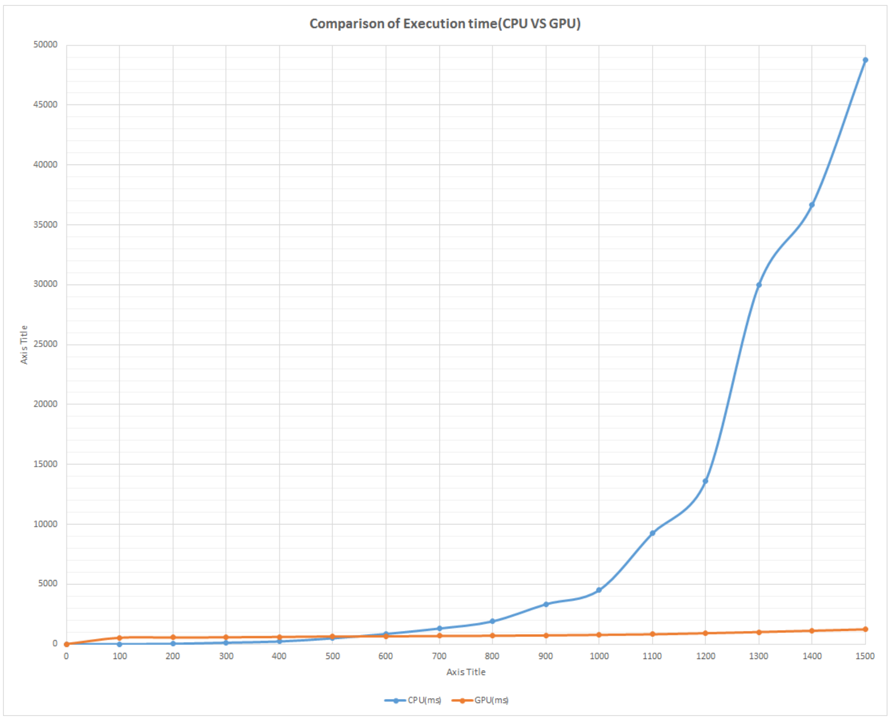


Figure 3.11: Execution Time Comparison Graph2

Chapter 4

Gaussian Elimination

4.1 General

Gaussian Elimination is algorithm for solving system of linear equation. This method is consist of two parts, Triangularization and Back Substitution.

In Triangulariztion , Matrix is converted to Uppertriangular matix using row operation. If we represent system of linear equation using AX=B where X is unknown, then we need to convert it in to UX=G using row operations. In Back Substitution we first solve last equation and after that next to last.

In Gaussian Elimination if we denote the original linear system by Ax=b where, A=[a_{ij}] ,b=$[b_i]^T$,1 $\leq$ i $\leq$ n and n is order of system. We reduce the system to the triangular form Ux=g by adding multiples of one equation to another equation, eliminating some unknown from the second equation.

4.2 Algorithm of Gaussian Elimination

Assume $a_{11} \neq 0$ and Define row multiplier by

$$\mathrm{m}_{i1}=\frac{a_{i1}{}^{(1)}}{a_{1}1^{(1)}}, i=2,3,...,n$$

These are used in eliminating the x_1 term form below equation through n.

Define

$$a_{ij}{}^{(2)} = a_{ij}{}^{(1)} - m_{i1}a_{1j}{}^{(1)} \qquad i,j = 2,..,n$$

$$b_i{}^{(2)} = b_i{}^{(1)} - m_{i1}b_1{}^{(1)} \qquad i = 2,..,n$$

Here, First row of A and b are left undistributed, and the first column of $A_{(1)}$, below the diagonal,is set to zero. The system $A^{(2)}x = b^{(2)} lookslike$

$$\begin{bmatrix} a_{11}{}^{(1)} & a_{12}{}^{(1)} & . & . & a_{1n}{}^{(1)} \\ 0 & a_{22}{}^{(2)} & . & . & a_{2n}{}^{(2)} \\ . & . & . & . & . \\ . & . & . & . & . \\ 0 & a_{n2}{}^{(2)} & . & . & a_{nn}{}^{(2)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ . \\ . \\ x_n \end{bmatrix} = \begin{bmatrix} b_1{}^{(1)} \\ b_2{}^{(2)} \\ . \\ . \\ b_n{}^{(2)} \end{bmatrix}$$

We continue to eliminate unknowns,going onto columns 2, 3, etc., and this is expressed generally as follows.

Let $1 \leq k \leq n-1$. Assume that $A_x{}^{(\mathrm{k})} = b^{(2)}$ has been constructed with $x_1, x_2, ...x_{k-1}$ eliminated at successive stages and $A^{(\mathrm{k})}$ has the form

$$A_x{}^{(\mathrm{k})} = \begin{bmatrix} a_{11}{}^{(1)} & a_{12}{}^{(1)} & . & . & . & . & . & a_{1n}{}^{(1)} \\ 0 & a_{22}{}^{(2)} & . & . & . & . & . & a_{2n}{}^{(2)} \\ 0 & . & . & 0 & a_{kk}{}^{(\mathrm{k})} & . & . & a_{kn}{}^{(\mathrm{k})} \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ 0 & . & . & 0 & a_{nk}{}^{(\mathrm{k})} & . & . & a_{nn}{}^{(\mathrm{k})} \end{bmatrix}$$

Assume $a_{kk}{}^{(\mathrm{k})} \neq 0$Define the multipliers.

$$m_{ij} = \frac{a_{ik}^{(}k)}{a_{kk}^{(}k)} \qquad ,i = k+1,...,n$$

Use these to remove the unknown's x_k from equations k+1 through n. Define

$$a_{ij}^{(k+1)} = a_{ij}^{(k)} - m_{ik}a_{kj}^{(k)}$$
$$b_i^{(k+1)} = b_i^{(k)} - m_{ik}b_k^{(k)} \qquad ,i = k+1, ..., n$$

The earlier rows 1 through k are left undisturbed, and zeros are introduced into column k below the diagonal element. By continuing in this manner, after n-1 steps, we obtain $A^{(n)}x = b^{(n)}$ i.e.

$$a_{ij}{}^{(2)} = a_{ij}{}^{(1)} - m_{i1}a_{1j}{}^{(1)} \qquad i, j = 2, .., n$$

$$b_i{}^{(2)} = b_i{}^{(1)} - m_{i1}b_1{}^{(1)} \qquad i = 2, .., n$$

Here, First row of A and b are left undistributed, and the first column of $A_{(1)}$, below the diagonal,is set to zero. The system $A^{(2)}x = b^{(2)} lookslike$

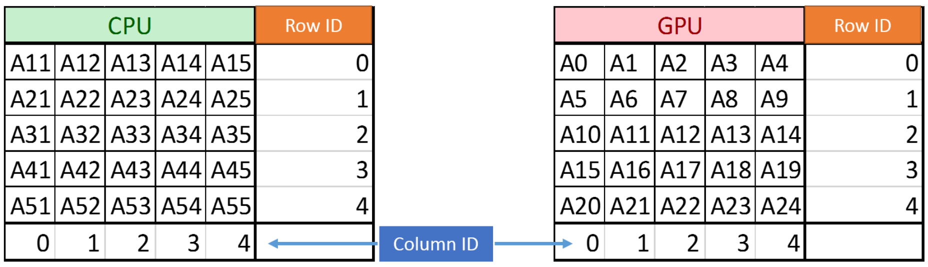
$$\begin{bmatrix} a_{11}{}^{(1)} & . & . & . & a_{1n}{}^{(1)} \\ 0 & . & & & . \\ . & & . & & . \\ . & & & . & . \\ 0 & . & . & . & a_{nn}{}^{(\mathrm{n})} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ . \\ . \\ x_n \end{bmatrix} = \begin{bmatrix} b_1{}^{(1)} \\ . \\ . \\ . \\ b_n{}^{(\mathrm{n})} \end{bmatrix}$$

Let U= A$^{(n)}$ and g=b$^{(n)}$. The system Ux=g is Upper triangular and easy to solve by back substitution.

$$x_n = \frac{g_n}{unn}$$

and

$$x_k = \frac{1}{u_{kk}}[g_k - \textstyle\sum_{j=k+1}^{n} u_{kj}x_j] \qquad \text{,k=n-1,n-2,...1}$$



CPU					Row ID
A11	A12	A13	A14	A15	0
A21	A22	A23	A24	A25	1
A31	A32	A33	A34	A35	2
A41	A42	A43	A44	A45	3
A51	A52	A53	A54	A55	4
0	1	2	3	4	

GPU					Row ID
A0	A1	A2	A3	A4	0
A5	A6	A7	A8	A9	1
A10	A11	A12	A13	A14	2
A15	A16	A17	A18	A19	3
A20	A21	A22	A23	A24	4
0	1	2	3	4	

Figure 4.1: Storage of Matrix for Parallel computing

4.2.1 Storage

As we can see in Figure 4.1 How we can store matrix for parallel computing in vector format because parallel programming does not support multi dimensional array. In this Figure 4.1 left and right table show storage of matrix in sequential and parallel program. In parallel program index of Matrix is calculated by

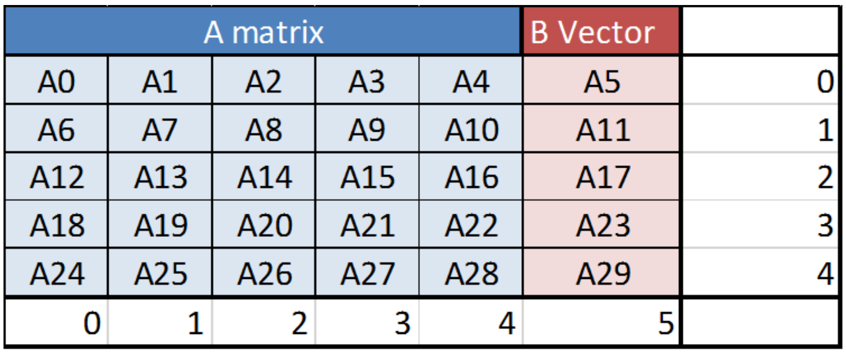
$$\text{Index}=\text{RowID}*\text{Width of Matrix}+\text{ColumnID}.$$

$$\text{For A32}=(2\times5)+1=11,\ \text{For A54}=(4\times5)+3=23$$

Gaussian Elimination parallel program presented in this report is use to solve Ax=b type of linear system which is similar to Kx=F where K=stiffness matrix, x=Displacement Vector and F=Force vector.
In Gaussian Elimination parallel program we need to store Stiffness matrix and Force Vector. Program presented in this report made to store Stiffness matrix and Force Vector as single matrix because of parallel programming does not support multi-dimension array. In Gaussian Elimination method matrix shown in Figure 4.2 is referred as Augmented Matrix.

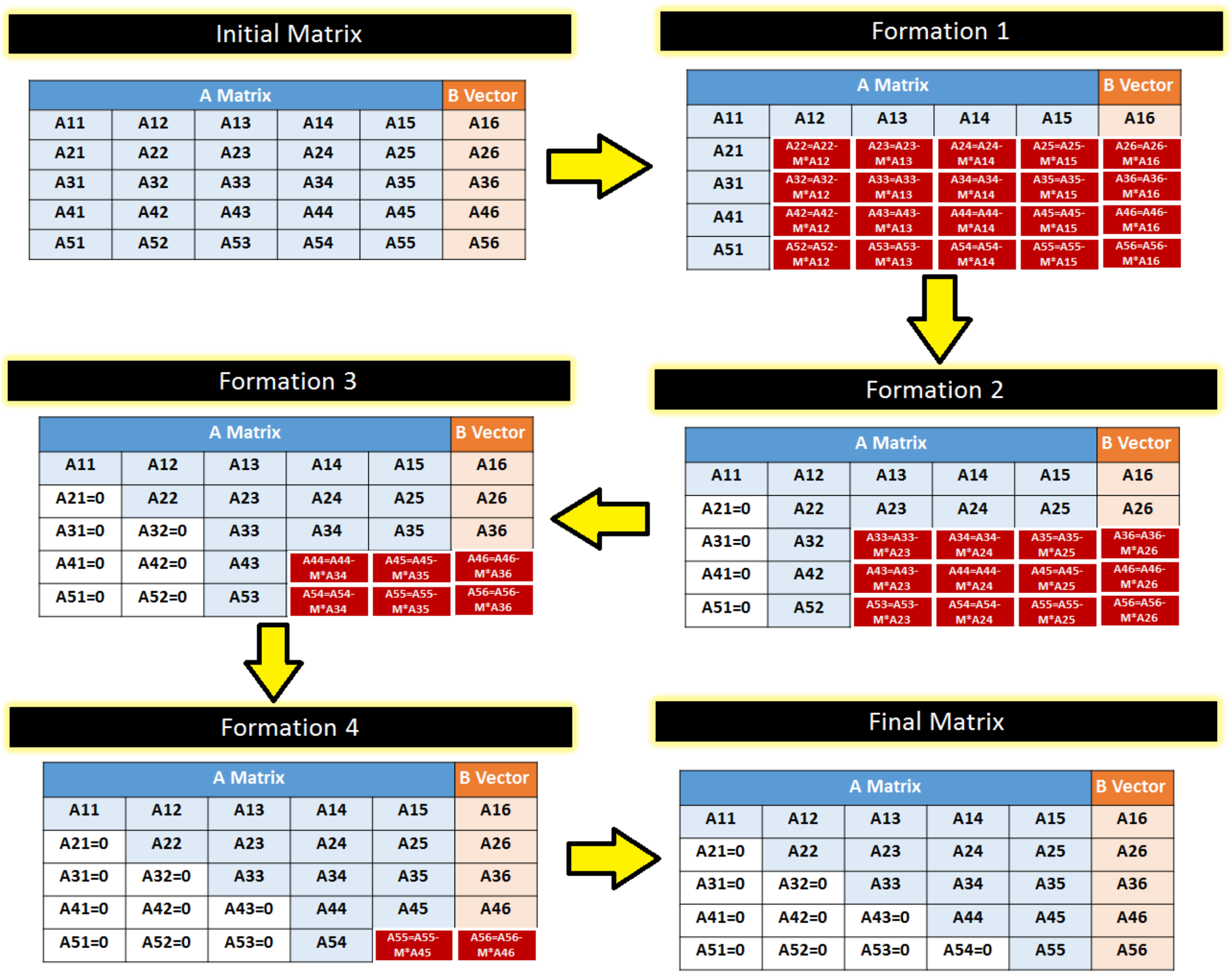
4.2.2 Formation of Upper Triangular Matrix

As shown in Figure4.3, in sequential process all the Red Elements are calculated one after one in sequence.As in case of Parallel Program all red Elements are calculated in parallel.



A matrix					B Vector	
A0	A1	A2	A3	A4	A5	0
A6	A7	A8	A9	A10	A11	1
A12	A13	A14	A15	A16	A17	2
A18	A19	A20	A21	A22	A23	3
A24	A25	A26	A27	A28	A29	4
0	1	2	3	4	5	

Figure 4.2: Storage of stiffness matrix(A) and Force Vector(B)



Initial Matrix

A Matrix					B Vector
A11	A12	A13	A14	A15	A16
A21	A22	A23	A24	A25	A26
A31	A32	A33	A34	A35	A36
A41	A42	A43	A44	A45	A46
A51	A52	A53	A54	A55	A56

Formation 1

A Matrix					B Vector
A11	A12	A13	A14	A15	A16
A21	A22=A22-M*A12	A23=A23-M*A13	A24=A24-M*A14	A25=A25-M*A15	A26=A26-M*A16
A31	A32=A32-M*A12	A33=A33-M*A13	A34=A34-M*A14	A35=A35-M*A15	A36=A36-M*A16
A41	A42=A42-M*A12	A43=A43-M*A13	A44=A44-M*A14	A45=A45-M*A15	A46=A46-M*A16
A51	A52=A52-M*A12	A53=A53-M*A13	A54=A54-M*A14	A55=A55-M*A15	A56=A56-M*A16

Formation 2

A Matrix					B Vector
A11	A12	A13	A14	A15	A16
A21=0	A22	A23	A24	A25	A26
A31=0	A32	A33=A33-M*A23	A34=A34-M*A24	A35=A35-M*A25	A36=A36-M*A26
A41=0	A42	A43=A43-M*A23	A44=A44-M*A24	A45=A45-M*A25	A46=A46-M*A26
A51=0	A52	A53=A53-M*A23	A54=A54-M*A24	A55=A55-M*A25	A56=A56-M*A26

Formation 3

A Matrix					B Vector
A11	A12	A13	A14	A15	A16
A21=0	A22	A23	A24	A25	A26
A31=0	A32=0	A33	A34	A35	A36
A41=0	A42=0	A43	A44=A44-M*A34	A45=A45-M*A35	A46=A46-M*A36
A51=0	A52=0	A53	A54=A54-M*A34	A55=A55-M*A35	A56=A56-M*A36

Formation 4

A Matrix					B Vector
A11	A12	A13	A14	A15	A16
A21=0	A22	A23	A24	A25	A26
A31=0	A32=0	A33	A34	A35	A36
A41=0	A42=0	A43=0	A44	A45	A46
A51=0	A52=0	A53=0	A54	A55=A55-M*A45	A56=A56-M*A46

Final Matrix

A Matrix					B Vector
A11	A12	A13	A14	A15	A16
A21=0	A22	A23	A24	A25	A26
A31=0	A32=0	A33	A34	A35	A36
A41=0	A42=0	A43=0	A44	A45	A46
A51=0	A52=0	A53=0	A54=0	A55	A56

Figure 4.3: Formation of Upper triangular matrix in Program

4.3 Sequential Program

In this section sequential code of Gaussian Elimination solver is given. Code is divided into two parts. Part 1 convert augmented matrix to upper triangular matrix and Part 2 takes care of back substitution.

4.3.1 Generation of Upper Triangular Matrix

Code to generate upper triangular matrix from augmented matrix is given below.

```
// loop for the generation of upper triangular matrix
for (j = 0; j < N; j++)
{
fprintf(out, "\n\n****Formation of Upper trinagular matrix=%d ****\n", j + 1);
fprintf(out,"------M vector-----");
for (i = 0; i < N; i++)
{
if (i>j)
{
c = A[i*(N+1)+j] / A[j*(N+1)+j];
fprintf(out,"\n%f",c);
for (k = 0; k < N + 1; k++)
{
A[i*(N+1)+k] = A[i*(N+1)+k] - c*A[j*(N+1)+k];
}
}
}
```

4.3.2 Back Substitution

Code for Back Substitution process is given below.

```
float *x;
x = (float *)malloc(N * sizeof(float));
x[N-  1] = A[N*(N + 1)-1] / A[N*(N+1)-2];
fprintf(out, "X[%d]=%f", N - 1, x[N - 1]);
//this loop is for backward substitution
float sum;
for (int i = N - 2; i >= 0; i--)
{
sum = 0;
for (int j = i + 1; j < N; j++)
{
sum = sum + A[i*(N + 1) + j] * x[j];
}
x[i] = (A[i*(N + 1) + N] - sum) / A[i*(N + 1) + i];
}
```

4.4 Parallel Program

In this section parallel code of Gaussian Elimination solver is given. Same as sequential program, Code is divided into two parts. Part 1 convert Augmented matrix to upper triangular matrix in parallel and Part 2 takes care of back substitution in sequence.

4.4.1 Generation of Upper Triangular Matrix

Code to generate upper triangular matrix from Augmented matrix is given below.

```
// Kernel-1 for Generation of m vector
```

```
__global__ void k1(float *A,float *M,float c,int i,int N)
{
int tid = blockIdx.x + gridDim.x*blockIdx.y;
M[tid] = A[(tid+i)*(N+1)+i-1]/c;
}

// Kernel-2 for Update A Matrix
__global__ void k2(float *A,float *M,int i,int N)
{
int tid = blockIdx.x + gridDim.x*blockIdx.y;
int row_id = tid / (N-i + 2);
int col_id = tid % (N-i + 2);
int index = 0;
if (i == 1)
{
index = i*(N + 2) - 1 + tid;
}
else
{
index = i*(N + 2) - 1 + tid+i*row_id-row_id;

}
int s_index = col_id + (i-1)*(N+2);
A[index] = A[index] - M[row_id] * A[s_index];

}
```

4.4.2 Back Substitution

Code for Back Substitution process is given below which is same as sequential program.

```
float *x;
x = (float *)malloc(N * sizeof(float));
x[N - 1] = A[N*(N + 1) - 1] / A[N*(N + 1) - 2];
fprintf(out, "X[%d]=%f",N-1, x[N-1]);
//this loop is for backward substitution
float sum;
for (int i = N - 2; i >=0; i--)
{
sum = 0;
for (int j = i+1 ; j < N; j++)
{
sum = sum + A[i*(N+1)+j] * x[j];
}
x[i] = (A[i*(N+1)+N] - sum) / A[i*(N+1)+i];
}
```

4.5 Results

In Table 4.1 it is significant that with increase in number of equation speed up factor of program is also increasing. So for higher Number of equation execution time of Program on GPU compared CPU program will be much lower. So by implementing this program as solver in another program in which number of equations are in terms of 100000-500000 we can significantly improve execution time. For testing of this program core i7-4500U processor CPU and Nvidia GeForce GT 750M GPU is used as hardware platform.

Table 4.1: Execution Time of Gaussian Elimination

Number of Equations	**CPU (ms)**	**Communication Time (ms)**	**Execution time (ms)**	**Total time(ms)**	**SpeedUp Factor**
100	16	1	75	76	0.2105
200	65	1	255	256	0.2539
300	150	1	477	478	0.3138
400	290	2	652	654	0.4434
500	503	2	770	772	0.6516
600	740	2	992	994	0.7445
700	1100	2	1366	1368	0.8041
800	1550	3	1897	1900	0.8158
900	2110	3	2485	2488	0.8481
1000	2764	4	3159	3163	0.8739

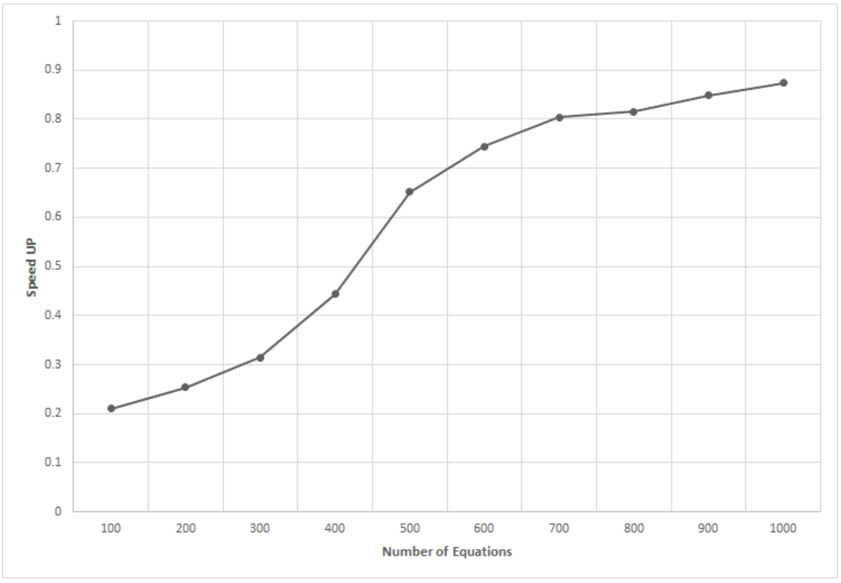


Figure 4.4: Comparison of Speed Up Factor with number of equations

4.6 Summary

In this chapter Gaussian Elimination Algorithm for solution of Ax=b type of linear equation is presented. Based on that algorithm, Gaussian Elimination Sequential and parallel programs are implemented on CPU and GPU respectively.Comparison of both program is done based on execution time and Speed Up factor .

Chapter 5

Half Band Matrix

5.1 General

Half Band Matrix solver is improved version of Gaussian Elimination Solver mainly suitable for process like Analysis of Structure. In Process of Structure analysis, Final version of Assemble stiffness matrix is always same like symmetrical along diagonal member. So, to reduce storage and number of calculation only Upper triangular Matrix is stored.

If we take example of 5×5 matrix size than as shown in Figure 5.1 it shows matrix used for gaussian elimination in which matrix is diagonally symmetrical and element shown with red colour are zero. Now in Gaussian elimination matrix only green elements shown in figure is important for storage which is diagonally symmetrical. In Half band Matrix we store only Upper triangular matrix and fill remaining portion with zero elements.

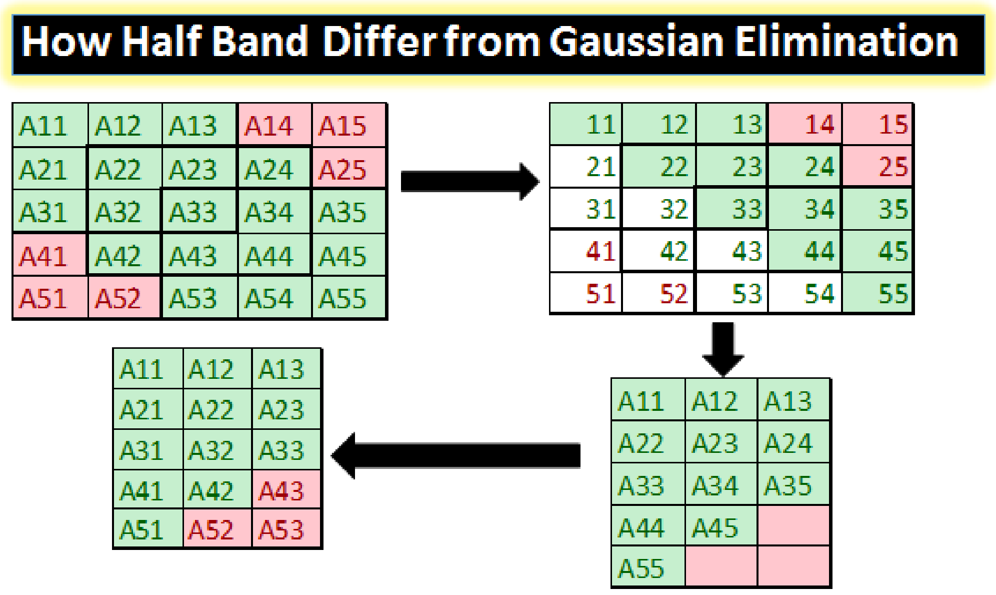


Figure 5.1: How Half Band Matrix differ from Gaussian Elimination Matrix

5.2 Algorithm of Half Band Solver

Storage scheme of half band matrix

$$\begin{bmatrix} a11 & a12 & a13 & a14 & a15 \\ a22 & a23 & a24 & a25 & a26 \\ a33 & a34 & a35 & a36 & a37 \\ a44 & a45 & a46 & a47 & a48 \\ a55 & a56 & a57 & a58 & \\ a66 & a67 & a68 & & \\ a77 & a78 & & & \\ a88 & & & & \end{bmatrix}$$

a. c=$\frac{a12}{a11}$

b. Modify second row element

$a_{21}^{'} = a_{21} - \frac{a_{12}a_{12}}{a_{11}}$

$a_{22}^{'} = a_{22} - \frac{a_{13}a_{12}}{a_{11}}$

$a_{23}^{'} = a_{23} - \frac{a_{14}a_{12}}{a_{11}}$

$a_{24}^{'} = a_{24} - \frac{a_{15}a_{12}}{a_{11}}$

c. Modify right hand side b_2 as

$b_2^{'} = b_2 - \frac{a_{12}b_1}{a_{11}}$

d. Now change value of a_{12} as follows

$a_{12}^{'} = c$

e. Now change value of a_{12} as follows

$a_{12}^{'} = c$

After Following above four steps we get following matrix

$$\begin{bmatrix} a\acute{1}1 & a12 & a13 & a14 & a15 \\ a\acute{2}1 & a\acute{2}2 & a\acute{2}3 & a24 & a25 \\ a31 & a32 & a33 & a34 & a35 \\ a41 & a42 & a43 & a44 & a45 \\ a51 & a52 & a53 & a54 & \\ a61 & a62 & a63 & & \\ a71 & a72 & & & \\ a81 & & & & \\ & & & & \end{bmatrix}$$

Taking the second equation, one may follow the same steps as 1 to 4 as before. This is to be repeated for (Number of Equation-1) times. For N^{th} equation and modifying the a_{NL}^{th} element

$$a_{IJ}^{'} = a_{IJ} - \frac{a_{NL}a_{NK}}{a_{N1}}$$
$$b_I^{'} = b_I - \frac{a_{NL}b_N}{a_{N1}}$$

5.2.1 Storage of Half Band Matrix

To understand Half Band Matrix storage consider symmetric matric shown below

$$\begin{bmatrix} a11 & a12 & a13 & a14 & a15 & & & \\ a21 & a22 & a23 & a24 & a25 & a26 & & \\ a31 & a32 & a33 & a34 & a35 & a36 & a37 & \\ a41 & a42 & a43 & a44 & a45 & a46 & a47 & a48 \\ a51 & a52 & a53 & a54 & a55 & a56 & a57 & a58 \\ & a62 & a63 & a64 & a65 & a66 & a67 & a68 \\ & & a73 & a74 & a75 & a76 & a77 & a78 \\ & & & a84 & a85 & a86 & a87 & a88 \end{bmatrix}$$

Above matrix is symmetrical about main diagonal, so in Half Band Matrix only storage of Upper Triangular take in to consideration as shown in below matrix

$$\begin{bmatrix} a11 & a12 & a13 & a14 & a15 \\ a22 & a23 & a24 & a25 & a26 \\ a33 & a34 & a35 & a36 & a37 \\ a44 & a45 & a46 & a47 & a48 \\ a55 & a56 & a57 & a58 & \\ a66 & a67 & a68 & & \\ a77 & a78 & & & \\ a88 & & & & \end{bmatrix}$$

Total Number of Element in Gaussian Elimination Matrix=8×8=64.Total Number of Element in Half Band Matrix =8×5=40. With increase size of matrix this storage efficiency improve drastically.

5.2.2 Formation of Upper Triangular matrix in Half Band Format

As per shown in Figure 5.2 number of formation to covert Half Band matrix to Upper triangular matrix depend on number of row. Each formation is also divided into two sub part.

Part 1 calculated M vector which is required to update A and B matrix. Part 2 Update Matrix A and B as per calculated value of M vector. Figure 5.2 shows how Half band matrix is converted to upper triangular matrix with each formation level. In this figure elements shown with red colour are updated with each formation. For sequential program red colour elements are updated in sequence and for parallel program all red elements are updated in parallel.

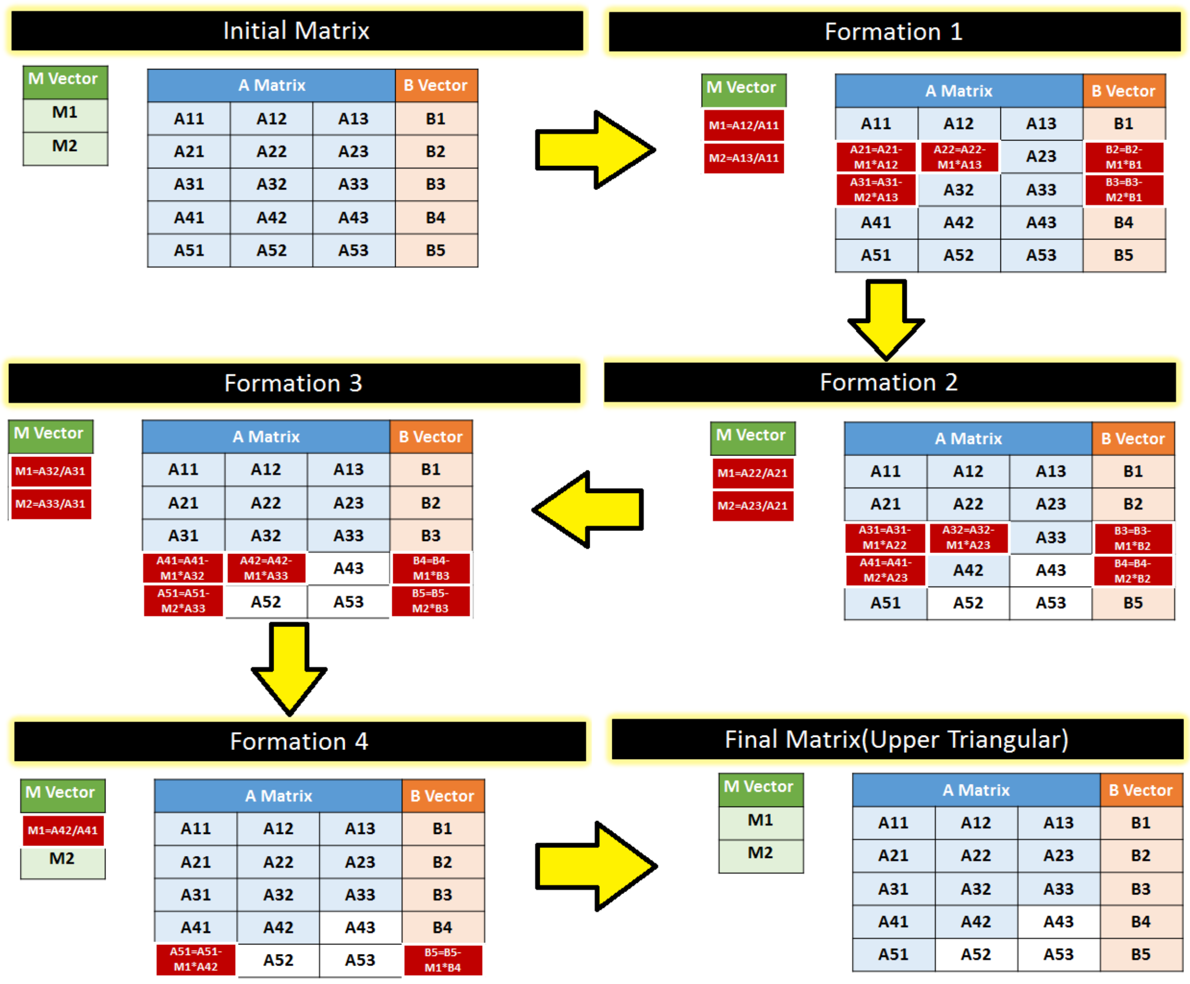


Figure 5.2: Formation of Upper Triangular matrix

5.3 Sequential Program

In this section sequential code of Half Band matrix solver is given. Code is divided into two parts. Part 1 convert half band matrix to upper triangular matrix and Part 2 takes care of back substitution.

5.3.1 Generation of Upper triangular Matrix

Code to generate upper triangular matrix from Half Band is given below.

```
void band(float *A, float *B, int NN, int MM)
{
//triangularise and reduce right hand side
int NL, NM, MR, N, L, K, i, j;
float BN, C;
NL = NN - MM + 1;
NM = NN - 1;
MR = MM;
for (int N = 1; N <= NM; N++)
{
if (A[N*(MM)+1] == 0)
{
fprintf(out, "ZERO OR NEGATIVE ELEMENT ON MAIN DIAGONAL OF \n");
fprintf(in, "TRIANGULARIZED MATRIX FOR EQUATION %d ", N);
}
BN = B[N];
B[N] = BN / A[N*MM+1];
if (N>NL)
MR = NN - N + 1;
for (int L = 2; L <= MR; L++)
{
if (A[N*MM+L] == 0)
continue;
C = A[N*MM + L] / A[N*MM + 1];
i = N + L - 1;
j = 0;
for (int K = L; K <= MR; K++)
{
j = j + 1;
A[i*MM + j] = A[i*MM + j] - C*A[N*MM + K];
}
B[i] = B[i] - C*BN;
A[N*MM + L] = C;
}
}
```

5.3.2 Back Substitution

Code for Back substitution process is given below.

```
//Back Substitute
i = NN;
B[NN] = B[NN] / A[NN*MM + 1];
for (int N = 1; N <= NM; N++)
{
i = i - 1;
if (N<MM)
MR = N + 1;
for (int j = 2; j <= MR; j++)
{
```

```
K = i + j - 1;
B[i] = B[i] - A[i*MM + j] * B[K];
}
}
}
```

5.4 Parallel Program

In this section parallel code of Half Band matrix solver is given. Code is divided into two parts. Part 1 convert half band matrix to upper triangular matrix in parallel and Part 2 takes care of back substitution in sequence.

5.4.1 Generation of Upper triangular matrix

Code to generate upper triangular matrix from Half Band is given below.

```
//Kernel1 for Calculation of  M vector
__global__ void kernel1(float *A, float *M, int i,int BW)
{
int tid = blockIdx.x ;
M[tid] = A[(i - 1)*(BW + 1) + tid + 1] / A[(i - 1)*(BW + 1)];

}

//kernel 2 for update A matrix
__global__ void kernel2(float *A, float *M,int i,int BW)
{
int tid = blockIdx.x + gridDim.x*blockIdx.y;
int index = (BW+1)*i + tid;
int row_id = tid / (BW + 1);
int col_id = tid % (BW + 1);
int max = BW - 1;
int min = (BW - 1) - row_id;
if (col_id<min || col_id>max)
{
int s_index;
if (col_id == BW)
{
s_index = (i - 1)*(BW + 1) + col_id;
}
else
{
s_index= (i - 1)*(BW + 1) + col_id + row_id + 1;
}
A[index] = A[index]-M[row_id]*A[s_index];
}
}
```

5.4.2 Back Substitution

Code for Back substitution process is given below. Back substitution is sequential process so it's parallel code remain same as sequential code because it can't be solved in parallel.

```
//Back Substitute
i = NN;
B[NN] = B[NN] / A[NN*MM + 1];
for (int N = 1; N <= NM; N++)
```

```
{
i = i - 1;
if (N<MM)
MR = N + 1;
for (int j = 2; j <= MR; j++)
{
K = i + j - 1;
B[i] = B[i] - A[i*MM + j] * B[K];
}
}
}
```

5.5 Results

For testing of Half Band program, A separate program of 3 node axially loaded bar element is used for generation of input data. As we can see in table 5.1 up to 10000 number of equation performance of sequential program is better than parallel program. Reason for poor performance of parallel program is that number of parallel calculation in Half Band solver depend on size of Band with and for 3 node bar element band width is only 3.

5.5.1 Comparison of Gaussian Elimination and Half Band Program

In Table 5.2 comparison of Sequential time, Communication time and execution time for parallel program and speed up factor is given. In this table terms like GE and HB are used instead of Gaussian Elimination and Half Band method.

Table 5.1: Execution Time of Half Band

Number of Equations	**CPU (ms)**	**Communication Time (ms)**	**Execution time (ms)**	**Total time(ms)**
100	3	0.0477	11.0386	11.0863
200	4	0.0761	20.3266	20.4027
300	6	0.0928	33.8397	33.9325
400	8	0.0798	53.7541	53.8339
500	18	0.0733	64.2443	64.3176
600	22	0.0727	66.8758	66.9485
700	27	0.0762	86.6793	86.7555
800	28	0.0783	96.3533	96.4316
900	30	0.0937	104.8847	104.9785
1000	37	0.0686	128.2848	128.3534
2000	41	0.0960	262.6367	262.7327
3000	43	0.0974	443.4242	443.5216
4000	61	0.1460	611.2761	611.4221
5000	72	0.2120	841.1025	841.3145
6000	90	0.1820	1071.1183	1071.3003
7000	104	0.2126	1357.6125	1357.8252
8000	110	0.1638	1651.3832	1651.5470
9000	115	0.1840	1989.2682	1989.4522
10000	125	0.2256	2334.5437	2334.7693

As per Figure 5.3 we can say that for smaller size of Band with sequential program is much faster than parallel program.For band width size 3, number of parallel calculation will be 3+2+1=5 which means only 5 number of elements are calculated in parallel which leads to poor parallel performance. To test true performance of Half Band Solver on GPU it is important that Size of Band width must be large.

As per Figure 5.4 it is clear that for smaller band width, Half Band Matrix Program clearly outperform Gaussian Elimination program. With increase in size of Band width this difference will become smaller.

In Figure 5.5 comparison of parallel program based on Gaussian Elimination method

Table 5.2: Comparison:Execution Time of Gaussian Elimination(GE) and Half Band(HB)

Number of Equations	**CPU (ms)**	**Communication Time (ms)**	**Execution time (ms)**	**Total time(ms)**	**SpeedUp Factor**
100(GE)	16	1	75	76	0.2105
100(HB)	3	0.0477	11.0386	11.0863	0.2706
200(GE)	65	1	255	256	0.2539
200(HB)	4	0.0761	20.3266	20.4027	0.1961
300(GE)	150	1	477	478	0.3138
300(HB)	6	0.0928	33.8397	33.9325	0.1768
400(GE)	290	2	652	654	0.4434
400(HB)	8	0.0798	53.7541	53.8339	0.1486
500(GE)	503	2	770	772	0.6516
500(HB)	18	0.0733	64.2443	64.3176	0.2799
600(GE)	740	2	992	994	0.7445
600(HB)	22	0.0727	66.8758	66.9485	0.3286
700(GE)	1100	2	1366	1368	0.8041
700(HB)	27	0.0762	86.6793	86.7555	0.3112
800(GE)	1550	3	1897	1900	0.8158
800(HB)	28	0.0783	96.3533	96.4316	0.2904
900(GE)	2110	3	2485	2488	0.8481
900(HB)	30	0.0937	104.8847	104.9785	0.2858
1000(GE)	2764	4	3159	3163	0.8739
1000(HB)	37	0.0686	128.2848	128.3534	0.2883

and Half Band Method is done. Same as sequential program the difference of execution time between both program is high but in case of parallel program difference between execution time is lower than sequential program.

In Figure 5.6 comparison of communication time between Gaussian elimination parallel program and Half Band Parallel program is done. From the figure it is clearly evident that for Half Band requirement of storage is less compared to Gaussian elimination method, thus communication time is low.

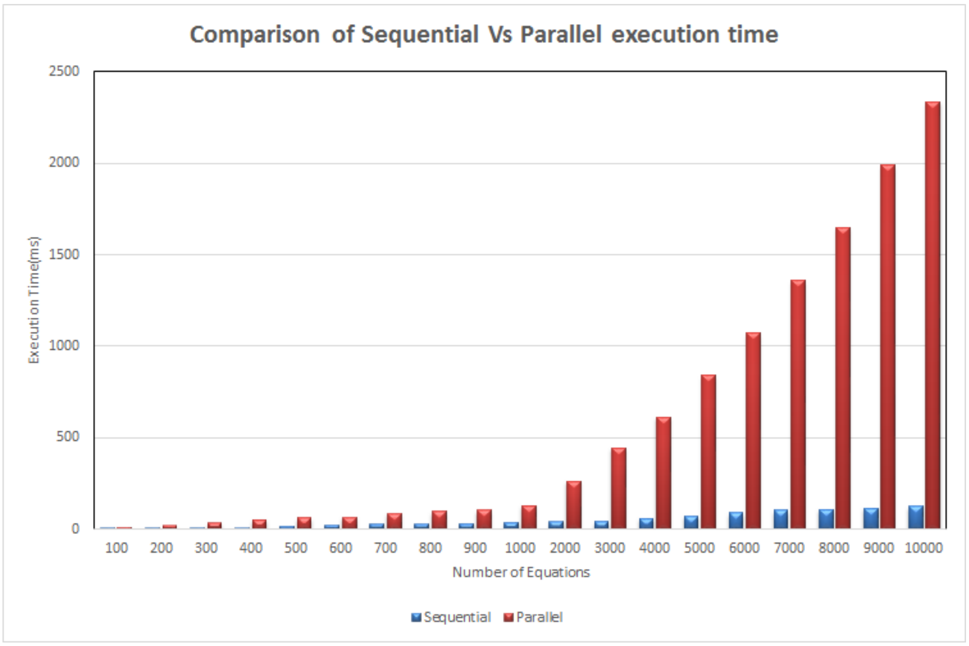


Figure 5.3: Comparison of sequential and parallel Execution time Half Band Matrix

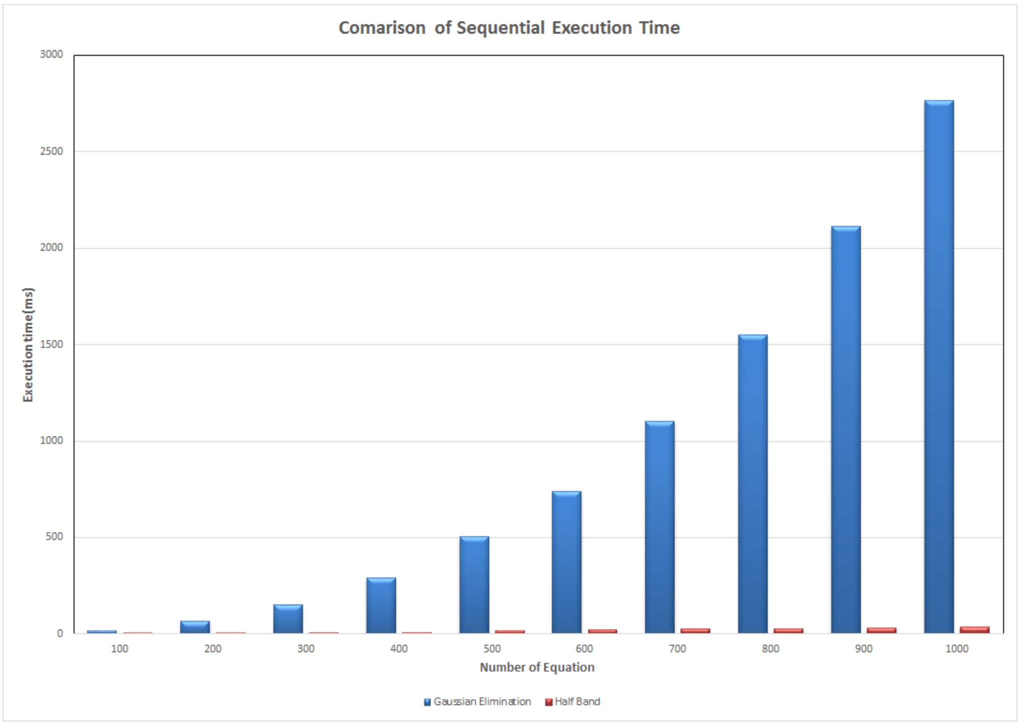


Figure 5.4: Comparison of sequential Execution time between

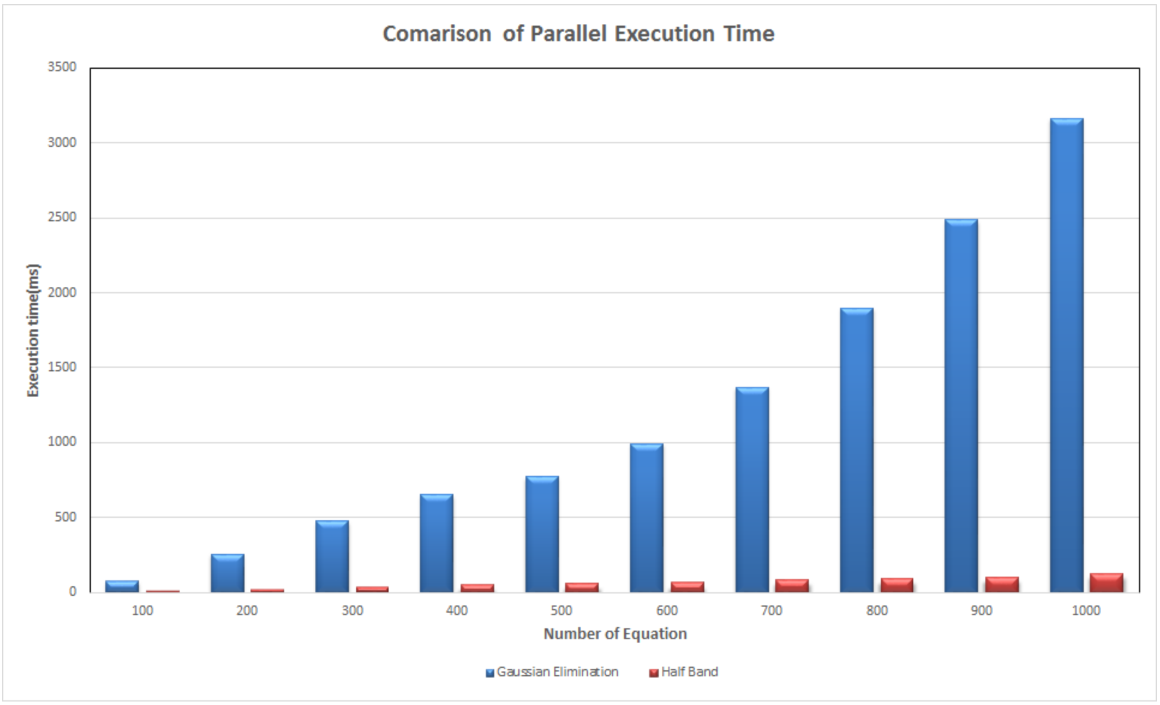


Figure 5.5: Comparison of parallel execution time

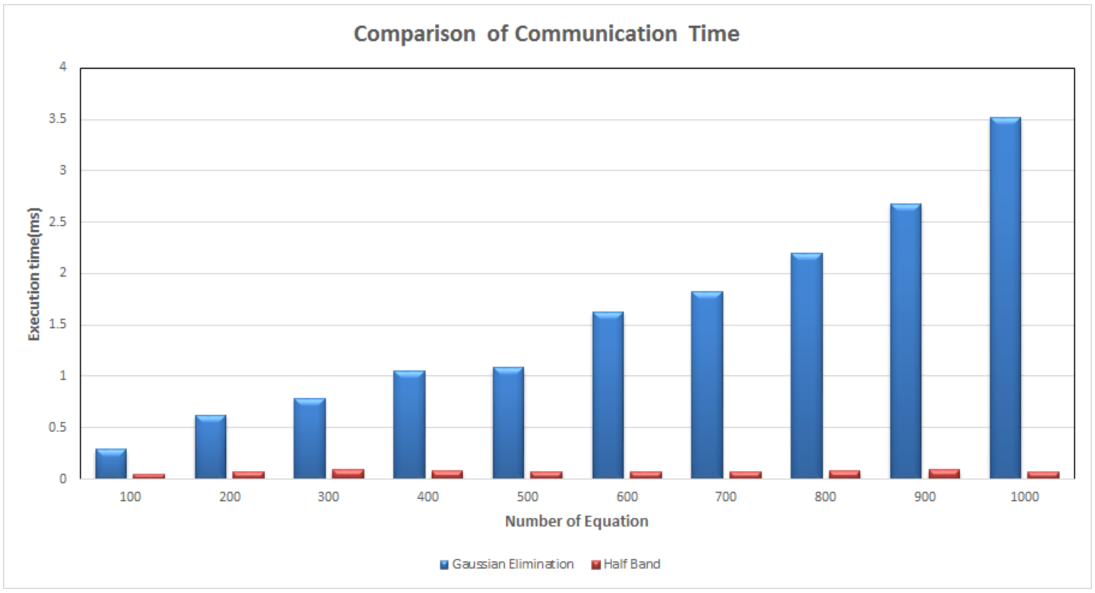


Figure 5.6: Comparison of communication time

5.6 Summary

In this chapter Storage of Stiffness matrix to Half Band Matrix is discribed with example. Program Based on Half Band Matrix for Sequential and parallel program Developed and implemented on CPU and GPU respectively. Results of Both programs are presented and comparison is done based on Execution time of program. In end comparison of Execution time of Gaussian Elimination and Half Band Matrix is given.

Chapter 6

Finite Element Analysis Using Parallel Programming

6.1 General

In Finite Element analysis Matrix Multiplication and Solution of linear equation are most computational intensive part. Execution of Computational intensive part in parallel will lead to lower execution time. In this project program is made to run Finite Element Analysis of Cantilever beam using CST Element in parallel. Sequential program is prepared using C language and implemented on CPU. Parallel program is prepared using NVIDIA CUDA C and implemented on GPU. For parallel program in FEM Analysis computational intensive part like Calculation of B Matrix , K Matrix , Calculation of Displacement Vector and calculation stress are replaced with parallel program.

6.2 Algorithm

As we can see in Figure 6.1, For simplicity FEM Analysis of Cantilever beam with point load at end is done to explain concept of parallel programming in FEM. Reason for choosing cantilever beam is that we can easily calculate max displacement from available formula to cross check results given by program.

6.2.1 Problem details

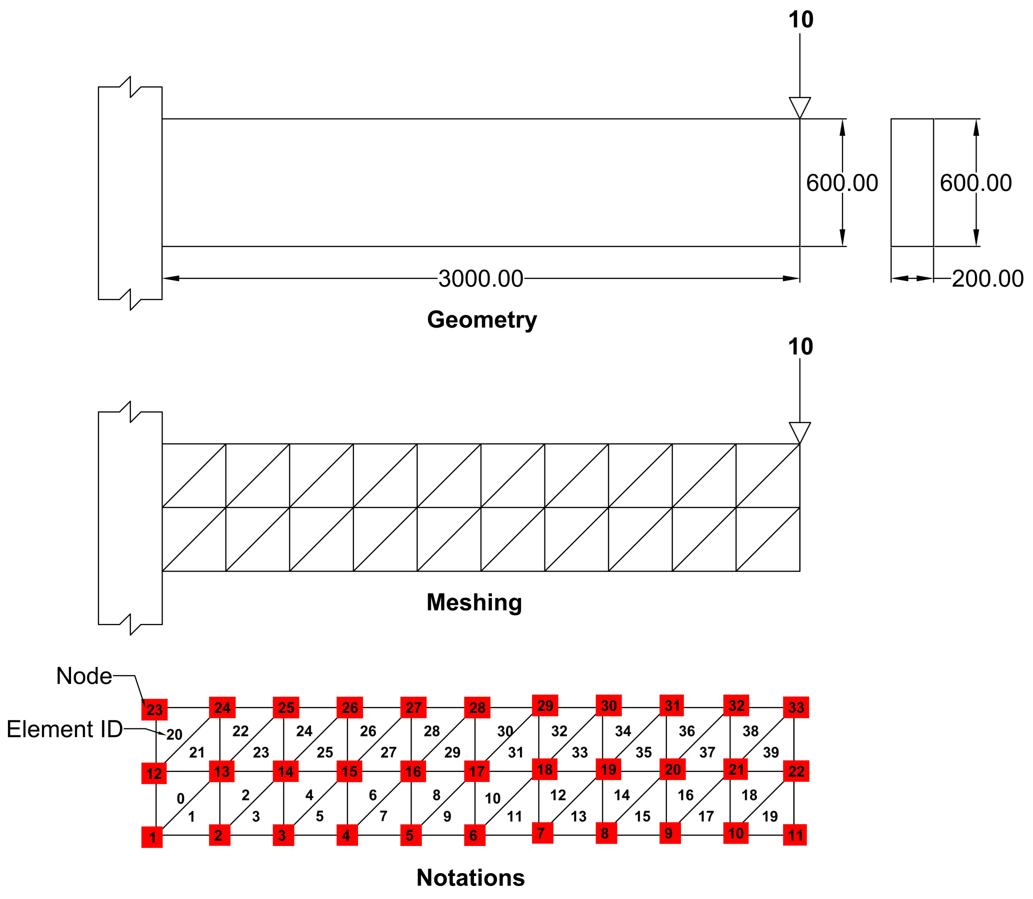
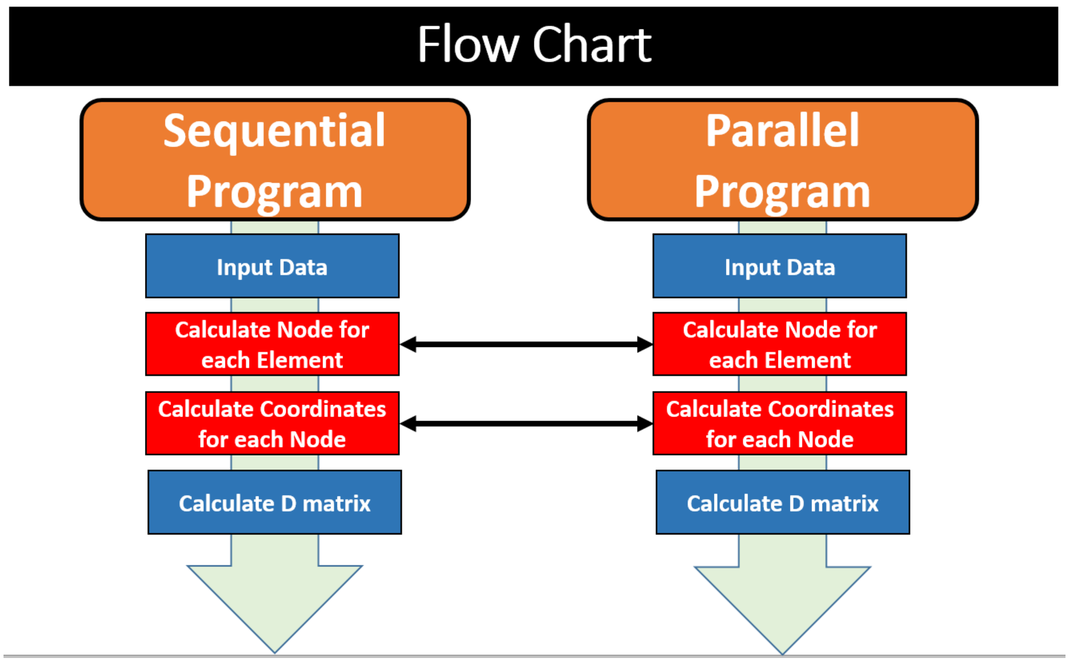


Figure 6.1: Problem details

As Input parameters, Program will take geometry, element along length and depth for meshing, material property and point load at end. To reduce complexity of problem program is developed such a way that it generate boundary condition for cantilever beam by itself. So, if we want to use this program for another type of beam than we just need to take care of boundary conditions in program. Type of element is fixed in this program and all nodal points for cantilever beam are calculated as per CST element.

Example of meshing for 10 parts along length and 2 parts along depth is shown in Figure 6.1. In last picture how program will assign Element ID and nodes to each element is shown.

6.2.2 Flow of program


Flow Chart
Sequential Program
Parallel Program
Input Data
Calculate Node for each Element
Calculate Coordinates for each Node
Calculate D matrix
Input Data
Calculate Node for each Element
Calculate Coordinates for each Node
Calculate D matrix

In this section, main difference between Sequential and Parallel program is given. In below figure parts with blue colour is calculated in sequence in both program. Part with Red color is calculated in sequence for sequential program and parallel for parallel program.

6.3 Results

From Table 6.1 it is evident that for Number of elements more than 10000 parallel program will give better results compared to sequential Program. Figure 6.2 also explain same thing in graphical format.

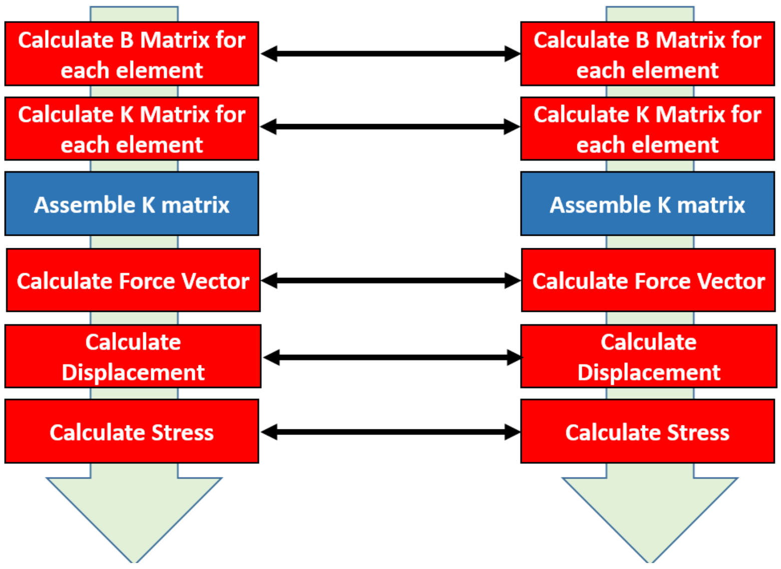


Table 6.1: Execution Time of FE Analysis Program

No	Elements along length	Elements along width	Number of Elements	Number of Nodes	DOF	CPU	GPU	Speed Up
1	5	1	10	12	24	15	561	0.0267
2	10	2	40	33	66	48	598	0.0802
3	20	4	160	105	210	114	651	0.1751
4	50	10	1000	561	1122	906	1386	0.6536
5	160	32	10240	5313	10626	57926	55927	1.0357

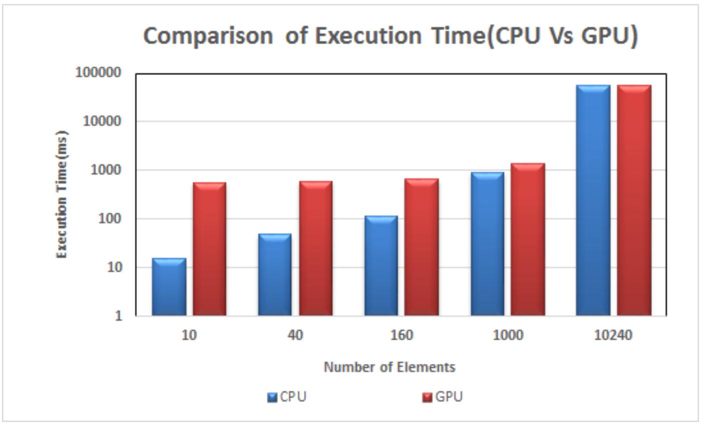


Figure 6.2: Comparison of Execution time CPU Vs GPU

6.4 Summary

In this chapter Finite Element based program for calculation of Cantilever beam is presented. For Sequential program c language is used and implemented on CPU. Parallel Program CUDA c lanuage is used and implemented on GPU. Comparison of both program is done based on Execution time and Speed Up factor and presented.

Chapter 7

Summary and Conclusion

7.1 Summary

Solution of linear equation is a computational intensive process in analysis of structural system. With increase in size of problems more linear equations need to be solved which increases execution time of structural analysis dramatically.In this project the concept of parallel programming is used in Finite Element Analysis of structure using NVIDIA GPU as Hardware. Parallel programming on GPU is implemented using CUDA C language and than compared with sequential program.

In present study computation intensive part of Finite Element analysis like Matrix multiplication and Solution of linear equation is developed using concept of parallel programming. The concept of parallel matrix multiplication and solution of linear equation is implemented in Finite Element Analysis using triangular element for cantilever beam.

For Matrix Multiplication Program is tested for different sizes of matrix up to 1500$\times$1500. Sequential program is developed in C language and parallel program is developed in CUDA C. Sequential and Parallel Program compared based on execution time and speed up Factor.

Gaussian Elimination program is developed for solution of linear system of equation like [A][x]=[b] which is quite similar to [K][d]=[F] linear equations generated in Finite Element Analysis. For generation of input Axially loaded bar using 3 node finite element program is developed.Sequential program is developed in C language and parallel program is developed in CUDA C.Program is tested up to 1000 numbers of equations and compared based on Execution time and speed up factor.

Half Band solution of equations is improved version of Gaussian Elimination solution technique in terms of Storage and calculations. Half Band Program is developed to solve [A][x]=[b] type of linear system, where [A]=Stiffness matrix in Half Band form ,[x]=displacement vector and [b]=Force vector.Sequential program is developed in C language and parallel program is developed in CUDA C. For generation of input Axially loaded bar using 3 node finite element program is developed.Program is tested up to 10000 number of equations and compared based on Execution time and speed up factor.

Parallel Finite Element program for stress analysis of Cantilever beam is presented. For Sequential program C language is used and implemented on CPU. Parallel Program is developed using CUDA C lanuage and implemented on GPU. Comparison of both program is done based on Execution time and Speed Up factor and presented.For 10000 number of equation Speed Up factor is higher than 1 means that for more than 10000 number of element parallel program can give better performance than sequential program.

7.2 Conclusion

Based on study carried out in this project following conclusions are drawn.

- Performance of Parallel program depend on type of numerical algorithm adopted.
- Performance of parallel and sequential program also depends on Hardware platform used for implementation.
- In Finite Element analysis by parallelizing computational intensive part like Matrix multiplication and Solution of linear equation significant reduction in total execution time can be achieved.
- Speed Up factor which is ratio of Sequential time to Parallel time improve significantly with increase in size of problem and increase in parallel part of program.
- Parallelization of entire Program instead of only equation solver leads to higher computational efficiency.
- From Parallel implementation of Gaussian Elimination:
  - Maximum speed up 0.87 achieved for solving 1000 number of equation .
  - It is evident that speed up factor for Gaussian Elimination increase with increase in number of equations.
- From Parallel implementation of Half Band equation Solver
  - Speed up factor is depend upon size of Band width not number of equation.
  - For smaller size of band width, Half Band Program clearly out perform Gaussian elimination program in terms of execution time.
  - Maximum speed up of 0.28 for Half Band program compared to 0.87 for gaussian elimination program it is evident that for smaller band width this method is not suitable for parallel computing.

- From Parallel implementation of Finite Element Method
  - Maximum speed up of 1.03 is achieved in this study when number of equation is 10000 which means parallel program will perform better for more than 10000 number of equation.
  - Speed up factor of Finite element program depend on number of equation and type of solver used in program.

7.3 Future Scope of Work

The study carried in this project can be extended to include following aspects:

- Performance comparison between OpenCL and CUDA platform.
- Development of efficient algorithm to suit different hardware platforms.
- Development of Computer program for dynamic analysis using parallel programming on GPU.
- Various structural engineering problems like FEM analysis using numerical integration, non-linear dynamic analysis.

Appendix A

Matrix Multiplication

A.1 Sequential Program

```
//Matrix Multiplication in CPU
#include<stdio.h>
#include<time.h>

//BLOCK_SIZE is size Multiplier for Square matrix
#define BLOCK_SIZE  1

void  main( )
{
int N, K;
K = 100;
N = K*BLOCK_SIZE;//N=size of square Matrix
float *A, *B, *C;//A*B=C

//for measurement of execution time
clock_t begin, end;
```

```
double ExecTime;

//Output file
FILE *out;

//starting of Matrix Multiplication
begin = clock();

out = fopen("output.txt", "w");

printf("Executing Matrix Multiplcation");
printf("\nMatrix size: %d\n", N);

A = new float[N*N];
B = new float[N*N];
C = new float[N*N];

// Initialize matrices on the host
for (int row = 0; row<N; row++)
{
for (int col= 0; col<N; col++)
{
A[row*N + col] = row*col;
B[row*N + col] = -row*col;
}
}
// Print A and B matrix
fprintf(out, "Matrix A\n");
for (int row = 0; row<N; row++)
```

```
{
for (int col = 0; col<N; col++)
{
fprintf(out, "%f\t", A[row*N + col]);
}
fprintf(out,"\n");
}

fprintf(out, "Matrix B\n");
for (int row = 0; row<N; row++)
{
for (int col = 0; col<N; col++)
{
fprintf(out, "%f\t", B[row*N + col]);
}
fprintf(out, "\n");
}



// Now do the matrix multiplication on the CPU
float sum;
for (int row = 0; row<N; row++){
for (int col = 0; col<N; col++){
sum = 0;
for (int n = 0; n<N; n++){
sum += A[row*N + n] * B[n*N + col];
}
C[row*N + col] = sum;
}
}

fprintf(out, "Matrix C\n");
for (int row = 0; row<N; row++)
{
for (int col = 0; col<N; col++)
{
fprintf(out, "%f\t", C[row*N + col]);
}
fprintf(out, "\n");
}

//end of Matrix Multiplication
end = clock();
ExecTime = end - begin;
   printf("\nExecution Time=%f",ExecTime);
fprintf(out, "\ntime spent=%f",ExecTime);

}
```

A.2 Parallel Program

```
//Program for Matrix multiplication on GPU
#include<cuda.h>
#include<cuda_runtime.h>
#include<stdio.h>
#include<time.h>
```

```
//define dimension of block
#define BLOCK_SIZE 1  //Block_size=Number of threads per block

//kernal for Matrix Multiplication
__global__ void gpuMM(float *A, float *B, float *C, int N)
{
// Matrix multiplication for NxN matrices C=A*B
// Each thread computes a single element of C
int row = blockIdx.y*blockDim.y + threadIdx.y;
int col = blockIdx.x*blockDim.x + threadIdx.x;

float sum = 0;

for (int n = 0; n < N; ++n)
sum += A[row*N + n] * B[n*N + col];

C[row*N + col] = sum;
}

int main( )
{
// where A, B and C are NxN matrices on Device
//hA,hB and hC are matrices on Host(CPU)
//dA,dB and dC are for Device(GPU)

int N, K;  //K=size of Grid in 2d
K = 100;
N = K*BLOCK_SIZE; // N=Size of square matrix for 100*100 size of Matrix N=100
```

```
clock_t begin, end; //for mesurement of execution time
double dt;

FILE *out; // output file
out = fopen("output.txt", "w");

//begin time of Matrix Multiplication
begin = clock();

printf("Executing Matrix Multiplcation");
printf("\nMatrix size: %d", N);



// Allocate memory on the host
float *hA, *hB;

hA = new float[N*N];
hB = new float[N*N];



// Initialize matrices on the host
for (int j = 0; j<N; j++){
for (int i = 0; i<N; i++){
hA[j*N + i] = i*j;
hB[j*N + i] = -i*j;
    }
}

//print Matrix A and B
```

```
fprintf(out,"\n A matrix \n");
for (int j = 0; j<N; j++){
for (int i = 0; i<N; i++){
fprintf(out, "%f\t", hA[j*N + i]);

}
fprintf(out, "\n");
}
fprintf(out, "\n B matrix \n");
for (int j = 0; j<N; j++){
for (int i = 0; i<N; i++){
fprintf(out, "%f\t", hB[j*N + i]);

}
fprintf(out, "\n");
}



// Allocate memory on the device
int size = N*N*sizeof(float);
float *dA, *dB, *dC;
cudaMalloc(&dA, size);
cudaMalloc(&dB, size);
cudaMalloc(&dC, size);

//specify size of Block and Thread
dim3 threadBlock(BLOCK_SIZE, BLOCK_SIZE);
dim3 grid(K, K);
```

```
// Copy matrices from the host to device
cudaMemcpy(dA, hA, size, cudaMemcpyHostToDevice);
cudaMemcpy(dB, hB, size, cudaMemcpyHostToDevice);

//Execute the matrix multiplication kernel

gpuMM <<< grid, threadBlock >>> (dA, dB, dC, N);



// Allocate memory to store the GPU answer on the host
float *hC;
hC = new float[N*N];

// Now copy the GPU result back to CPU
cudaMemcpy(hC, dC, size, cudaMemcpyDeviceToHost);
fprintf(out, "\n C matrix \n");
for (int j = 0; j<N; j++){
for (int i = 0; i<N; i++){
fprintf(out, "%f\t", hC[j*N + i]);

}
fprintf(out, "\n");
}
//end of Matrix Multiplication
end = clock();
dt = (end - begin);//total Execution time
printf("Execution time=%f\n",dt);
//fprintf(out, "\ntime spent=%f", dt);
}
```

Appendix B

Gauss Elimination

B.1 Sequential Program

```
//Program for Solution of Matrix based on Gaussian Elimination
#include<stdio.h>
#include<stdlib.h>
#include<time.h>

FILE *in, *out,*t;

int main()
{
int i, j, k,N;
clock_t begin, end;
double ExecTime;
begin = clock();
in = fopen("input.txt", "r");
out = fopen("output.txt", "w");
t = fopen("time.txt","w");
```

```
fscanf(in, "%d", &N);
fprintf(out,"Number of Equations=%d\n",N);
fprintf(t, "Number of Equations=%d\n", N);

float *A;
int size = (N)*(N+1);
A = (float *)malloc(size * sizeof(float));
int index=0;

fprintf(out, "\nAugmented Matrix\n");
for (i = 0; i < N; i++)
{

for (j = 0; j < (N + 1); j++)
{
index = i*(N + 1) + j;
fscanf(in,"%f", &A[i*(N+1)+j]);
fprintf(out, "%f\t", A[i*(N+1)+j]);
}
fprintf(out, "\n");

}
float c = 0;
// loop for the generation of upper triangular matrix
for (j = 0; j < N; j++)
{
fprintf(out, "\n\n****Formation of Upper trinagular matrix=%d ****\n", j + 1);
fprintf(out,"------M vector-----");
for (i = 0; i < N; i++)
{
if (i>j)
{
c = A[i*(N+1)+j] / A[j*(N+1)+j];
fprintf(out,"\n%f",c);
for (k = 0; k < N + 1; k++)
{
A[i*(N+1)+k] = A[i*(N+1)+k] - c*A[j*(N+1)+k];
}
}
}

fprintf(out, "\n------A Matrix-----");
for (int l = 0; l < N; l++)
{
fprintf(out, "\n");
for (int m = 0; m < (N + 1); m++)
{
fprintf(out, "%f\t", A[l*(N + 1) + m]);
}



}
}

fprintf(out, "\n\n******  final Formation of Upper trinagular matrix\n");
```

```
for (i = 0; i < N; i++)
{

for (j = 0; j < (N + 1); j++)
{
fprintf(out, "%f\t", A[i*(N + 1) + j]);
}
fprintf(out, "\n");

}
float *x;
x = (float *)malloc(N * sizeof(float));
x[N-  1] = A[N*(N + 1)-1] / A[N*(N+1)-2];
fprintf(out, "X[%d]=%f", N - 1, x[N - 1]);
//this loop is for backward substitution
float sum;
for (int i = N - 2; i >= 0; i--)
{
sum = 0;
for (int j = i + 1; j < N; j++)
{
sum = sum + A[i*(N + 1) + j] * x[j];
}
x[i] = (A[i*(N + 1) + N] - sum) / A[i*(N + 1) + i];
}

fprintf(out, "\nSolution");
for (int i = 0; i < N; i++)
{
```

```
fprintf(out, "\nx%d=%f\t", i, x[i]);
}

end = clock();
ExecTime = end - begin;
fprintf(t,"\nExecution Time=%f", ExecTime);
return(0);
}
```

B.2 Parallel Program

```
//Program for Gaussian Elimination on GPU

#include<cuda.h>
#include<cuda_runtime.h>
#include<stdio.h>
#include<time.h>

FILE *in, *out,*t;

// Kernel-1 for Generation of m vector
__global__ void k1(float *A,float *M,float c,int i,int N)
{
int tid = blockIdx.x + gridDim.x*blockIdx.y;
M[tid] = A[(tid+i)*(N+1)+i-1]/c;
}

// Kernel-2 for Update A Matrix
```

```
__global__ void k2(float *A,float *M,int i,int N)
{
int tid = blockIdx.x + gridDim.x*blockIdx.y;
int row_id = tid / (N-i + 2);
int col_id = tid % (N-i + 2);
int index = 0;
if (i == 1)
{
index = i*(N + 2) - 1 + tid;
}
else
{
index = i*(N + 2) - 1 + tid+i*row_id-row_id;

}
int s_index = col_id + (i-1)*(N+2);
A[index] = A[index] - M[row_id] * A[s_index];

}

int main()
{
int N;
//Allocation of Variable on Host
//float A[N*(N + 1)],M[N-1],x[N];
clock_t begin, end;
double ExecTime;
begin = clock();
t = fopen("time.txt", "w");
```

```
//file input-Output
in = fopen("G4000.txt", "r");
out = fopen("output.txt", "w");

fscanf(in, "%d", &N);
fprintf(out, "Number of Equations=%d\n", N);

float *A;
int hsize = (N)*(N + 1);
A = (float *)malloc(hsize * sizeof(float));

float *M;
M = (float *)malloc((N-1) * sizeof(float));

//scan augmented matrix from input file on CPU
fprintf(out,"Scanned Matrix from input file\n");
for (int i = 0; i < N; i++)
{
for (int j = 0; j < N + 1; j++)
{
fscanf(in, "%f", &A[i*(N + 1) + j]);
//fprintf(out, "%f\t", A[i*(N + 1) + j]);
}
//fprintf(out, "\n");
}

//Allocation of Memory on GPU
int size = N*(N + 1)*sizeof(float);
int msize = (N - 1)*sizeof(float);
float *dA;
float *dM;
cudaMalloc(&dA,size);
cudaMalloc(&dM,msize);

//Specify Size of Block and Thread
dim3 grid(4000,4000);
dim3 threadBlock(1,1);

//Copy Augmented matrix from host to device
cudaMemcpy(dA,A,size,cudaMemcpyHostToDevice);
float c;
for (int i = 1; i < N; i++)
{
//fprintf(out, "\n-----Loop=%d-------", i);
c = A[(i-1)*(N+1)+i-1];
//fprintf(out, "\n C=%f\n",c);
k1 << <(N - i), 1 >> >(dA,dM,c,i,N);

cudaMemcpy(M, dM, msize, cudaMemcpyDeviceToHost);
/*fprintf(out, " \nM vector\n");
for (int l = 0; l < N - 1; l++)
{
// fprintf(out, "%f\t", M[l]);
} */



k2 << < ((N-i)*(N+2-i)),1>> >(dA,dM,i,N);
```

```
//copy result back to CPU
cudaMemcpy(A, dA, size, cudaMemcpyDeviceToHost);


/*print updated A matrix
fprintf(out, "\n\n Updated A matrix\n");
for (int l = 0; l < N; l++)
{
for (int o = 0; o< N + 1; o++)
{
fprintf(out, "%f\t", A[l*(N + 1) + o]);
}
fprintf(out, "\n");
} */


}
float *x;
x = (float *)malloc(N * sizeof(float));
x[N - 1] = A[N*(N + 1) - 1] / A[N*(N + 1) - 2];
fprintf(out, "X[%d]=%f",N-1, x[N-1]);
//this loop is for backward substitution
float sum;
for (int i = N - 2; i >=0; i--)
{
sum = 0;
for (int j = i+1 ; j < N; j++)
{
sum = sum + A[i*(N+1)+j] * x[j];
}
x[i] = (A[i*(N+1)+N] - sum) / A[i*(N+1)+i];
```

```
}



fprintf(out, "\nSolution");
for (int i = 0; i < N; i++)
{
fprintf(out, "\nx%d=%f\t", i, x[i]);
}
end = clock();
ExecTime = end - begin;
fprintf(t, "\nExecution Time=%f", ExecTime);
fclose(in);
fclose(out);
fclose(t);
return 0;

}
```

Appendix C

Half Band

C.1 Sequential Program

```
//computer program for band solver
#include<stdio.h>
#include<conio.h>
#include<math.h>
#include<stdlib.h>
#include<time.h>

FILE *in, *out,*t;
void band(float *A, float *B, int NN, int MM);
int main()
{
//float A[30][10], B[20], temp;
int NN, MM;
clock_t begin, end;
double ExecTime;
begin = clock();
```

```
in = fopen("input.txt", "r");

t = fopen("time.txt", "w");
out = fopen("output.txt", "w");

fscanf(in, "%d ", &NN);
MM = 8;
float *A;
A = (float *)malloc(8*NN * sizeof(float));
float *B;
B = (float *)malloc( NN * sizeof(float));

fprintf(out, "No of equations =%d \n Band width %d\n", NN, MM);
//fprintf(out, "\n A matrix");
for (int i = 1; i <= NN; i++)
{
for (int j = 1; j <= MM; j++)
{
fscanf(in, "%f", &A[i*MM+j]);

//fprintf(out, "%f\t", A[i*MM + j]);
}
fscanf(in, "%f", &B[i]);
//fprintf(out, "%f \n", B[i]);
}

band(A, B, NN, MM);
fprintf(out, "B[%d]=%f\n", NN, B[NN]);
fprintf(out, "\nsolution Vector\n");

for (int i = 1; i <= NN; i++)
{
fprintf(out, "B[%d]=%f\n",i, B[i]);
}
end = clock();
ExecTime = end - begin;
//fprintf(t, "\nExecution Time=%f", ExecTime);



}
void band(float *A, float *B, int NN, int MM)
{
//triangularise and reduce right hand side
int NL, NM, MR, N, L, K, i, j;
float BN, C;
NL = NN - MM + 1;
NM = NN - 1;
MR = MM;
for (int N = 1; N <= NM; N++)
{
if (A[N*(MM)+1] == 0)
{
fprintf(out, "ZERO OR NEGATIVE ELEMENT ON MAIN DIAGONAL OF \n");
fprintf(in, "TRIANGULARIZED MATRIX FOR EQUATION %d ", N);
}
BN = B[N];
B[N] = BN / A[N*MM+1];
```

```
if (N>NL)
MR = NN - N + 1;
for (int L = 2; L <= MR; L++)
{
if (A[N*MM+L] == 0)
continue;
C = A[N*MM + L] / A[N*MM + 1];
i = N + L - 1;
j = 0;
for (int K = L; K <= MR; K++)
{
j = j + 1;
A[i*MM + j] = A[i*MM + j] - C*A[N*MM + K];
}
B[i] = B[i] - C*BN;
A[N*MM + L] = C;
}
}
//Back Substitute
i = NN;
B[NN] = B[NN] / A[NN*MM + 1];
for (int N = 1; N <= NM; N++)
{
i = i - 1;
if (N<MM)
MR = N + 1;
for (int j = 2; j <= MR; j++)
{
K = i + j - 1;
B[i] = B[i] - A[i*MM + j] * B[K];
}




}
}
```

C.2 Parallel Program

```
//Program for Gaussian Elimination on GPU

#include<cuda.h>
#include<cuda_runtime.h>
#include<stdio.h>
#include<time.h>

//Number of equation

FILE *in, *out, *t;

//Kernel1 for Calculation of  M vector
__global__ void kernel1(float *A, float *M, int i,int BW)
{
int tid = blockIdx.x ;
M[tid] = A[(i - 1)*(BW + 1) + tid + 1] / A[(i - 1)*(BW + 1)];

}
```

```
//kernel 2 for update A matrix
__global__ void kernel2(float *A, float *M,int i,int BW)
{
int tid = blockIdx.x + gridDim.x*blockIdx.y;
int index = (BW+1)*i + tid;
int row_id = tid / (BW + 1);
int col_id = tid % (BW + 1);
int max = BW - 1;
int min = (BW - 1) - row_id;
if (col_id<min || col_id>max)
{
int s_index;
if (col_id == BW)
{
s_index = (i - 1)*(BW + 1) + col_id;
}
else
{
s_index= (i - 1)*(BW + 1) + col_id + row_id + 1;
}
A[index] = A[index]-M[row_id]*A[s_index];
}
}

int main()
{
int N, BW;
//Allocation of Variable on Host
```

```
//float A[N*(BW + 1)], M[BW - 1],x[N];
clock_t begin, end;
double ExecTime;
begin = clock();

//file input-Output
in = fopen("HB10000.txt", "r");
out = fopen("output.txt", "w");
t = fopen("time.txt", "w");
fscanf(in,"%d",&N);
BW = 3;

float *A;
int hsize = (N)*(BW + 1);
A = (float *)malloc(hsize * sizeof(float));

float *M;
M = (float *)malloc((BW - 1) * sizeof(float));

//scan augmented matrix from input file on CPU
//fprintf(out, "Scanned Matrix from input file\n");
for (int i = 0; i < N; i++)
{
for (int j = 0; j < BW+1; j++)
{
fscanf(in, "%f", &A[i*(BW + 1) + j]);
// fprintf(out, "%f\t", A[i*(BW + 1) + j]);
}
//fprintf(out, "\n");
```

```
}

//Allocation of Memory on GPU
int size = N*(BW + 1)*sizeof(float);
int msize = (BW - 1)*sizeof(float);
float *dA;
float *dM;
cudaMalloc(&dA, size);
cudaMalloc(&dM, msize);

//Specify Size of Block and Thread
dim3 Blocks(3, 4, 1);
dim3 threadsPerBlock(1, 1, 1);

//Copy Augmented matrix from host to device
cudaMemcpy(dA, A, size, cudaMemcpyHostToDevice);


for (int i = 1; i <N; i++)
{
kernel1 << < BW-1, 1 >> >(dA, dM, i,BW);
kernel2 << < (BW-1)*(BW+1), 1 >> >(dA, dM, i,BW);


cudaMemcpy(M, dM, msize, cudaMemcpyDeviceToHost);
/*
fprintf(out, "**********************Formation =%d***************************",i)
fprintf(out, " \nM vector\n");
for (int i = 0; i < BW - 1; i++)
```

```
{
fprintf(out, "%f\t", M[i]);
}  */
cudaMemcpy(A, dA, size, cudaMemcpyDeviceToHost);
/*
fprintf(out, "\nUpdated Matrix from input file\n");
for (int i = 0; i < N; i++)
{
for (int j = 0; j < BW + 1; j++)
{
fscanf(in, "%f", &A[i*(BW + 1) + j]);
fprintf(out, "%f\t", A[i*(BW + 1) + j]);
}
fprintf(out, "\n");
}  */
}

//fprintf(out, "***********Back Substitution**************\n");
float *x;
x = (float *)malloc(N * sizeof(float));
x[N - 1] = A[N*(BW + 1) - 1] / A[N*(BW + 1) - (BW+1)];
fprintf(out, "\nX[%d]=%f\n", N - 1, x[N - 1]);

//this loop is for backward substitution
float sum;
for (int i = N - 2; i >= 0; i--)
{
sum = 0;
for (int j =  1; j < BW; j++)
```

```
{
if (A[i*(BW + 1) + j]!=0)
{
sum = sum + A[i*(BW + 1) + j] * x[i + j];
}

}
x[i] = (A[i*(BW + 1) + BW] - sum) / A[i*(BW + 1)];
}



fprintf(out, "\nSolution");
for (int i = 0; i < N; i++)
{
fprintf(out, "\nx%d=%f\t", i, x[i]);
}

end = clock();
ExecTime = end - begin;
fprintf(t, "\nExecution Time=%f", ExecTime);
fclose(in);
fclose(out);
fclose(t);
return 0;
}
```

Appendix D

Finite Element

D.1 Sequential Program

```
//**************Finite Element Analysis Based on CST Element*****************



#include<stdio.h>
//for dynamic array
#include<stdlib.h>
#include<conio.h>
#include<math.h>
//to measure Execution time
#include<time.h>

//Generate input-Output File
FILE *in, *out,*temp,*t;

char filename[15],file[15];
```

```
//Half Band Solver
void band(float *HBMatrix, float *ForceVector, int NN, int MM);

int main()
{

float Length,Depth,Thickness,ModulasOfElasticity,PoissonRatio,PointLoad;
int PartX,PartY,NumberOfElement,NumberOfNode,TypeofElement;

//clock data type to mesure time interval at different points
clock_t tstart, tinput,tnode,tcoordinate,tDmat,tBmat,tKmat,tHB,tForce,tdis;
double ExecTime;



//**************PART A: INPUT DATA ************************

tstart = clock();
//scan input file
in = fopen("input.dat","r");

fscanf(in, "%f %f %f", &Length, &Depth, &Thickness);
fscanf(in,"%d %d",&PartX,&PartY);
fscanf(in,"%f %f %d %f",&ModulasOfElasticity,&PoissonRatio,&TypeofElement,&Point

NumberOfElement = PartX*PartY * 2;
NumberOfNode = (PartX + 1)*(PartY+1);

//OutPut File
```

```
sprintf(filename, "out_%d.txt",NumberOfElement);
out = fopen(filename, "w");
temp = fopen("temp.txt","w");

//File to store execution time at different intervals
sprintf(file, "time_%d_%d.txt", PartX,PartY);
t = fopen(file,"w");

//Printing input data
fprintf(out,"*****input data*******\n");
fprintf(out, "Length=%.2f\nDepth=%.2f\nThickness=%.2f\n",Length,Depth,Thickness)
fprintf(out,"Elements along Length=%d\nElement along Depth=%d\n",PartX,PartY);
fprintf(out,"Modulas of Elasticity=%.2f\npoisson's Ratio=%.2f\nPoint Load=%.2f\n

if (TypeofElement == 0)
{
fprintf(out,"Type of Element = Plain stress Element");
}
if (TypeofElement == 1)
{
fprintf(out, "Type of Element = Plain strain Element");
}

tinput = clock();
ExecTime = tstart - tinput;
fprintf(t,"Input data=%f\n",ExecTime);
```

```
//*********PART B:GENERATION OF NODE FOR EACH ELEMENT*********

int *DataNode;
DataNode = (int *)calloc(NumberOfElement*3, sizeof(int));
int BlockID = 0;
int ColID = 0;

fprintf(out,"\n\n------Nodal Point Calculation----------");

for (int   i = 0; i < NumberOfElement; i++)
{
BlockID = (i/2)+1;
ColID = i / (PartX * 2);
DataNode[i*3] = BlockID+ColID;



if (i % 2 == 0)
{
DataNode[i * 3 + 1] = (PartX + 1) + BlockID + 1+ColID;
DataNode[i * 3 + 2] = DataNode[i * 3 + 1] - 1;
}
else
{
DataNode[i * 3 + 1] = DataNode[i * 3]+1;
DataNode[i * 3 + 2] = (PartX + 1) + BlockID + 1 + ColID;

}

fprintf(out, "\n %d\t%d\t%d", DataNode[i*3],DataNode[i*3+1],DataNode[i*3+2]);
```

```
}

tnode = clock();
ExecTime = tnode-tinput;
fprintf(t, "Node=%f\n", ExecTime);



//********PART C: Generation of joint coordinates for each node********

float *DataJoint;
DataJoint = (float *)calloc(NumberOfNode*2,sizeof(float));
float lengthX = 0;
float lengthY = 0;
lengthX = Length / PartX;
lengthY = Depth / PartY;
float CoordinateX = 0;
float CoordinateY = 0;

fprintf(out, "\n\n------joint coordinates Calculation----------");

for (int i = 0; i < NumberOfNode; i++)
{


DataJoint[i * 2] = CoordinateX;
DataJoint[i * 2 + 1] = CoordinateY;
CoordinateX = CoordinateX + lengthX;
```

```
if ((i + 1) % (PartX + 1) == 0)
{
CoordinateX = 0;
CoordinateY = CoordinateY + lengthY;

}



fprintf(out,"\n%d\t%f\t%f",i+1,DataJoint[i*2],DataJoint[i*2+1]);
}

tcoordinate = clock();
ExecTime = tcoordinate - tnode;
fprintf(t, "Coordinates=%f\n", ExecTime);




//*********PART D:Generation of [D] matrix for all elements**********
fprintf(out, "\n\n------Calculation of D Matrix----------");

float DMatrix[9];
float c;
if (TypeofElement == 0)
{
c =ModulasOfElasticity/(1-PoissonRatio*PoissonRatio);
DMatrix[0] = DMatrix[4] = c * 1;
DMatrix[1] = DMatrix[3] = c*PoissonRatio;
DMatrix[2] = DMatrix[5] = DMatrix[6] = DMatrix[7] = 0;
```

```
DMatrix[8] = c*(1 - PoissonRatio)*0.5;

//Print Dmatrix

for (int i = 0; i < 3; i++)
{
for (int j = 0; j < 3; j++)
{
fprintf(out,"%.2f\t",DMatrix[i*3+j]);
}
fprintf(out, "\n");
}



}

tDmat = clock();
ExecTime = tDmat-tcoordinate;
fprintf(t, "D Matrix=%f\n", ExecTime);




//********PART E : Generation of B matrix for all the element**************

float *BMatrix;
BMatrix = (float *)calloc(NumberOfElement * 7 , sizeof(float));

//to store temporary value of nodes and coordinates
```

```
int ii,jj,kk;
float ix, iy, jx, jy, kx, ky;
float b1, b2, b3, c1, c2, c3;
float Delta;

fprintf(out, "\n\n------Calculation of B Matrix----------");

for (int n = 0; n < NumberOfElement; n++)
{
ii = DataNode[n * 3];
jj = DataNode[n * 3 + 1];
kk = DataNode[n * 3 + 2];

ix = DataJoint[(ii - 1) * 2];
iy = DataJoint[(ii - 1) * 2 + 1];

jx = DataJoint[(jj - 1) * 2];
jy = DataJoint[(jj - 1) * 2 + 1];

kx = DataJoint[(kk - 1) * 2];
ky = DataJoint[(kk - 1) * 2 + 1];

b1 = jy - ky;
b2 = ky - iy;
b3 = iy - jy;

c1 = kx - jx;
c2 = ix - kx;
c3 = jx - ix;
```

```
Delta = 0.5*((jx*ky - jy*kx) - (ix*ky-kx*iy) + (ix*jy-iy*jx));

if (Delta<0)
{
Delta = -Delta;
}

BMatrix[7 * n] =  Delta;
BMatrix[7 * n + 1] = b1;
BMatrix[7 * n + 2] = b2;
BMatrix[7 * n + 3] = b3;
BMatrix[7 * n + 4] = c1;
BMatrix[7 * n + 5] = c2;
BMatrix[7 * n + 6] = c3;

//print B matrix on temp file
fprintf(out,"\nB matrix element=%d",n);
fprintf(out,"\n%0.2f\t0.00\t%0.2f\t0.00\t%0.2f\t0.00",BMatrix[7*n+1],BMatrix[7*n
fprintf(out, "\n0.00\t%0.2f\t0.00\t%0.2f\t0.00\t%0.2f", BMatrix[7 * n + 4], BMat
fprintf(out, "\n%0.2f\t%0.2f\t%0.2f\t%0.2f\t%0.2f\t%0.2f", BMatrix[7 * n + 4], B
fprintf(out,"\nCalculated value of Delta=%f\n",Delta);
}

tBmat = clock();
ExecTime = tBmat - tDmat;
fprintf(t, "B Matrix=%f\n", ExecTime);
```

```
//**********PART F: calculation for stifness matrix for each member*********

float *KMatrix;
KMatrix = (float *)calloc(NumberOfElement *36 , sizeof(float));
//to store B matrix for each element
float tempB[18];
//to store resultant matrix for D*B
float tempK[18] = {0};
//to store transpose of B matrix
float tempBT[18] = {0};
float sum = 0;
//to store temp matrix during row col conversation
float tempKmat[36] = { 0 };

fprintf(out, "\n\n------Calculation of K Matrix----------");

for (int i = 0; i < NumberOfElement; i++)
{
tempB[0] = tempB[13] = BMatrix[7 * i + 1] * 0.5 / BMatrix[7 * i];
tempB[2] = tempB[15] = BMatrix[7 * i + 2] * 0.5 / BMatrix[7 * i];
tempB[4] = tempB[17] = BMatrix[7 * i + 3] * 0.5 / BMatrix[7 * i];

tempB[7] = tempB[12] = BMatrix[7 * i + 4] * 0.5 / BMatrix[7 * i];
tempB[9] = tempB[14] = BMatrix[7 * i + 5] * 0.5 / BMatrix[7 * i];
tempB[11] = tempB[16] = BMatrix[7 * i + 6] * 0.5 / BMatrix[7 * i];
```

```
tempB[1] = tempB[3] = tempB[5] = tempB[6] = tempB[8] = tempB[10] = 0;

//part1:Matrix multiplication of D and B Matrix
for (int l = 0; l < 3; l++)
{
for (int k = 0; k < 6; k++)
{
sum = 0;
for (int  j = 0; j < 3; j++)
{
sum = sum + DMatrix[l * 3 +j ] * tempB[j*6+k];
}
tempK[l * 6 + k]=sum;
}

}

//Part2: Matrix Multiplication for obtaining K matrix
//here first we need to obtain transpose of B matrix
for (int j = 0; j < 6; j++)
{
for (int k = 0; k < 3; k++)
{
tempBT[j * 3 + k] = tempB[k * 6 + j];
}
}

fprintf(out, "\n\nFinal K matrix for Element=%d\n", i);
```

```
for (int l = 0; l < 6; l++)
{
for (int k = 0; k < 6; k++)
{
sum = 0;
for (int j = 0; j < 3; j++)
{
sum = sum + tempBT[l*3+j] * tempK[j*6+k];
}
KMatrix[l * 6 + k+i*36]=sum*Thickness*Delta;
fprintf(out, "%0.2f\t", KMatrix[l * 6 + k + i * 36]);
}
fprintf(out,"\n");
}

//arrange K matrix row and column in proper sequence
//this is required for assemble global matrix

if (i%2==0)
{
for (int j = 0; j < 6; j++)
{
for (int k = 0; k < 6; k++)
{
tempKmat[j * 6 + k] = KMatrix[i * 36+j * 6 + k];
}
}
//conversion
//part 1 Upper diagonal blocks
```

```
KMatrix[i * 36 + 2] = tempKmat[4];
KMatrix[i * 36 +3] = tempKmat[5];
KMatrix[i * 36 + 8] = tempKmat[10];
KMatrix[i * 36 + 9] = tempKmat[11];

KMatrix[i * 36 + 4] = tempKmat[2];
KMatrix[i * 36 + 5] = tempKmat[3];
KMatrix[i * 36 + 10] = tempKmat[8];
KMatrix[i * 36 + 11] = tempKmat[9];

KMatrix[i * 36 +14] = tempKmat[28];
KMatrix[i * 36 + 15] = tempKmat[29];
KMatrix[i * 36 + 20] = tempKmat[34];
KMatrix[i * 36 + 21] = tempKmat[35];

KMatrix[i * 36 + 16] = tempKmat[26];
KMatrix[i * 36 + 17] = tempKmat[27];
KMatrix[i * 36 + 22] = tempKmat[32];
KMatrix[i * 36 + 23] = tempKmat[33];

KMatrix[i * 36 + 28] = tempKmat[14];
KMatrix[i * 36 + 29] = tempKmat[15];
KMatrix[i * 36 + 34] = tempKmat[20];
KMatrix[i * 36 + 35] = tempKmat[21];

//lower blocks
KMatrix[i * 36 + 12] = tempKmat[24];
KMatrix[i * 36 + 13] = tempKmat[25];
KMatrix[i * 36 + 18] = tempKmat[30];
```

```
KMatrix[i * 36 + 19] = tempKmat[31];

KMatrix[i * 36 + 24] = tempKmat[12];
KMatrix[i * 36 + 25] = tempKmat[13];
KMatrix[i * 36 + 30] = tempKmat[18];
KMatrix[i * 36 + 31] = tempKmat[19];

KMatrix[i * 36 + 26] = tempKmat[16];
KMatrix[i * 36 + 27] = tempKmat[17];
KMatrix[i * 36 + 32] = tempKmat[22];
KMatrix[i * 36 + 33] = tempKmat[23];



}

fprintf(out, "\n\nFinal K matrix after re arrangement Element=%d\n", i);

for (int j = 0; j < 6; j++)
{
for (int k = 0; k < 6; k++)
{
fprintf(out, "%.2f\t", KMatrix[j * 6 + k + i * 36]);
}
fprintf(out, "\n");
}

}

tKmat = clock();
```

```
ExecTime = tKmat - tBmat;
fprintf(t, "K Matrix=%f\n", ExecTime);




//****************PART G:Assemble stiffness matrix******************

//here logic is update Global stiffness matrix based on it's node value
float *GlobalKMatrix;
//GlobalKMatrix = (float *)malloc(NumberOfNode * 4*(PartX+3) * sizeof(float));
GlobalKMatrix = (float *)calloc(NumberOfNode * 4 * NumberOfNode , sizeof(float))
//fprintf(temp, "\nGlobal Matix initialization to zero\n");

fprintf(out, "\n\n------Assemble of K Matrix----------");

//array to store row and col ID
int RID[6] = {0};

int index = 0;
for (int i = 0; i < NumberOfElement; i++)
{

if (i%2==0)
{
ii = DataNode[i * 3];
jj = DataNode[i * 3 + 2];
kk = DataNode[i * 3 + 1];
}
```

```
else
{
ii = DataNode[i * 3];
jj = DataNode[i * 3 + 1];
kk = DataNode[i * 3 + 2];

}


    RID[0] = ii * 2 - 2;
RID[1] = ii * 2 - 1;


RID[2] = jj * 2 - 2;
RID[3] = jj * 2 - 1;


RID[4] = kk * 2 - 2;
RID[5] = kk * 2 - 1;

for (int j = 0; j < 6; j++)
{
for (int k = j; k < 6; k++)
{
//For Half Band
//index = RID[j] * (PartX + 3) * 2 + RID[k] - RID[j];;
//GlobalKMatrix[index] = GlobalKMatrix[index]+ KMatrix[i * 36 + j * 6 + k];
index = RID[j] * (NumberOfNode)* 2 + RID[k];
GlobalKMatrix[index] = GlobalKMatrix[index] + KMatrix[i * 36 + j * 6 + k];

}
}
```

```
 }



fprintf(temp, "\n\n------Assemble of Global Matrix----------\n");
//initialize Global Matrix to zero
for (int i = 0; i < NumberOfNode * 2; i++)
{
fprintf(temp, "%d\t", i);
for (int j = 0; j < NumberOfNode * 2; j++)
{
fprintf(temp, "%0.2f\t", GlobalKMatrix[i*(NumberOfNode) * 2 + j]);
}
fprintf(temp,"\n");

}

//apply Boundry conditions
//Node for Fixed suport
int *BC;
BC = (int *)malloc((PartY+1) * sizeof(float));
BC[0] = 1;
for (int i = 0; i < PartY; i++)
{
BC[i+1] = BC[i]+PartX+1;

}
//fprintf(temp, "\nBoundry conditions");
for (int i = 0; i < PartY+1; i++)
{
```

```
// fprintf(temp,"\nBC[%d]=%d",i,BC[i]);


}
//Calculate Row ID for Global Matrix
int *RowBC;
RowBC = (int *)malloc((PartY + 1)*2 * sizeof(float));
for (int i = 0; i < PartY + 1; i++)
{
RowBC[2*i] = BC[i] * 2 - 2;
RowBC[2 * i + 1] = BC[i] * 2 - 1;


}
//fprintf(temp, "\nBoundry conditions Row Index");
for (int i = 0; i < PartY + 1; i++)
{
//fprintf(temp, "\nRowBC[%d]=%d\nRowBC[%d]=%d", i * 2, RowBC[i * 2], i * 2+1, Ro


}


//Store global matrix to halfBand
float *HBMatrix;
//GlobalKMatrix = (float *)malloc(NumberOfNode * 4*(PartX+3) * sizeof(float));
int HBsize = (NumberOfNode - (PartY + 1)) * 2+1;

HBMatrix = (float *)calloc(HBsize * (2 * (PartX + 3)+1) , sizeof(float));

//Store HB  Matrix after applying BC
int HBrow = 1;
int HBcol = 1;
int bid = 0;
int cid = 0;
for (int i = 0; i < NumberOfNode * 2; i++)
{
if (i!=RowBC[bid])
{
HBcol = 1;
cid = bid;
for (int j = i; j < NumberOfNode * 2; j++)
{
if (j!=RowBC[cid])
{
HBMatrix[HBrow*(PartX + 3) * 2 + HBcol] = GlobalKMatrix[i * (NumberOfNode)* 2 +
HBcol = HBcol + 1;
}
else
{

cid = cid + 1;
if (cid==(PartY+1)*2)
{
cid = cid - 1;
}
}



}
HBrow = HBrow + 1;
}
else
{
bid = bid + 1;
}




}


fprintf(temp, "\n\n-----------Final Half Band -------------\n");
//initialize Global Matrix to zero
for (int i = 1; i < HBsize; i++)
{
fprintf(temp, "%d\t", i);
for (int j = 1; j <= (PartX + 3) * 2; j++)
{

fprintf(temp, "%0.2f\t", HBMatrix[i*(PartX + 3) * 2 + j]);
}
fprintf(temp,"\n");

}
tHB = clock();
ExecTime = tHB-tKmat ;
fprintf(t, "GLobal=%f\n", ExecTime);
//Calculation of Force Vector
float *ForceVector;
ForceVector = (float *)malloc((HBsize+1)* sizeof(float));
```

```
for (int i = 1; i <= HBsize; i++)
{

ForceVector[i] = 0;



}
ForceVector[HBsize-1] = PointLoad;



fprintf(temp,"\nForce Vector\n");
for (int i = 1; i < HBsize; i++)
{
fprintf(temp,"\n%d\t%f",i,ForceVector[i]);
}

tForce = clock();
ExecTime = tForce-tHB;
fprintf(t, "Force=%f\n", ExecTime);

//calculation to find displacement vector

int BandWidth = (PartX + 3) * 2;
band(HBMatrix, ForceVector, HBsize-1,BandWidth);
fprintf(temp, "\nsolution Vector\n");

for (int i = 1; i <HBsize; i++)
{
fprintf(temp, "\n%d\t%f", i, ForceVector[i]);
```

```
}
fprintf(temp, "\nDisplacement Vector\n");
bid = 0;
for (int i = 0; i <NumberOfNode*2; i++)
{
if (i==RowBC[bid])
{
fprintf(temp, "\n%d\t0", i);
bid = bid + 1;
}
else
{
fprintf(temp, "\n%d\t%f", i, ForceVector[i-bid+1]);

}

}

//Calculation of stress for each Element

float *Stress = (float*)calloc(3*NumberOfElement,sizeof(float));

for (int i = 0; i < NumberOfElement; i++)
{

tempB[0] = tempB[13] = BMatrix[7 * i + 1] * 0.5 / BMatrix[7 * i];
tempB[2] = tempB[15] = BMatrix[7 * i + 2] * 0.5 / BMatrix[7 * i];
tempB[4] = tempB[17] = BMatrix[7 * i + 3] * 0.5 / BMatrix[7 * i];
```

```
tempB[7] = tempB[12] = BMatrix[7 * i + 4] * 0.5 / BMatrix[7 * i];
tempB[9] = tempB[14] = BMatrix[7 * i + 5] * 0.5 / BMatrix[7 * i];
tempB[11] = tempB[16] = BMatrix[7 * i + 6] * 0.5 / BMatrix[7 * i];

tempB[1] = tempB[3] = tempB[5] = tempB[6] = tempB[8] = tempB[10] = 0;

//part1:Matrix multiplication of D and B Matrix
for (int l = 0; l < 3; l++)
{
for (int k = 0; k < 6; k++)
{
sum = 0;
for (int j = 0; j < 3; j++)
{
sum = sum + DMatrix[l * 3 + j] * tempB[j * 6 + k];
}
tempK[l * 6 + k] = sum;
}

}

///Part 2: Matrix multilication of tempk and q


}

tdis = clock();
ExecTime = tdis-tForce;
fprintf(t, "Disp=%f\n", ExecTime);
```

```
ExecTime = tdis - tinput;
fprintf(t, "Total=%f\n", ExecTime);
fclose(in);
fclose(out);
fclose(temp);
return 0;
}

void band(float *HBMatrix, float *ForceVector, int NN, int MM)
{
//triangularise and reduce right hand side
int NL, NM, MR, N, L, K, i, j;
float BN, C;
NL = NN - MM + 1;
NM = NN - 1;
MR = MM;
for (int N = 1; N <= NM; N++)
{
if (HBMatrix[N*(MM)+1] == 0)
{
fprintf(out, "ZERO OR NEGATIVE ELEMENT ON MAIN DIAGONAL OF \n");
fprintf(in, "TRIANGULARIZED MATRIX FOR EQUATION %d ", N);
}
BN = ForceVector[N];
ForceVector[N] = BN / HBMatrix[N*MM + 1];
if (N>NL)
MR = NN - N + 1;
for (int L = 2; L <= MR; L++)
{
```

```
if (HBMatrix[N*MM + L] == 0)
continue;
C = HBMatrix[N*MM + L] / HBMatrix[N*MM + 1];
i = N + L - 1;
j = 0;
for (int K = L; K <= MR; K++)
{
j = j + 1;
HBMatrix[i*MM + j] = HBMatrix[i*MM + j] - C*HBMatrix[N*MM + K];
}
ForceVector[i] = ForceVector[i] - C*BN;
HBMatrix[N*MM + L] = C;
}
}
//Back Substitute
i = NN;
ForceVector[NN] = ForceVector[NN] / HBMatrix[NN*MM + 1];
for (int N = 1; N <= NM; N++)
{
i = i - 1;
if (N<MM)
MR = N + 1;
for (int j = 2; j <= MR; j++)
{
K = i + j - 1;
ForceVector[i] = ForceVector[i] - HBMatrix[i*MM + j] * ForceVector[K];
}
```

```
}
}

/*****Sample Input File********


3000 600 200
10 2

200000 0.24

0 - 100

Details

length depth thickness
parts-along-length  parts-along-depth
Elasticity Poisson's-Ratio

Type-of-Element(0 for plain stress/1 for plain strain)
LoadatEnd-Ydir

*/
```

D.2 Parallel Program

```
//Finite Element Analysis Based on CST Element Parallel Program
#include<cuda.h>
```

```
#include<cuda_runtime.h>
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
#include<math.h>
#include<time.h>

FILE *in, *out, *temp, *t;
char filename[15];
char file[15];
void band(float *HBMatrix, float *ForceVector, int NN, int MM);
//kernel-1:Generation of Node for each element
__global__ void k1(float *DataNode,int X)
{
int tid = blockIdx.x + gridDim.x* blockIdx.y;
int BlockID = 0;
int ColID = 0;
BlockID = (tid / 2) + 1;
ColID = tid / (X * 2);
DataNode[tid * 3] = BlockID + ColID;
//ColID = ColID + 1;

if (tid % 2 == 0)
{
DataNode[tid * 3 + 1] = (X + 1) + BlockID + 1 + ColID;
DataNode[tid * 3 + 2] = DataNode[tid * 3 + 1] - 1;
}
else
{
```

```
DataNode[tid * 3 + 1] = DataNode[tid * 3] + 1;
DataNode[tid * 3 + 2] = (X + 1) + BlockID + 1 + ColID;

}
}


//Kernel-2:Generation of joint coordinate for each node
__global__ void k2(float *DataJoint, int X,float Lx,float Ly)
{
int tid = blockIdx.x + gridDim.x* blockIdx.y;
float CoordinateX = 0;
float CoordinateY = 0;

int Rid = 0;
int Cid=0;
Rid = (tid) / (X + 1);
Cid = (tid) % (X+1);
CoordinateX = Lx * Cid;
CoordinateY = Ly * Rid;


DataJoint[tid * 2] = CoordinateX;
DataJoint[tid * 2 + 1] = CoordinateY;





}
```

```
//Kernel-3:Calculate B matrix for each element
__global__ void k3(float *BMatrix, float *DataNode, float *DataJoint)
{

int tid = blockIdx.x + gridDim.x* blockIdx.y;
//to store temporary value of nodes and coordinates
int ii, jj, kk;

float ix, iy, jx, jy, kx, ky;
float b1, b2, b3, c1, c2, c3;
float Delta;
ii = DataNode[tid * 3];
jj = DataNode[tid * 3 + 1];
kk = DataNode[tid * 3 + 2];

ix = DataJoint[(ii - 1) * 2];
iy = DataJoint[(ii - 1) * 2 + 1];

jx = DataJoint[(jj - 1) * 2];
jy = DataJoint[(jj - 1) * 2 + 1];

kx = DataJoint[(kk - 1) * 2];
ky = DataJoint[(kk - 1) * 2 + 1];

b1 = jy - ky;
b2 = ky - iy;
b3 = iy - jy;

c1 = kx - jx;
```

```
c2 = ix - kx;
c3 = jx - ix;


Delta = 0.5*((jx*ky - jy*kx) - (ix*ky - kx*iy) + (ix*jy - iy*jx));
if (Delta<0)
{
Delta = -Delta;
}

BMatrix[7 * tid] = Delta;
BMatrix[7 * tid + 1] = b1;
BMatrix[7 * tid + 2] = b2;
BMatrix[7 * tid + 3] = b3;
BMatrix[7 * tid + 4] = c1;
BMatrix[7 * tid + 5] = c2;
BMatrix[7 * tid + 6] = c3;



}
//kernel-4:Calculate K matrix for each element
__global__ void k4(float *KMatrix, float *BMatrix, float *DMatrix, float Thickne
{

int tid = blockIdx.x + gridDim.x* blockIdx.y;
//to store B matrix for each element
float tempB[18];
//to store resultant matrix for D*B
float tempK[18] = { 0 };
```

```
//to store transpose of B matrix
float tempBT[18] = { 0 };
//to store temp matrix during row col conversation
float tempKmat[36] = { 0 };
float sum = 0;
tempB[0] = tempB[13] = BMatrix[7 * tid + 1] * 0.5 / BMatrix[7 * tid];
tempB[2] = tempB[15] = BMatrix[7 * tid + 2] * 0.5 / BMatrix[7 * tid];
tempB[4] = tempB[17] = BMatrix[7 * tid + 3] * 0.5 / BMatrix[7 * tid];

tempB[7] = tempB[12] = BMatrix[7 * tid + 4] * 0.5 / BMatrix[7 * tid];
tempB[9] = tempB[14] = BMatrix[7 * tid + 5] * 0.5 / BMatrix[7 * tid];
tempB[11] = tempB[16] = BMatrix[7 * tid + 6] * 0.5 / BMatrix[7 * tid];

tempB[1] = tempB[3] = tempB[5] = tempB[6] = tempB[8] = tempB[10] = 0;

//part1:Matrix multiplication of D and B Matrix
for (int l = 0; l < 3; l++)
{
for (int k = 0; k < 6; k++)
{
sum = 0;
for (int j = 0; j < 3; j++)
{
sum = sum + DMatrix[l * 3 + j] * tempB[j * 6 + k];
}
tempK[l * 6 + k] = sum;
}

}
```

```
//Part2: Matrix Multiplication for obtaining K matrix
//here first we need to obtain transpose of B matrix
for (int j = 0; j < 6; j++)
{
for (int k = 0; k < 3; k++)
{
tempBT[j * 3 + k] = tempB[k * 6 + j];
}
}
//fprintf(temp, "\n\nFinal K matrix for Element=%d\n", i);
for (int l = 0; l < 6; l++)
{
for (int k = 0; k < 6; k++)
{
sum = 0;
for (int j = 0; j < 3; j++)
{
sum = sum + tempBT[l * 3 + j] * tempK[j * 6 + k];
}
KMatrix[l * 6 + k + tid * 36] = sum*Thickness*BMatrix[7 * tid];
//fprintf(temp, "%.2f\t", KMatrix[l * 6 + k + i * 36]);
}
//fprintf(temp, "\n");
}

//arrange K matrix row and column in proper sequence
//this is required for assemble global matrix
if (tid % 2 == 0)
```

```
{
for (int j = 0; j < 6; j++)
{
for (int k = 0; k < 6; k++)
{
tempKmat[j * 6 + k] = KMatrix[tid * 36 + j * 6 + k];
}
}
//conversion
//part 1 Upper diagonal blocks
KMatrix[tid * 36 + 2] = tempKmat[4];
KMatrix[tid * 36 + 3] = tempKmat[5];
KMatrix[tid * 36 + 8] = tempKmat[10];
KMatrix[tid * 36 + 9] = tempKmat[11];

KMatrix[tid * 36 + 4] = tempKmat[2];
KMatrix[tid * 36 + 5] = tempKmat[3];
KMatrix[tid * 36 + 10] = tempKmat[8];
KMatrix[tid * 36 + 11] = tempKmat[9];

KMatrix[tid * 36 + 14] = tempKmat[28];
KMatrix[tid * 36 + 15] = tempKmat[29];
KMatrix[tid * 36 + 20] = tempKmat[34];
KMatrix[tid * 36 + 21] = tempKmat[35];

KMatrix[tid * 36 + 16] = tempKmat[26];
KMatrix[tid * 36 + 17] = tempKmat[27];
KMatrix[tid * 36 + 22] = tempKmat[32];
KMatrix[tid * 36 + 23] = tempKmat[33];
KMatrix[tid * 36 + 28] = tempKmat[14];
KMatrix[tid * 36 + 29] = tempKmat[15];
KMatrix[tid * 36 + 34] = tempKmat[20];
KMatrix[tid * 36 + 35] = tempKmat[21];

//lower blocks
KMatrix[tid * 36 + 12] = tempKmat[24];
KMatrix[tid * 36 + 13] = tempKmat[25];
KMatrix[tid * 36 + 18] = tempKmat[30];
KMatrix[tid * 36 + 19] = tempKmat[31];

KMatrix[tid * 36 + 24] = tempKmat[12];
KMatrix[tid * 36 + 25] = tempKmat[13];
KMatrix[tid * 36 + 30] = tempKmat[18];
KMatrix[tid * 36 + 31] = tempKmat[19];

KMatrix[tid * 36 + 26] = tempKmat[16];
KMatrix[tid * 36 + 27] = tempKmat[17];
KMatrix[tid * 36 + 32] = tempKmat[22];
KMatrix[tid * 36 + 33] = tempKmat[23];


}


}
//Kernel-5:Calculate F vector for each node
__global__ void k5(float *ForceVector, int NumberOfNode, int PointLoad)
```

```
{
int tid = blockIdx.x + gridDim.x* blockIdx.y;
ForceVector[tid] = 0;

}
//kernel-5:Calculate stress

int main()
{
//float A[N*(N + 1)];
float Length, Depth, Thickness, ModulasOfElasticity, PoissonRatio, PointLoad;
int PartX, PartY, NumberOfElement, NumberOfNode, TypeofElement;
clock_t tstart, tinput, tnode, tcoordinate, tDmat, tBmat, tKmat, tHB,tForce,tdis
double ExecTime;
tstart = clock();
in = fopen("input.dat", "r");
fscanf(in, "%f %f %f", &Length, &Depth, &Thickness);
fscanf(in, "%d %d", &PartX, &PartY);
fscanf(in, "%f %f %d %f", &ModulasOfElasticity, &PoissonRatio, &TypeofElement, &

NumberOfElement = PartX*PartY * 2;
NumberOfNode = (PartX + 1)*(PartY + 1);
sprintf(filename, "out_%d.txt", NumberOfElement);
out = fopen(filename, "w");
temp = fopen("temp.txt", "w");
sprintf(file, "time_%d_%d.txt", PartX, PartY);
t = fopen(file, "w");
//Printing input data
fprintf(out, "*****input data*******\n");
fprintf(out, "Length=%.2f\nDepth=%.2f\nThickness=%.2f\n", Length, Depth, Thickne
fprintf(out, "Elements along Length=%d\nElement along Depth=%d\n", PartX, PartY)
fprintf(out, "Modulas of Elasticity=%.2f\npoisson's Ratio=%.2f\nPoint Load=%.2f\
if (TypeofElement == 0)
{
fprintf(out, "Type of Element = Plain stress Element");
}
if (TypeofElement == 1)
{
fprintf(out, "Type of Element = Plain strain Element");
}
tinput = clock();
ExecTime = tstart - tinput;
fprintf(t, "Input data=%f\n", ExecTime);
//Generation of Nodal data for each element
float *DataNode;
DataNode = (float *)malloc(NumberOfElement * 3 * sizeof(float));
int BlockID = 0;
int ColID = 0;

//Allocation of Memory on GPU
int size = NumberOfElement*3*sizeof(float);
float *DataNodeGPU;
cudaMalloc(&DataNodeGPU, size);

//Specify Size of Block and Thread
dim3 grid(NumberOfElement, 1);
dim3 threadBlock(1, 1);
```

```
k1 << <NumberOfElement,1  >> >(DataNodeGPU,PartX);

cudaMemcpy(DataNode, DataNodeGPU, size, cudaMemcpyDeviceToHost);
for (int i = 0; i < NumberOfElement; i++)
{
//fprintf(temp, "No of Element =%d\tnode:%.2f\t%.2f\t%.2f\n",i, DataNode[i * 3],
}

tnode = clock();
ExecTime = tnode - tinput;
fprintf(t, "Node=%f\n", ExecTime);

//Generation of joint coordinates for each node
float *DataJoint;
DataJoint = (float *)malloc(NumberOfNode * 2 * sizeof(float));



//Allocation of Memory on GPU
size = NumberOfNode * 2 * sizeof(float);
float *DataJointGPU;
cudaMalloc(&DataJointGPU, size);
float lengthX = 0;
float lengthY = 0;
lengthX = Length / PartX;
lengthY = Depth / PartY;

//Specify Size of Block and Thread
//dim3 grid(NumberOfNode, 1);
```

```
//dim3 threadBlock(1, 1);

k2 << <NumberOfNode, 1 >> >(DataJointGPU,PartX,lengthX,lengthY);

cudaMemcpy(DataJoint, DataJointGPU, size, cudaMemcpyDeviceToHost);

for (int i = 0; i < NumberOfNode; i++)
{
// fprintf(temp, "\n%d\t%f\t%f", i + 1, DataJoint[i * 2], DataJoint[i * 2 + 1]);
}
tcoordinate = clock();
ExecTime = tcoordinate - tnode;
fprintf(t, "Coordinates=%f\n", ExecTime);

//Generation of [D] matrix for all elements

float DMatrix[9];
float c;
//Allocation of Memory on GPU
size = 9 * sizeof(float);
float *DMatrixGPU;
cudaMalloc(&DMatrixGPU, size);
if (TypeofElement == 0)
{
c = ModulasOfElasticity / (1 - PoissonRatio*PoissonRatio);
DMatrix[0] = DMatrix[4] = c * 1;
DMatrix[1] = DMatrix[3] = c*PoissonRatio;
DMatrix[2] = DMatrix[5] = DMatrix[6] = DMatrix[7] = 0;
DMatrix[8] = c*(1 - PoissonRatio)*0.5;
//Print Dmatrix
//fprintf(temp, "D matrix\n");
for (int i = 0; i < 3; i++)
{
for (int j = 0; j < 3; j++)
{
//fprintf(temp, "%.2f\t", DMatrix[i * 3 + j]);
}
//fprintf(temp, "\n");
}


}

cudaMemcpy(DMatrixGPU, DMatrix, size, cudaMemcpyHostToDevice);

tDmat = clock();
ExecTime = tDmat - tcoordinate;
fprintf(t, "D Matrix=%f\n", ExecTime);

//Generation of B matrix for all the element
//no need to store whole matrix only three b and three c members+ value of delta
//total 7 value of each member
float *BMatrix;
BMatrix = (float *)malloc(NumberOfElement * 7 * sizeof(float));
//Allocation of Memory on GPU
size = NumberOfElement*7 * sizeof(float);
float *BMatrixGPU;
cudaMalloc(&BMatrixGPU, size);
```

```
k3 << <NumberOfElement, 1 >> >(BMatrixGPU, DataNodeGPU,DataJointGPU);
cudaMemcpy(BMatrix, BMatrixGPU, size, cudaMemcpyDeviceToHost);
for (int n = 0; n < NumberOfElement; n++)
{

//print B matrix on temp file
//fprintf(temp, "\nB matrix element=%d", n);
//fprintf(temp, "\n%0.2f\t0.00\t%0.2f\t0.00\t%0.2f\t0.00", BMatrix[7 * n + 1], B
//fprintf(temp, "\n0.00\t%0.2f\t0.00\t%0.2f\t0.00\t%0.2f", BMatrix[7 * n + 4], B
//fprintf(temp, "\n%0.2f\t%0.2f\t%0.2f\t%0.2f\t%0.2f\t%0.2f", BMatrix[7 * n + 4]
//fprintf(temp, "\nCalculated value of Delta=%f\n", BMatrix[7 * n ]);
}
tBmat = clock();
ExecTime = tBmat - tDmat;
fprintf(t, "B Matrix=%f\n", ExecTime);

// calculation for stifness matrix for each member
float *KMatrix;
KMatrix = (float *)malloc(NumberOfElement * 36 * sizeof(float));
//Allocation of Memory on GPU
size = NumberOfElement * 36 * sizeof(float);
float *KMatrixGPU;
cudaMalloc(&KMatrixGPU, size);

k4 << <NumberOfElement, 1 >> >(KMatrixGPU, BMatrixGPU, DMatrixGPU,Thickness);
cudaMemcpy(KMatrix, KMatrixGPU, size, cudaMemcpyDeviceToHost);
for (int i = 0; i < NumberOfElement; i++)
{
//fprintf(temp, "\n\nFinal K matrix for Element=%d\n", i);
```

```
for (int l = 0; l < 6; l++)
{
for (int k = 0; k < 6; k++)
{
//fprintf(temp, "%.2f\t", KMatrix[l * 6 + k + i * 36]);
}
//fprintf(temp, "\n");
}
}

tKmat = clock();
ExecTime = tKmat - tBmat;
fprintf(t, "K Matrix=%f\n", ExecTime);


// Assemble stiffness matrix
//here logic is update Global stiffness matrix based on it's node value
float *GlobalKMatrix;
//GlobalKMatrix = (float *)malloc(NumberOfNode * 4*(PartX+3) * sizeof(float));
GlobalKMatrix = (float *)malloc(NumberOfNode * 4 * NumberOfNode * sizeof(float))
//fprintf(temp, "\nGlobal Matix initialization to zero\n");
fprintf(out, "\n\n------Assemble of K Matrix----------");
//initialize Global Matrix to zero
for (int i = 0; i < NumberOfNode * 2; i++)
{
//fprintf(temp, "%d\t", i);
for (int j = 0; j < NumberOfNode * 2; j++)
{
GlobalKMatrix[i*(NumberOfNode * 2) + j] = 0;
//fprintf(temp, "%0.2f\t", GlobalKMatrix[i*(PartX + 3) * 2 + j]);
}
//fprintf(temp,"\n");

}

//array to store row and col ID
int RID[6] = { 0 };
int ii, jj, kk;
int index = 0;
for (int i = 0; i < NumberOfElement; i++)
{

if (i % 2 == 0)
{
ii = DataNode[i * 3];
jj = DataNode[i * 3 + 2];
kk = DataNode[i * 3 + 1];
}
else
{
ii = DataNode[i * 3];
jj = DataNode[i * 3 + 1];
kk = DataNode[i * 3 + 2];

}



RID[0] = ii * 2 - 2;
```

```
RID[1] = ii * 2 - 1;


RID[2] = jj * 2 - 2;
RID[3] = jj * 2 - 1;


RID[4] = kk * 2 - 2;
RID[5] = kk * 2 - 1;


for (int j = 0; j < 6; j++)
{
for (int k = j; k < 6; k++)
{
//For Half Band
//index = RID[j] * (PartX + 3) * 2 + RID[k] - RID[j];;
//GlobalKMatrix[index] = GlobalKMatrix[index]+ KMatrix[i * 36 + j * 6 + k];
index = RID[j] * (NumberOfNode)* 2 + RID[k];
GlobalKMatrix[index] = GlobalKMatrix[index] + KMatrix[i * 36 + j * 6 + k];


}
}
}




fprintf(temp, "\n\n------Assemble of Global Matrix----------\n");
//initialize Global Matrix to zero
for (int i = 0; i < NumberOfNode * 2; i++)
{
fprintf(temp, "%d\t", i);
for (int j = 0; j < NumberOfNode * 2; j++)
```

```
{
fprintf(temp, "%0.2f\t", GlobalKMatrix[i*(NumberOfNode)* 2 + j]);
}
fprintf(temp, "\n");

}

//apply Boundry conditions
//Node for Fixed suport
int *BC;
BC = (int *)malloc((PartY + 1) * sizeof(float));
BC[0] = 1;
for (int i = 0; i < PartY; i++)
{
BC[i + 1] = BC[i] + PartX + 1;

}
//fprintf(temp, "\nBoundry conditions");
for (int i = 0; i < PartY + 1; i++)
{
// fprintf(temp,"\nBC[%d]=%d",i,BC[i]);

}
//Calculate Row ID for Global Matrix
int *RowBC;
RowBC = (int *)malloc((PartY + 1) * 2 * sizeof(float));
for (int i = 0; i < PartY + 1; i++)
{
RowBC[2 * i] = BC[i] * 2 - 2;
```

```
RowBC[2 * i + 1] = BC[i] * 2 - 1;

}
//fprintf(temp, "\nBoundry conditions Row Index");
for (int i = 0; i < PartY + 1; i++)
{
//fprintf(temp, "\nRowBC[%d]=%d\nRowBC[%d]=%d", i * 2, RowBC[i * 2], i * 2+1, Ro

}

//Store global matrix to halfBand
float *HBMatrix;
//GlobalKMatrix = (float *)malloc(NumberOfNode * 4*(PartX+3) * sizeof(float));
int HBsize = (NumberOfNode - (PartY + 1)) * 2 + 1;

HBMatrix = (float *)malloc(HBsize * (2 * (PartX + 3) + 1) * sizeof(float));
fprintf(temp, "\n\n-----------Half Band Initialization-------------\n");
//initialize Global Matrix to zero
for (int i = 1; i < HBsize; i++)
{
fprintf(temp, "%d\t", i);
for (int j = 1; j <= (PartX + 3) * 2; j++)
{
HBMatrix[i*(PartX + 3) * 2 + j] = 0;
fprintf(temp, "%0.2f\t", HBMatrix[i*(PartX + 3) * 2 + j]);
}
fprintf(temp, "\n");

}
```

```
//Store HB  Matrix after applying BC
int HBrow = 1;
int HBcol = 1;
int bid = 0;
int cid = 0;
for (int i = 0; i < NumberOfNode * 2; i++)
{
if (i != RowBC[bid])
{
HBcol = 1;
cid = bid;
for (int j = i; j < NumberOfNode * 2; j++)
{
if (j != RowBC[cid])
{
HBMatrix[HBrow*(PartX + 3) * 2 + HBcol] = GlobalKMatrix[i * (NumberOfNode)* 2 +
HBcol = HBcol + 1;
}
else
{

cid = cid + 1;
if (cid == (PartY + 1) * 2)
{
cid = cid - 1;
}
}



}
HBrow = HBrow + 1;

}
else
{
bid = bid + 1;
}




}

fprintf(temp, "\n\n-----------Final Half Band -------------\n");
//initialize Global Matrix to zero
for (int i = 1; i < HBsize; i++)
{
fprintf(temp, "%d\t", i);
for (int j = 1; j <= (PartX + 3) * 2; j++)
{

fprintf(temp, "%0.2f\t", HBMatrix[i*(PartX + 3) * 2 + j]);
}
fprintf(temp, "\n");

}
```

```
tHB = clock();
ExecTime = tHB - tKmat;
fprintf(t, "GLobal=%f\n", ExecTime);

//Calculation of Force Vector
float *ForceVector;
ForceVector = (float *)malloc((HBsize + 1)* sizeof(float));
//Allocation of Memory on GPU
size = (HBsize + 1 )* sizeof(float);
float *ForceVectorGPU;
cudaMalloc(&ForceVectorGPU, size);

k5 << <HBsize+1, 1 >> >(ForceVectorGPU, NumberOfNode,PointLoad);
cudaMemcpy(ForceVector, ForceVectorGPU, size, cudaMemcpyDeviceToHost);


ForceVector[HBsize - 1] = PointLoad;
fprintf(temp, "\nForce Vector\n");
for (int i = 1; i < HBsize; i++)
{
// fprintf(temp, "\n%d\t%f\t%f", i + 1, ForceVector[i * 2], ForceVector[i * 2 +
}
ForceVector[HBsize - 1] = PointLoad;
//calculation to find displacement vector


int BandWidth = (PartX + 3) * 2;
band(HBMatrix, ForceVector, HBsize - 1, BandWidth);
```

```
fprintf(temp, "\nsolution Vector\n");

for (int i = 1; i <HBsize; i++)
{
fprintf(temp, "\n%d\t%f", i, ForceVector[i]);
}
fprintf(temp, "\nDisplacement Vector\n");
bid = 0;
for (int i = 0; i <NumberOfNode * 2; i++)
{
if (i == RowBC[bid])
{
fprintf(temp, "\n%d\t0", i);
bid = bid + 1;
}
else
{
fprintf(temp, "\n%d\t%f", i, ForceVector[i - bid + 1]);

}


}
tdis = clock();
//ExecTime = tdis - tForce;
//fprintf(t, "Disp=%f\n", ExecTime);
ExecTime = tdis - tinput;
fprintf(t, "Total=%f\n", ExecTime);
fclose(in);
fclose(out);
```

```
return 0;
}

void band(float *HBMatrix, float *ForceVector, int NN, int MM)
{
//triangularise and reduce right hand side
int NL, NM, MR, N, L, K, i, j;
float BN, C;
NL = NN - MM + 1;
NM = NN - 1;
MR = MM;
for (int N = 1; N <= NM; N++)
{
if (HBMatrix[N*(MM)+1] == 0)
{
fprintf(out, "ZERO OR NEGATIVE ELEMENT ON MAIN DIAGONAL OF \n");
fprintf(in, "TRIANGULARIZED MATRIX FOR EQUATION %d ", N);
}
BN = ForceVector[N];
ForceVector[N] = BN / HBMatrix[N*MM + 1];
if (N>NL)
MR = NN - N + 1;
for (int L = 2; L <= MR; L++)
{
if (HBMatrix[N*MM + L] == 0)
continue;
C = HBMatrix[N*MM + L] / HBMatrix[N*MM + 1];
i = N + L - 1;
j = 0;
for (int K = L; K <= MR; K++)
{
j = j + 1;
HBMatrix[i*MM + j] = HBMatrix[i*MM + j] - C*HBMatrix[N*MM + K];
}
ForceVector[i] = ForceVector[i] - C*BN;
HBMatrix[N*MM + L] = C;
}
}
//Back Substitute
i = NN;
ForceVector[NN] = ForceVector[NN] / HBMatrix[NN*MM + 1];
for (int N = 1; N <= NM; N++)
{
i = i - 1;
if (N<MM)
MR = N + 1;
for (int j = 2; j <= MR; j++)
{
K = i + j - 1;
ForceVector[i] = ForceVector[i] - HBMatrix[i*MM + j] * ForceVector[K];
}




}
}
```

Appendix E

List of Paper Published/Communicated

a. Application of Graphics processing unit for parallel processing in structural engineering, 7th National civil engineering student symposium(Aakaar 2015),Indian Institute of Technology Bombay,Mumbai.

References

[1] Amdahls law: Wikipedia -http://en.wikipedia.org/wiki/amdahlslaw.

[2] Classes of parallel computer wikipedia -http://en.wikipedia.org/wiki/classesparallelcomputer

[3] Parallel computing :wikipedia -http://en.wikipedia.org/wiki/parallelcomputing.

[4] Top 500 systems - top 500-http://www.top500.org/featured/top-systems.

[5] Types of parallelization:wikipedia -http://en.wikipedia.org/wiki/parallelization.

[6] A.Lamecki A.Dziekonski, P.Sypek and M.Mrozowski. Finite element matrix generation on gpu. *Progress In Electromagnetics Research,*, 128:249–265, May 2012.

[7] T. Bahcecioclu and O. Kurc. Nonlinear dynamic finite element analysis withgpu. In *14th International Conference on Computing in Civil and Building Engineering*. International Society for Computing in Civil and Building Engineering, June 2012.

[8] JunKu Lee Getao Liang David D. Jenkins Gregory D. Peterson Depeng Yang, Junqing Sun and Husheng Li. Performance comparison of cholesky decomposition on gpus and fpgas. Department of Electrical Engineering and Computer Science ,University of Tennessee.

[9] Chaojiang Fu. Parallel computing for finite element structural analysis on workstation cluster. *IEEE*, 2008.

[10] Baidurya Bhattacharya Girish Sharma, Abhishek Agrawala. A fast parallel gauss jordan algorithm for matrix inversion using cuda. *Computers and Structures*, 128:31–37, September 2013. Elsevier.

[11] Jerome F. HAJJARl and John F. ABEL. Parallel processing of nonlinear dynamic analysis of steel frame structures using domain decomposition. In Tokyo-Kyoto, editor, *Proceedings of Ninth World Conference on Earth.quake Engineering*, volume V. JAPAN, August 1988.

[12] Rajat Mittal Hang Liu, Jung-Hee Seo and H. Howie Huang. Gpu-accelerated scalable solver for banded linear systems. *IEEE*, 2013.

[13] Neil G. Dickson Kamran Karimi and Firas Hamze. A performance comparison of cuda and opencl. Master's thesis, ARXIV Cornell University, 2008.

[14] Vishnukanthan Kandasamy. Parallel fem simulation using gpus. Institute for Computing in Engineering,Ruhr University, 2011.

[15] Filip Kruzel and Krzysztof Banas. Vectorized opencl implementation of numerical integration for higher order finite elements. *Elesevier*, August 2013.

[16] Guoxin Zhang Zhaosong Ma Lixiang Wang, Shihai Li and Lei Zhang. A gpu based parallel procedure for non-linear analysis of complex structure using coupled fem/dem approach. *Mathematical Problems in Engineering*, page 15, September 2013. Hindawi Publishing Corporation.

[17] Konark Patel. Application of parallel processing in structural engineering. Master's thesis, Nirma University, May 2013.

[18] Alexey Lastovetsky Ravi Reddy and Pedro Alonso. Parallel solvers for dense linear systems for heterogeneous computational clusters. *IEEE*, 2009.

[19] Jason Sanders. *Cuda By Example, An Introduction to General-Purpose GPU Programming*. Addison Wesley, 1 edition, july 2010.

[20] S.F.McGinn and R.E.Shaw. Parallel gaussian elimination using openmp and mpi. *IEEE*, 2002.

[21] Yuan Sen Yang Shang Hsien Hsieh and Po Yao Hsu. Integration of general sparse matrix and parallel computing technologies for large-scale structural analysis. *Computer-Aided Civil and Infrastructure Engineering*, pages 423–438, 2002. BlackwellPublishing.

[22] E D Sotelino. Parallel processing techniques in structural engineering application. *Journal of Structural Engineering*, 129(12):1698–1706, December 2003. ASCE.

[23] N. Chavannes T.P.Stefanski, S. Benkler and N. Kuster. Parallel implementation of the finite-difference timedomain method in open computing language. *IEEE*, 2010.

[24] N. Balakrishnan V. Mani, B. Dattaguru and T.S. Ramamurthy. Parallel gassian elimnation for banded matrix - a computational model. *IEEE*, 1990.

[25] Jian-She Wang and Nathan Ida. Parallel algorithms for direct solution of large systems of equations. *IEEE*, 1988.

[26] Wang Chengguo Xiao Qian and GuoGe. The research of parallel computing for large-scale finite element model of wheeilrail rolling contact. *IEEE*, 2010.

[27] Pu Chen Zuogui Ning Xuanhua Fan, Rui-an Wu and Jian Li. Parallel computing of large eigenvalue problems for engineering structures. *IEEE*, 2011.

[28] Ibrahim Guven Erdogan Madenci Yoon Kah Leow, Ali Akoglu. High performance linear equation solver using nvidia gpus. University of Arizona.

[29] Qinghai Miao Zhihui Zhang and Ying Wang. Cuda-based jacobis iterative method. *IEEE*, 2009.