APPLICATION OF PARALLEL PROCESSING IN STRUCTURAL ENGINEERING

By

Konark P. Patel

12MCLC21

DEPARTMENT OF CIVIL ENGINEERING

INSTITUTE OF TECHNOLOGY

NIRMA UNIVERSITY

AHMEDABAD-382481

May-2014

APPLICATION OF PARALLEL PROCESSING IN STRUCTURAL ENGINEERING

Major Project

Submitted in Partial Fulfillment of the Requirements for Degree of

MASTER OF TECHNOLOGY

IN

CIVIL ENGINEERING

(Computer Aided Structural Analysis And Design)

By

Konark P. Patel

12MCLC21

DEPARTMENT OF CIVIL ENGINEERING

INSTITUTE OF TECHNOLOGY

NIRMA UNIVERSITY

AHMEDABAD-382481

May-2014

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.

Konark P. Patel

Certificate

This is to certify that the Major Project Report entitled **"APPLICATION OF PARALLEL PROCESSING IN STRUCTURAL ENGINEERING"** submitted by **Mr. Konark P. Patel (Roll No: 12MCLC21)** 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 has 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,
Professor and Head,
Department of Civil engineering,
Institute of Technology,
Nirma University,
Ahmedabad.

Dr. K. Kotecha
Director,
Institute of Technology,
Nirma University,
Ahmedabad.

_____
Examiner

_____
Date of Examination

Abstract

Today's computing environments are becoming more multifaceted, exploiting the capabilities of a range of multi-core microprocessors, Central Processing Units (CPUs), digital signal processors, and graphic processing units (GPUs). Due to heterogeneity in hardware, the process of developing efficient software for such a wide array of architectures poses a number of challenges to the programming community.

Solution of linear equations is a major mathematical process to solve many problems of solid mechanics, fluid dynamics, structural engineering and so on. Since the size of problems increases to achieve accuracy, number of linear equations to be solved also increases and so is the time, to solve equations increases. Advancement of new parallel computation technology using inexpensive graphic card processors (multi-core GPUs) and multi-core CPUs speed up the solution of various problems of structural engineering.

In the present study, computationally intensive problems of structural engineering are implemented on High Performance Computing Platforms like multi-core processors and graphic processing units (GPUs). Direct methods like, Gaussian Elimination and Modified Cholesky solver, for solving linear equations in form of [A]{x}={B} are used. GPUs and CPUs are used for parallel computations with help of OpenCL programming language. It is a step in the direction of heterogeneous computing for smarter, faster and better analysis of problem. The main purpose of using this parallel computation is to minimize the time of structural analysis of problem that involves large number of linear equations.

For parallel implementation of Gaussian Elimination solver, linear equations system representing equilibrium equations of finite element problem is used. Equations in form of [A]{x}={B} are generated from finite element analysis of axial bar

using 3-node bar element where A=Square Stiffness Matrix, B=Load Vector and x=Displacement vector. For solution of equations Matrix-[A] is inverted using sequential and parallel implementation of Gaussian Elimination. Sequential program is developed using C++ and parallel program is developed using OpenCL language. For comparing computational efficiency of parallel code, speedup factor which is ratio of sequential execution time to parallel execution time is calculated for different number of linear equations ranging 101 to 10001. Parallel execution time includes processing time and communication time. As data is transferred between various memories, communication time increases total computational time. Code is executed on different CPUs and GPUs for parametric study.

For parallel implementation of Half-Band solver, which is based on modified cholesky method, Direct Stiffness Method program of Plane Frame and Space Frame are used for generating set of linear equation system. Here stiffness matrix is stored in banded form to reduce memory requirements. Programs for sequential and parallel solution of banded equations are developed using C++ and OpenCL languages. Problems of varying size from 7650 Degrees of Freedom to 1,88,250 Degrees of Freedom are solved using sequential and parallel Half-Band solver. The computational efficiency of parallel code is studied based on speedup factor. Further to understand the efficiency of program on different hardware platform, the parallel code is executed on multi-core CPUs like Intel® Core™i3, i5, i7 processors with different specifications and NVIDIA GPU. Major factors affecting computational efficiency of parallel program are hardware specifications, algorithms used, size of problem, communication time. When parallel code is implemented on multi-core CPUs, communication time is less compared to implementation on GPU. In case of GPU, computational time is reduced because of parallel operations on large number of cores.

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.

Patel Konark P.

12MCLC21

Abbreviations

GPU	Graphics Processing Unit
HPC	High Performance Computing
$GPGPU$	General-purpose computing on graphics processing units
TPP	Theoretical Peak Performance
$Flops$	Floating-point Operations Per Second
$PFlops$	Peta Floating-Point Operations Per Second
API	Application Programming Interface
AMD	Advanced Micro Device
IBM	International Business Machine
$OpenCL$	Open Computing Language
$CUDA$	Compute Unified Device Architecture
$FPGA$	Field-programmable gate array
$LAPACK$	Linear Algebra Package
$MAGMA$	Matrix Algebra on GPU and Multicore Architectures
$GFlop/s$	Giga Floating-point Operations Per Second
FE	Finite Element
FEM	Finite Element Method
DEM	Discrete Element Method

Contents

List of Tables

List of Figures

Chapter 1

Introduction to High Performance Computing

1.1 General

HPC(High Performance Computing) requires substantially more computational resources than are available on current workstations, and typically require concurrent (parallel) computation.
Alternatively HPC is any computational technique that solves a large problem faster than possible using single, commodity systems. HPC can be achieved through

- Custom-designed, high-performance processors (e.g. Cray, NEC)
- Parallel computing
- Distributed computing
- Grid computing

First HPC systems were vector-based systems (e.g. Cray) named 'supercomputers' because they were an order of magnitude more powerful than commercial systems. Now, 'supercomputer' has little meaning "large systems are now just scaled up ver-

sions of smaller systems". However, 'high performance computing' has many meanings like

- can mean high floating-point Operations Per Second (flops) count
  - per processor
  - totaled over many processors working on the same problem
  - totaled over many processors working on related problems
- can mean faster turnaround time
  - more powerful system
  - scheduled to first available system(s)
  - using multiple systems simultaneously

HPC has had tremendous impact on all areas of computational science and engineering in academia, government, and industry. Many problems have been solved with HPC techniques that were impossible to solve with individual workstations or personal computers.

1.2 Advances in Hardware and Platforms for HPC

Approaches to HPC have taken dramatic turns since the earliest systems were introduced in the 1960s. Early HPC architectures pioneered by Seymour Cray relied on compact innovative designs and local parallelism to achieve superior computational peak performance. However, in time the demand for increased computational power ushered in the age of massively parallel systems.

While the HPC computers of the 1970s used only a few processors, in the 1990s, machines with thousands of processors began to appear and by the end of the 20th century, massively parallel supercomputers with tens of thousands of "off-the-shelf"

processors were the convention. Some of the computers of the 21st century can use over 100,000 processors (some being graphic units) connected by fast connections.

Throughout the decades, the management of heat density has remained a key issue for most centralized supercomputers. The large amount of heat generated by a system may also have other effects, e.g. reducing the lifetime of other system components. There have been diverse approaches to heat management, from pumping Fluorinert through the system, to a hybrid liquid-air cooling system or air cooling with normal air conditioning temperatures.

Systems with a massive number of processors generally take one of two paths: in one approach, known as grid computing, the processing power of a large number of computers in distributed, diverse administrative domains, is opportunistically used whenever a computer is available. In another approach, a large number of processors are used in close proximity to each other, e.g. in a computer cluster. In such a centralized massively parallel system the speed and flexibility of the interconnect becomes very important. The use of multi-core processors combined with centralization is an emerging direction, e.g. as in the Cyclops64 system formerly known as Blue Gene which is a cellular architecture in development by IBM. The Cyclops64 project aims to create the first "supercomputer on a chip".[32]

As the price/performance of general purpose graphic processors (GPGPUs) has improved, a number of petaflop supercomputers such as Tianhe-I and Nebulae have started to rely on them. However, other systems such as the K computer continue to use conventional processors such as SPARC-based designs. The overall applicability of GPGPUs in general purpose high performance computing applications has been the subject of debate since past. A GPGPU may be tuned to score well on specific benchmarks its overall applicability to everyday algorithms may be limited unless significant effort is spent to tune the application towards it. However, GPUs are gaining ground and in 2012 the Jaguar supercomputer was transformed into Titan by replacing CPUs with GPUs.

1.3 Evolution of HPC

The history of evolution of high performance computing is discussed in this section.[33] Some of the benchmarks in computing are as follows:

Experimental/Special Purpose Computing

- 1939-Atanasoff and Berry build prototype electronic/digital computer at Iowa State University
- 1941-Conrad Zuse completed Z3, first functional programmable electromechanical digital computer
- 1943-Bletchley Park operated Colossus, computer based on vacuum tubes, by Turing, Flowers, and Newman
- 1946-ENIAC developed by Eckert and Mauchly, at the University of Pennsylvania
- 1951-UNIVAC I (also designed by Eckert and Mauchly), produced by Remington Rand, delivered to US Census Bureau
- 1952-ILLIAC I (based on Eckert, Mauchly, and von Neumann design), first electronic computer built and housed at a University

The Cray Years[33]

- 1962-Control Data Corp. introduced the first commercially successful supercomputer, the CDC 6600, designed by Seymour Cray. Theoretical Peak Performance(TPP) of 9 MFlop/s
- 1967-Texas Instruments Advanced Scientific Computer, similar to CDC 6600, included vector processing instructions
- 1968-CDC 6800, Crays redesign of 6600, remained fastest supercomputer until mid 1970s. TPP of about 40MFlop/s.

- 1976-Cray Research Incs Cray-I started vector revolution. TPP of about 250MFlop/s.
- 1982-Cray X-MP, Crays first multiprocessor computer, original 2 processor design had a TPP of 400MFlop/s, included shared memory access between processors

Clusters Take Over[33]

- 1993-Cray introduced the T3D - an MPP (Massively Parallel Processing) based on 32-2048 DEC Alpha (21064 RISC, 150MHz) processors and a proprietary 3D torus interconnect
- 1997-ASCI Red at Sandia delivered first TFlop/s (on the Linpack benchmark) using Intel Pentium Pro processors and a custom interconnect.
- 2002-NECs Earth Simulator was a cluster of 640 vector supercomputers, delivered 35.61 TFlop/s on the Linpack benchmark.
- 2005-IBMs Blue Gene systems regained top rankings (again, according to Linpack) using massive numbers of embedded processors and communication subsystems (more later), each running a stripped down Linux-based operating system.
- 2008-IBM deployed a hybrid system of Opteron nodes coupled with Cell processors interconnected via Infiniband, achieved first sustained Peta Floating-Point Operations Per Second (PFlop/s) on top500 list
- 2010-Tianhe-1A at the National Supercomputing Center in Tianjin, China, mix of 14,336 Intel Xeon X5670 processors (86,016 cores) and 7168 Nvidia Tesla M2050 general purpose GPUs, custom (Arch) interconnect, 2.566 PFlop/s
- 2011-K computer, at RIKEN in Kobe, Japan, 68544 2.0GHz 8-core Sparc64 VIIIfx processors (548,352 cores), custom (Tofu) interconnect, 8.162 PFlop/s

1.4 Domains of High Performance Computing

Various domains of high performance computing are discussed in this section.

1.4.1 Cluster Computing

A computer cluster consists of a set of loosely connected computers that work together so that in many respects they can be viewed as a single system. The components of a cluster are usually connected to each other through fastlocal area networks ("LAN"), each node (computer used as a server) running its own instance of an operating system. Computer clusters emerged as a result of convergence of a number of computing trends including the availability of low cost microprocessors, high speed networks, and software for high performance distributed computing.[34]

Clusters are usually deployed to improve performance and availability over that of a single computer, while typically being much more cost-effective than single computers of comparable speed or availability.[34]

Now let's consider Japan's Earth Simulator shown in Fig.1.1 and IBM Blue Gene is shown in Fig.1.2. Fig.1.3 shows topology of devices of IBM Blue Gene. Both clusters dominated the Top500 List from 2002-2004 with following configuartion

- 640 8-processor SX-8 (vector) SMPs (peak of 8GFlop/s per processor)
- 10 TB of Memory
- Custom crossbar interconnect
- 700 TB disk + 1.2 PB Mass Storage
- Reportedly consumes about 12MW of power

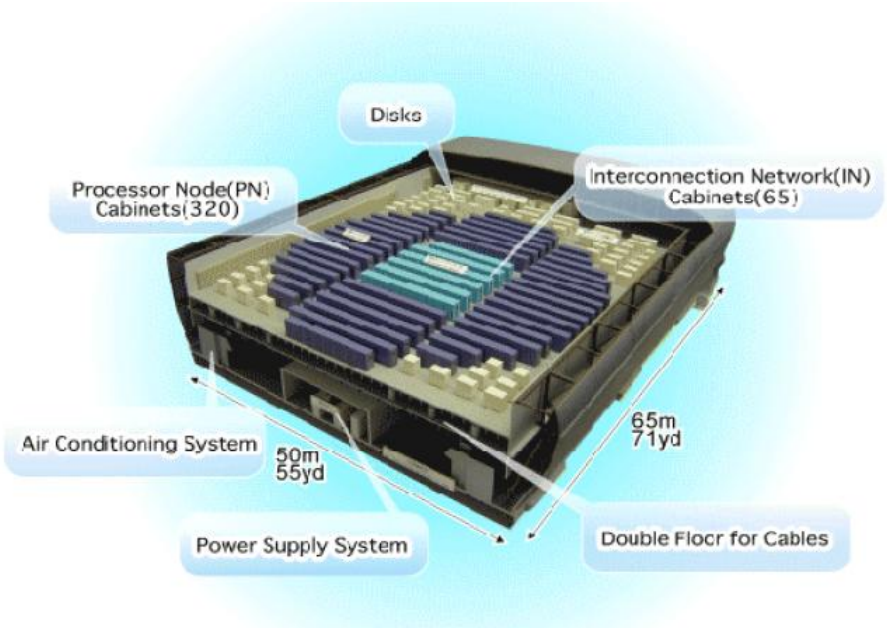

Figure 1.1: Japan's Earth Simulator[29]

Figure 1.2: IBM Blue Gene[30]

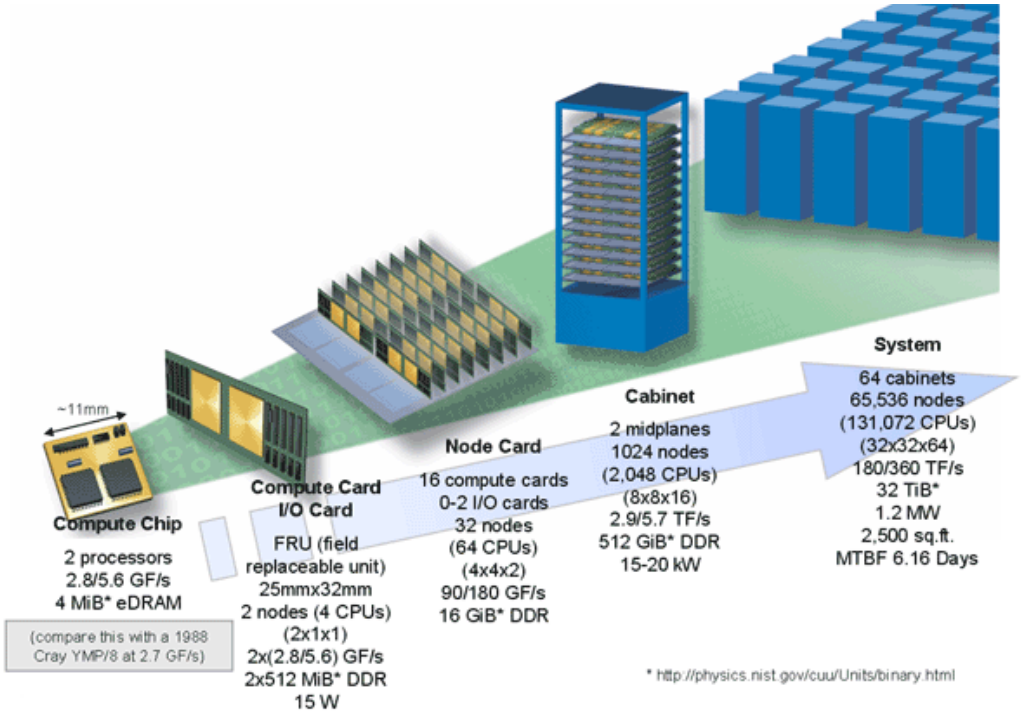

Figure 1.3: IBM Blue Gene Diagram[30]

1.4.2 Grid Computing

A grid computer is multiple number of same class of computers clustered together. A grid computer is connected through a super fast network and share the devices like disk drives, mass storage, printers and RAM. Grid Computing is a cost efficient solution with respect to Super Computing. Operating system has capability of parallelism

Grid computing combines computers from multiple administrative domains to reach a common goal, to solve a single task, and may then disappear just as quickly.

One of the main strategies of grid computing is to use middleware to divide and apportion pieces of a program among several computers, sometimes up to many thousands. Grid computing involves computation in a distributed fashion, which may also involve the aggregation of large-scale clusters. The size of a grid may vary from small-confined to a network of computer workstations within a corporation, for example-too large, public collaborations across many companies and networks. "The notion of a confined grid may also be known as an intra-nodes cooperation whilst the notion of a larger, wider grid may thus refer to an inter-nodes cooperation".

Grids are a form of distributed computing whereby a "super virtual computer" is composed of many networked loosely coupled computers acting together to perform very large tasks. This technology has been applied to computationally intensive scientific, mathematical, and academic problems through volunteer computing, and it is used in commercial enterprises for such diverse applications as drug discovery, economic forecasting, seismic analysis, and back office data processing in support for e-commerce and Web services.

Many distributed computing applications have been created, of which SETI@home and Folding@home are the best-known examples.

1.4.3 Multicore Computing

A multi-core 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 multi-core processor can issue multiple instructions per cycle from multiple instruction streams. Each core in a multi-core processor can potentially be superscalar as well that is, on every cycle, each core can issue multiple instructions from one instruction stream. Simultaneous multi-threading (of which Intel's HyperThreading is the best known) was an early form of pseudo-multi-coreism. 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. Fig.1.4 shows diagram of a generic dual-core processor.IBM's Cell microprocessor, designed for use in the Sony PlayStation 3, is another prominent multicore processor.

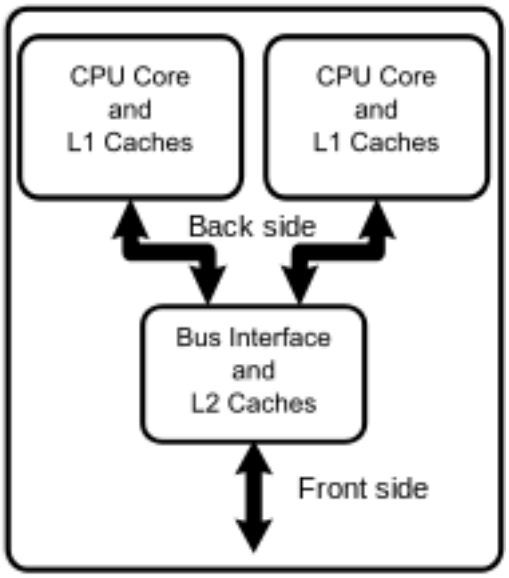

Figure 1.4: Generic dual-core processor

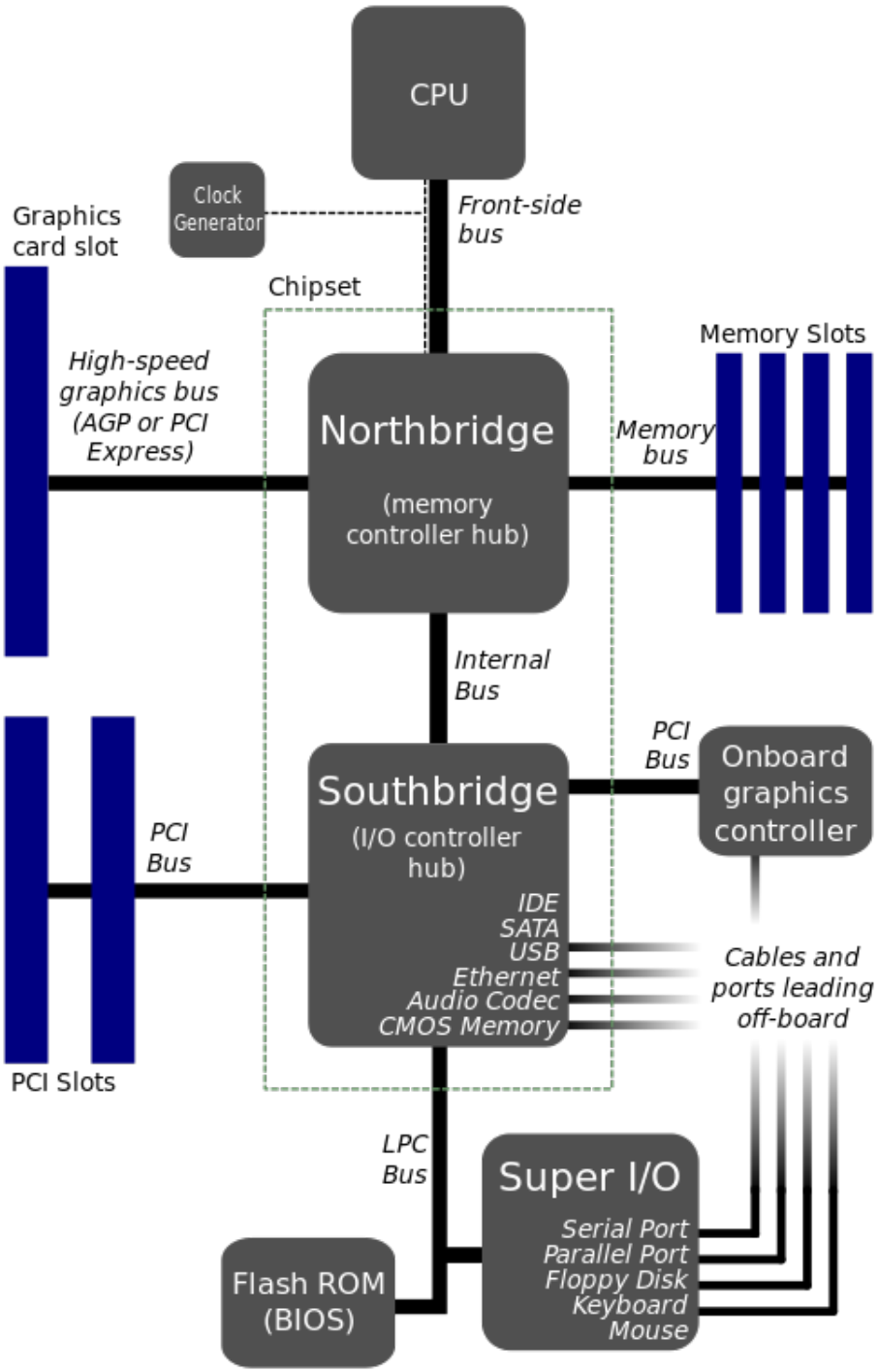

Figure 1.5: Connection Between Various Devices[23]

1.5 Parallel Computing

Parallel computing is a form of computation in which many calculations are carried out simultaneously,operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently ("in parallel"). There are several different forms of parallel computing: bit-level, instruction level, data, and task parallelism. Parallelism has been employed for many years, mainly in high-performance computing, but interest in it has grown lately due to the physical constraints preventing frequency scaling.As power consumption (and consequently heat generation) by computers has become a concern in recent years, parallel computing has become the dominant paradigm in computer architecture, mainly in the form of multicore processors.[22]

Parallel computers can be roughly classified according to the level at which the hardware supports parallelism, with multi-core and multi-processor computers having multiple processing elements within a single machine, while clusters, MPPs, and grids use multiple computers to work on the same task. Specialized parallel computer architectures are sometimes used alongside traditional processors, for accelerating specific tasks.Fig.1.5 shows block diagram of connection between various devices.[22]

Parallel computer programs are more difficult to write than sequential ones,[5] because concurrency introduces several new classes of potential software bugs, of which race conditions are the most common. Communication and synchronization between the different subtasks are typically some of the greatest obstacles to getting good parallel program performance. Nowadays, all of the hardware is parallel, as evident from following facts:

- Right now it is difficult to buy a computer that has only a single processor - even laptops have multiple cores.
- GPUs have a great many processing elements (Cell has 9, NVIDIA and ATI

offerings have hundreds).

- This increasing processor core count trend is going to continue for a while.

Some of the facts about the software that can actually concurrently use all of these hardware resources are as:

- Software is lagging behind the hardware.
- Some specialized parallel libraries for multicore systems.
- Some APIs for harnessing multiple cores (OpenMP) and multiple machines (MPI).
- Generally the software picture is one of tediously mapping applications to new architectures and machines.

1.5.1 Flynn's taxonomy

Michael J. Flynn created one of the earliest classification systems for parallel (and sequential) computers and programs, now known as Flynn's taxonomy. Flynn classified programs and computers by whether they were operating using a single set or multiple sets of instructions, whether or not those instructions were using a single or multiple sets of data. Based on that Flynn gave following taxonomy.Fig.1.6 to 1.9 shows graphical representation of Flynn's taxonomy.

	Single instruction	Multiple instruction
Single data	SISD	MISD
Multiple data	SIMD	MIMD

- **SISD-Single Instruction, Single Data**
  - Sequential (non-parallel) instruction flow
  - Predictable and deterministic

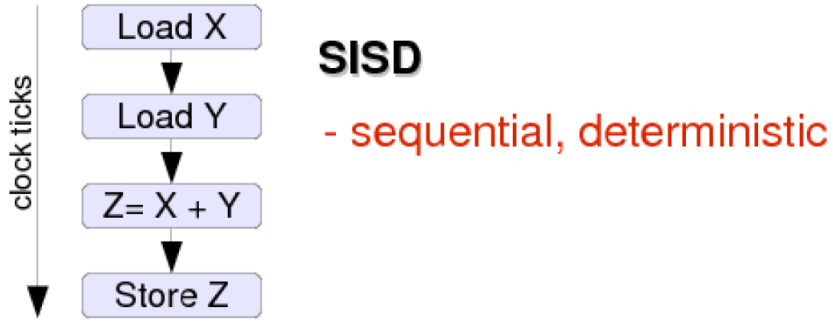

Figure 1.6: SISD flow

- **SIMD-Single Instruction, Multiple Data**
  - Single (typically assembly) instruction operates on different data in a given clock tick.
  - Requires predicatble data access patterns - "vectors" that are contiguous in memory.

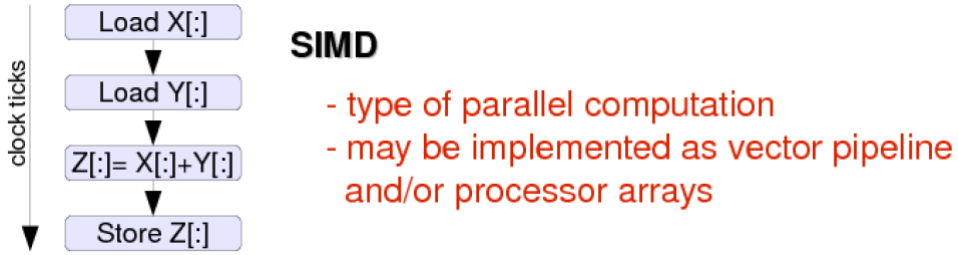

Figure 1.7: SIMD flow

- **MISD-Multiple Instruction, Single Data**

– Multiple instructions applied independently on same data.

– Theoretical rather than practical architecture - few implementations

Figure 1.8: MISD flow

- **MIMD-Multiple Instruction, Multiple Data**

– Multiple instructions applied independently on different data.

– Very flexible - can be asynchronous, non-deterministic

– Most modern supercomputers follow MIMD design, albeit with SIMD components/sub-systems

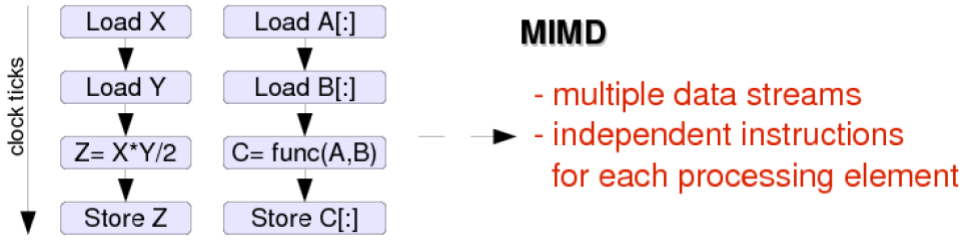

Figure 1.9: MIMD flow

1.6 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. 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.Fig.1.10 shows NVIDIA's Tesla GPGPU card. Fig.1.11 shows inside view of GPU. 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.[25]

Figure 1.10: Nvidia's Tesla GPGPU card[31]

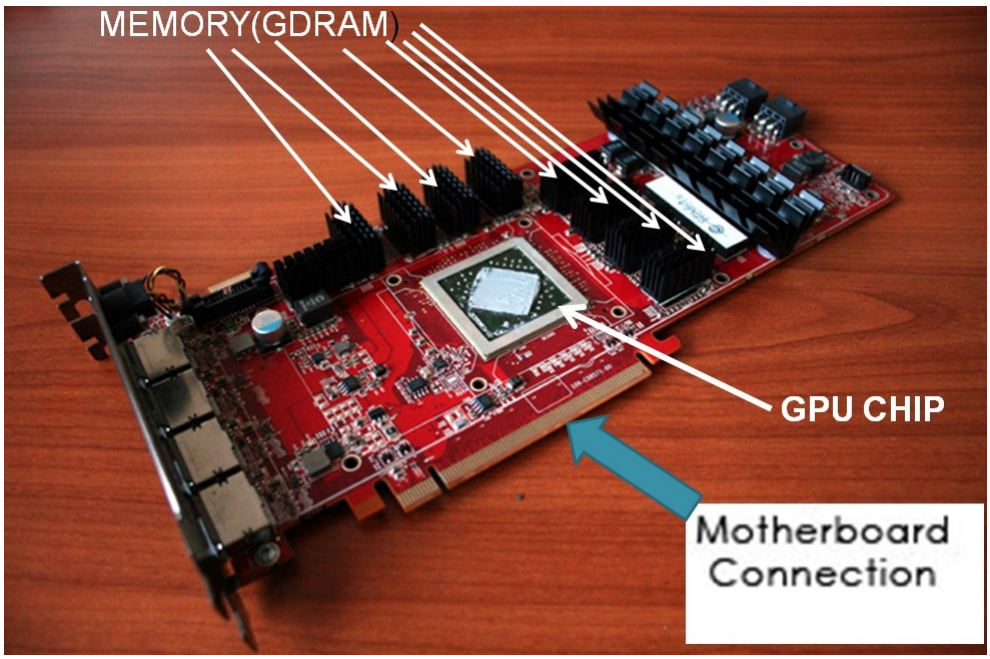

Figure 1.11: Inside View of GPU

1.7 Objective of Study

Numerical simulations based on the Finite Element Method (FEM) are often very time-consuming. Due to the fact that in the last decades FE Analysis is performed with greater accuracy and dynamic boundary conditions, efficient and highly parallel computation algorithms are still a major focus of current research. Also various structural engineering problems like Static analysis of skeletal structure using direct stiffness method, Static finite element analysis of plane stress and plate bending problems, Dynamic analysis of plane frame structures, Nonlinear analysis of structures etc. requires intense computations and thus parallel algorithms for all these problems is needed. Absence of such algorithms puts a hurdle for engineers to perform robust and accurate simulations promptly without delaying the overall engineering process.

The key objectives of study are as follows.

- To understand the parallel processing and its applications in structural engineering
- To understand OpenCL language for generating algorithms for parallel computing of various problems of structural engineering
- To study various numerical methods used in various analysis problems of structural engineering for writing parallel code
- To compare the performance of parallel code on various types of CPUs and GPUs.

1.8 Scope of Work

In order to achieve above objectives, the scope of work for major project is decided as follows.

- Understanding fundamentals of high performance computing and its terminology
- Study various techniques of parallel processing applicable in structural engineering
- Understand architecture of GPU and mylti-core processors for developing appropriate algorithms
- Understand OpenCL language and its specifications
- Development of computer program for numerical methods like Gauss Elimination which can run on GPU and multi-core processors
- Development of computer program for analysis of structures which can run effectively on GPU and multi-core processors

- Comparison of runtime and efficiency of computer program on different hardware configurations.

1.9 Organization of Report

The study carried out in this major project is related to the application of parallel processing in structural engineering. Study includes different type of parallel algorithms useful for solving various problems of structural engineering. The content of major project is divided into different chapter as follows.

Chapter 1, gives introduction of High Performance Computing , its history and evolutions in HPC. It also covers domains of High Performance Computing with examples of each. Introduction to parallel computing and GPU and other high performance computing machines are presented in this chapter.

Chapter 2, covers literature review. In this chapter, literature related to parallel processing is reviewed and work carried out by various researchers is presented. It gives idea about the work carried out in various areas of parallel processing through variety of problems.

Chapter 3, contains introduction to OpenCL Language and its specifications. There is brief discussion about benifits of OpenCL, its architecture. A code of matrix multiplication using GPU is prepared and its results are compared with CPU.

Chapter 4, contains study of Gaussian Elimination method using different devices for parallel computing.Comparison of speedup and efficiency of parallel code is also presented in this chapter.

Chapter 5, contains study of Half-Band matrix solver using different devices for par-

allel computing. Large size problems of plane frame and space frame analysis are implemented on computers with multiple cores and GPU . Comparison of speedup and efficiency of parallel code is also presented in this chapter.

Chapter 6, contains brief summary of the project and conclusions. Future scope of work is also discussed in this chapter.

Chapter 2

Literature Survey

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 algorithms originally designed to run on sequential machines as well as to develop new methods that take advantage of the parallelism offered by the multiprocessing computers. This work is concerned with some of the parallel algorithms that have been developed in this field. More specifically, it is a survey of parallel algorithms that are of interest to structural engineering. Such algorithms include parallel solvers direct and iterative! for linear systems of algebraic equations, techniques for the parallelization of the finite element method, and concurrent time-stepping algorithms for the solution of the equations arising in structural dynamic problems.

The rapid rate at which technology is evolving has led to a need for more sophisticated techniques that can take full advantage of the hardware available. Traditionally, software has lagged behind hardware and there is a continuous need to maximize software performance on multiprocessor systems. In chapter, work carried out by researchers about application of parallel processing/computing in field of engineering are discussed to have an idea about evolution of parallel processing and its current trends and its impact in field of structural engineering.

2.1 Parallel Computing

Sotelino[1] gave an idea about parallel processing techniques in structural engineering with various practical examples of application.Various parallel processing techniques in Finite Element Analysis such as Sub-structuring, Operator splitting, Element-by-element (EBE) strategies were explained.There was good discussion on Iterative solvers, Direct Solvers and Parallel Equation Solvers.The discussions in the sections Iterative Solvers and Direct Solvers were concerned with the solution of a system of linear algebraic equations.An attempt was made to provide a thorough survey of the methods that were directly related to structural engineering applications.

Hajjarz and Abel[2] presented strategy for the solution of the fully nonlinear transient structural dynamics problem in a coarse-grained parallel processing environment. Emphasis was placed on the analysis of three-dimensional framed structures subjected to seismic loading. Study included long-duration dynamic loading, geometric and material nonlinearity, and the wide distribution of vibrational frequencies found in frame models. The implicit domain decomposition method described employs sub-structuring techniques and then a preconditioned conjugate gradient algorithm for the iterative solution of the reduced set of unknowns along the substructure interfaces.The domain decomposition algorithm provided an efficient means for solving the fully nonlinear transient structural dynamics problem in parallel. The natural preconditioner which arised from sub-structuring analysis effectively reduced the condition number of the interface coefficient matrix sufficiently to allow an iterative conjugate gradient algorithm to be used for its solution. The iterative algorithm might be easily streamlined for parallel processing since it consisted exclusively of vector operations.

Bahcecioclu and Kurc[3] did Nonlinear dynamic finite element analysis with GPU using CUDA language. Newmark family of algorithms have been utilized by many engineering applications for the solution of nonlinear dynamic analysis of various structural models. Dynamic and nonlinear nature of such problems and numerical stability requirements of the algorithms increase the need for computation power in order to achieve practical solution times. Thus, this study intended 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 combined 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.

Kandasamy and Konig[4] presented an approach for meshing with the help of GPU for robust simulations in engineering field. Parallel mesh generation for FE element application was still an active research topic to compensate the demand for large scale, dynamic and real-time FE problem analyses. Even though many different CPU based parallel meshing implementation exist, the parallel computing power of graphic card processor unit (GPU) had not yet fully approached for meshing applications. The main focus of this paper was on presenting a research concept of parallel FE mesh generation using multiple GPUs and promising preliminary results of GPU adopted Delaunay triangulator. There were only few approaches of Delaunay triangulation employing GPU and CPU, but they were not utilized for a FE meshing application. This work was basically a 2D Delaunay mesh generator by incorporating the work

from GPU-DT triangulation, which could employ on a single card. They presented a mesh partitioning technique, an interfacing approach between subdomains, a parallel Delaunay triangulation by employing multiple GPUs and a validation method for Delaunay triangulation by edge flipping.

Kruzel and Banas[5] analyzed computational aspects of the problem of numerical integration in finite element calculations and considered an OpenCL implementation of related algorithms for processors with wide vector registers. As a platform for testing the implementation they choose the PowerXCell processor, being an example of the Cell Broadband Engine (CellBE) architecture. Although the processor was considered old for todays standards (its design dates back to year 2001), they investigated its performance due to two features that it shares with recent Xeon Phi family of coprocessors: 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, such as contemporary x86 microprocessors. They considered higher order finite element approximations and implemented the standard algorithm of numerical integration for prismatic elements.The obtained range of performance numbers showed that in many situations high utilization of vector capabilities could be achieved. This seems to be an important conclusion in light of a recently observed trend to equip standard processor cores with wide vector registers and execution units. Another conclusion was that OpenCL could be used for relatively simple porting of scientific codes to complex heterogeneous multi-core architectures, such as CBE. Moreover, OpenCL allowed one to obtain a high performance code, due to the support of explicit memory hierarchy management and vector operations.

Yang et al.[6] 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 implemented 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.Results showed that the FPGA implementation had the highest efficiency with respect to clock cycles compared with their pure GPU implementation, a hybrid system with MAGMA, and a CPU with LAPACK. The GPU implementation was better than other implementations using MAGMA and LAPACK library for small matrices, and the hybrid system with MAGMA was the best for larger matrices.

Wang et al.[7] presented the GPU parallelization of complex three-dimensional software for nonlinear analysis of concrete structures. It focused on coupled thermo-mechanical analysis of complex structures. A coupled FEM/DEM approach (CDEM) was given from a fundamental theoretical viewpoint. As the modeling of a large structure by means of FEM/DEM may lead to prohibitive computation times, a parallelization strategy was required. With the substantial development of computer science, a GPU-based parallel procedure was implemented. A 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 were gained from GPU parallelization.

Hsieh et al.[8] presented general sparse matrix and parallel computing technologies for a 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 requirements 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 pub-

licly available parallel equation solver, called SPOOLES, to directly solve the sparse finite element equations, while the 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 have 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 requirements in large-scale finite element structural analysis.

2.2 Application of Parallel Computing on FEM

Qian et al.[9] carried out research of Parallel Computing for Large-scale Finite Element Model of WheelRail Rolling Contact.The parallel computing methods of contact problem were analyzed firstly. Then, the contact algorithm and parallel computing of ABAQUS was introduced. Fig.2.1 shows logical structure of cluster used.

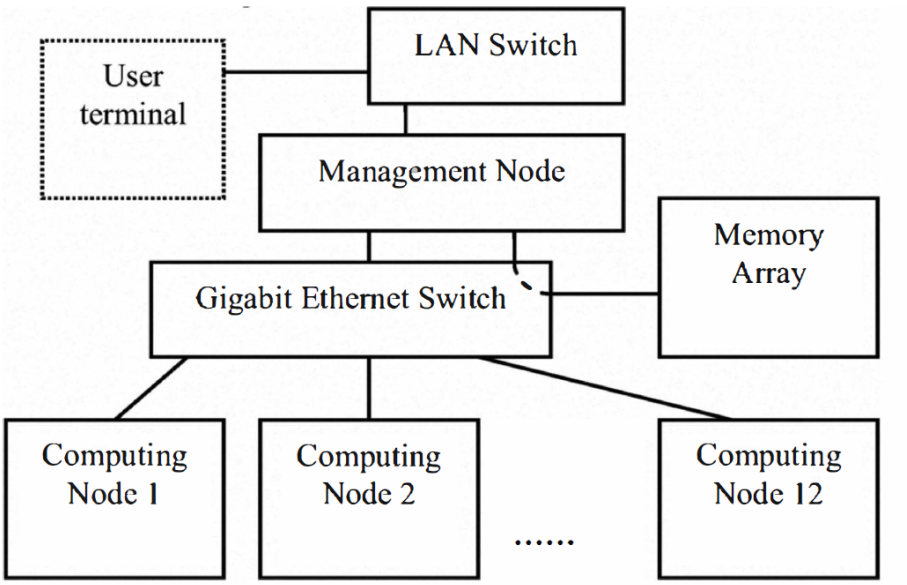

Figure 2.1: Logical Structure of Cluster[9]

The parallel computing environment using MPI in ABAQUS was put forward. On the basis of cluster, some different finite element model was solved by implicit and explicit solution. It was found that the mesh size of wheel/rail contact field was refined to 0.75mm in order to ensure accuracy for engineering.At last, the parallel computing for the contact problem of wheel/rail was discussed using the speedup and efficiency.

Fan et al.[10] presented application of parallel computing in Large Eigenvalue Problems for Engineering Structures.A parallel solving system was constructed via integrating predominant algorithms and their corresponding software packages into the finite-element parallel computing frameworkPANDA. The finite element model (FEM) of engineering structures was built in preprocessing commercial softwareMSC Patran. Based on the 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 showed that PANDA frame was competent for carrying out large-scale parallel computing of eigenvalue problems; in virtue of supercomputer, the computing scale attains millions of freedom degrees; and the parallel efficiency was favorable.They gave a brief review on some dominant algorithms and freely available software for the 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 freedom degree of the finite element model was about 2.3 million.

Fu[11] discussed implementation of distributed finite element method over cluster of workstation. A variety of parallel algorithms for finite element analysis were studied, in which the domain decomposition method that having relatively coarse granularity was suited for the distributed environment. With this scheme, the entire domain to be solved was divided into several sub-domains and each sub-domain was assigned to one of the processors engaged in the parallel computing.Using the developed code, a

dam structural analysis problem was solved on workstation cluster. From the performance test, the effectiveness of the distributed parallel computing algorithm of finite element method was verified. Important factors affecting the performance of the distributed parallel computing were found and analyzed.

On the basis of mode synthesis analysis, parallel algorithm of solving large-scale structural eigenproblem was presented by **Chaojiang Fu**[12]. The eigen value problem of the structure was solved using subspace iterative parallel method. The substructure subspace iterative method was implemented using the stiffness matrix and mass matrix of the substructures without assembling the stiffness and mass matrix of whole structure. 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 processors 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 a 100Mbps Ethernet interconnect. The MPI communication libraries have been used to manage the message passing.

2.3 Parallel Solvers

Leow et al.[13] presented parallel implementation of a direct method for solving Linear equations called Gaussian Elimination, which consists of forward elimination and back substitution.The solution of a linear system of equations constitutes 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. Thorough 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 reduced by a factor of 183X 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 8192 × 8192

Sharma et al.[14] presented the Gauss Jordan algorithm 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 and the performance metrics were studied and compared with CPU based parallel methods. They showed that the time complexity of matrix inversion scales as n as long as n^2 threads can be supported by the GPU. Matrix inversion was an essential step in a wide range of numerical problems starting with solving linear equations , structural analyses using finite element method , 3D rendering ,digital filtering , image filtering and image processing and constitutes an indispensable component in almost all mathematical/statistical software suites. Some of the common available algorithms for computing the inverse of a matrix were Strassen, Strassen-Newton, Gaussian elimination, GaussJordan, Coppersmith and Winograd, LUP Decomposition, Cholesky decomposition, QR decomposition, RRQR factorization, Monte Carlo Methods for inverse etc.The ability to invert large matrices accurately and quickly determines the effectiveness of a wide range of computational algorithms and products. GPU computing was ideally suited for massively parallel tasks as the thread creation and memory transfer overheads were negligible. They had redesigned the Gauss Jordan algorithm for matrix inversion on GPU based CUDA platform, tested it on five different types of matrices (identity, sparse, banded, random and hollow) of various sizes. Computation time for inverting different types of matrices was presented in the paper.

Reddy et al.[15] described the design and the implementation of parallel routines in the 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 im-

portant feature of the library, which was the automation of the difficult optimization tasks of parallel programming on heterogeneous computing clusters. Other features were the determination of the accurate values of the platform parameters such as the speeds of the processors and the latencies and bandwidths 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.They described this process of automation.The Heterogeneous ScaLAPACK program used the multiprocessing approach, where more than one process executed on each processor. The number of processes to run on each processor during the program startup was determined automatically by the Heterogeneous ScaLAPACK command-line interface tools.

Stefanski et al.[16] 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 implementations 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 GPUs can be executed on multi-core CPUs, a direct porting did not provide satisfactory performance due to an application of architecture specific features 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.[17] used Gauss elimination and the 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 systems were presented. Sparse systems were not considered here although, these can obviously be handled.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 times could be achieved or, alternatively, larger problems could be solved.

For the purpose of this work, the MPP was configured as an 128 $\times$ 128 array with a 32 bit word length. For the solution of linear systems, the two most important aspects related to the MPP were the number of memory planes in the ARray Unit (ARU) and the size of the staging memory. The ARU contains 900 usable bit planes of memory. This limits the number of real arrays (128*128, 32 bit) in the ARU to 28. The staging memory was limited to 512 real arrays. Parallel Pascal callable $I/0$ procedures could transfer only one 128 $\times$ 128 array in or out of the ARU at any one time. This makes it necessary for any array larger than 128 $\times$ 128 to be blocked into sub-arrays of 128 $\times$ 128. Thus, the smallest system considered is a 128 $\times$ 128 system of equations.

Mani et al[18] proposed a parallel Gaussian elimination technique for the solution of linear 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 operations applied to

the elements of A and the arcs represent the precedence relation that exists among the operations in the solution process. This graph gave the clear picture to the 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 of 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.

The usefulness of acyclic directed graph in identifying parallel operations, computing the minimum completion time, the minimum number of processors to complete the graph in minimum time and the maximum achievable speedup were presented. The absolute minimum completion time was dependent on the number of equations and independent of the bandwidth. On the other hand, maximum achievable speedup and the optimal number of processors required to complete the job in minimum time were dependent on the half bandwidth and was independent of the number of equations. A method of incorporating the inter-processor communication time and its effect on the overall computation time was also brought out. This study was useful for engineers working with large system of equations on a multiprocessor system.

McGinn et al.[19] presented a 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 from the results that the impact on performance that occurs as one changes the size of Matrix i.e n. When there was increase the value of n, the MPI program displays an improvement in performance as opposed 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 will show a greater increase in performance than the shared platform.

Liu et al.[20] designed and developed a GPU based Bi-Conjugate Gradient STA-Bilized (BiCGSTAB) solver that meets both generality and scalability requirements. It was well suited for all types of banded linear systems. And this solver combined a new matrix decomposition method with several optimizations 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 the linear systems that can be partitioned into independent subsystems.They designed a number of GPU and MPI optimizations to speedup inter-GPU and inter-machine communications and evaluated the solver on Poisson equation and advection diffusion equation as well as several other banded linear 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 overhead, can scale upto 32 GPUs on Amazon EC2 with relatively slow ethernet network.

Zhang et al.[21] presented a GPU based parallel Jacobis iterative solver for dense linear equations.Modern GPUs are high performance many-core processors fit for large scale parallel computing. They provided a novel way for accelerating the solving process.First, they introduced the backgrounds for accelerating linear equations solver together with GPUs and the corresponding parallel platform CUDA on it. Then implementation of Jacobis iterative method on CUDA was discussed. They compared the experimental results of CUDA programs on GPU with traditional programs on CPU. Experiments showed that it obtained a speedup of approximately 59 times with single floating point at a low precision, 19 times with double at a high precision.

2.4 Summary

In this chapter literature on parallel computing, parallel solvers 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 OpenCL Programming

3.1 General

The Open Computing Language (OpenCL) is an open and royalty-free parallel computing API designed to enable GPUs and other coprocessors to work in tandem with the CPU, providing additional raw computing power. As a standard, OpenCL 1.0 was released on December 8, 2008, by The Khronos Group, an independent standards consortium. Developers have long sought to divide computing problems into a mix of concurrent subsets, making it feasible for a GPU to be used as a math coprocessor working with the CPU to handle general problems efficiently. The potential of this heterogeneous computing model was encumbered by the fact that programmers could only choose proprietary programming languages, limiting their ability to write vendor-neutral, cross-platform applications. Proprietary implementations such as NVIDIA's CUDA limited the hardware choices of developers wishing to run their application on another system without having to retool it.[24][26]

3.1.1 Benefits of OpenCL

A primary benefit of OpenCL is substantial acceleration in parallel processing. OpenCL takes all computational resources, such as multi-core CPUs and GPUs, as peer computational units and correspondingly allocates different levels of memory, taking advantage of the resources available in the system. OpenCL also complements the existing OpenGL visualization API by sharing data structures and memory locations without any copy or conversion overhead. A second benefit of OpenCL is cross-vendor software portability. This low-level layer draws an explicit line between hardware and the upper software layer. All the hardware implementation specifics, such as drivers and runtime, are invisible to the upper-level software programmers through the use of high-level abstractions, allowing the developer to take advantage of the best hardware without having to reshuffle the upper software infrastructure. The change from proprietary programming to open standard also contributes to the acceleration of general computation in a cross-vendor fashion.[24][26]

3.2 Modules of OpenCL

The OpenCL development framework is made up of three main parts:

1. Language specification
2. Platform layer API
3. Runtime API

3.2.1 Language Specification

The language specification describes the syntax and programming interface for writing kernel programs that run on the supported accelerator (GPU, multi-core CPU, or DSP). Kernels can be precompiled or the developer can allow the OpenCL to compile the kernel program at runtime.[24][26]

3.2.2 Platform API

The platform-layer API gives the developer access to software application routines that can query the system for the existence of OpenCL-supported devices. This layer also lets the developer use the concepts of device context and work-queues to select and initialize OpenCL devices, submit work to the devices, and enable data transfer to and from the devices.[24][26]

3.2.3 Runtime API

The OpenCL framework uses contexts to manage one or more OpenCL devices. The runtime API uses contexts for managing objects such as command queues, memory objects, and kernel objects, as well as for executing kernels on one or more devices specified in the context.

3.3 OpenCL Architecture

3.3.1 The Platform Model

The OpenCL platform model is defined as a host connected to one or more OpenCL devices. Fig.3.1 shows OpenCL Platform Model which comprises one host plus multiple compute devices, each having multiple compute units, each of which have multiple processing elements. A host is any computer with a CPU running a standard operating system. OpenCL devices can be a GPU, DSP, or a multi-core CPU. An OpenCL device consists of a collection of one or more compute units (cores). A compute unit is further composed of one or more processing elements. Processing elements execute instructions as SIMD (Single Instruction, Multiple Data) or SPMD (Single Program, Multiple Data). SPMD instructions are typically executed on general purpose devices such as CPUs, while SIMD instructions require a vector processor such as a GPU or vector units in a CPU.[24][26]

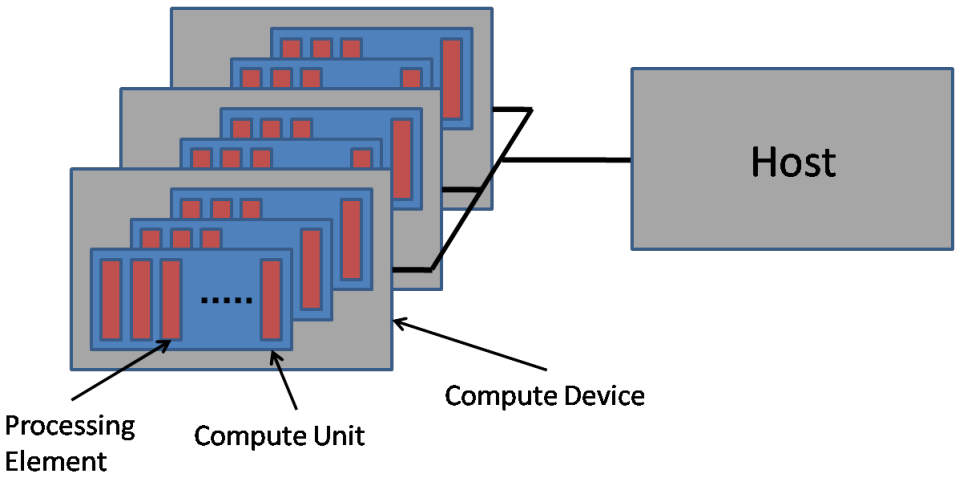

Figure 3.1: OpenCL Platform Model[24][26]

Fig.3.2 shows ATI RadeonTM HD 5870 GPU architecture illustrating a compute device construct. The ATI Radeon HD 5870 GPU is made up of 20 SIMD units, which translates to 20 compute units in OpenCL. Each SIMD unit contains 16 stream cores, and each stream core houses five processing elements. Thus, each compute unit in the ATI Radeon HD 5870 has 80 (16 $\times$ 5) processing elements.

3.3.2 The Execution Model

The OpenCL execution model comprises two components: kernels and host programs. Kernels are the basic unit of executable code that runs on one or more OpenCL devices. Kernels are similar to a C function that can be data- or task-parallel. The host program executed on the host system, defines devices context, and queues kernel execution instances using command queues. Kernels are queued in-order, but can be executed in-order or out-of-order.[24][26]

3.3.2.1 Kernels

OpenCL exploits parallel computation on compute devices by defining the problem into an N-dimensional index space. When a kernel is queued for execution by the host

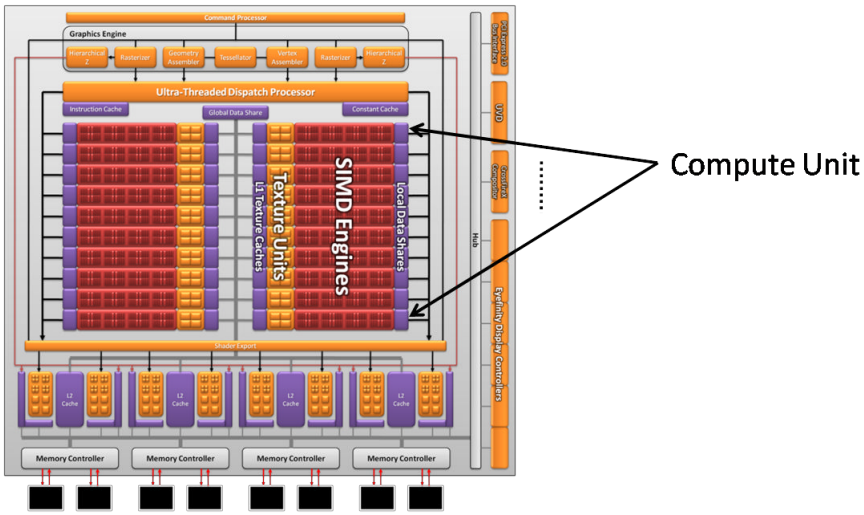

Figure 3.2: ATI RadeonTM HD 5870 GPU architecture[24][26]

program, an index space is defined. Each independent element of execution in this index space is called a work-item. Each work-item executes the same kernel function but on different data. When a kernel command is placed into the command queue, an index space must be defined to let the device keep track of the total number of work- items that require execution. The N-dimensional index space can be N=1, 2, or 3. Processing a linear array of data would be considered N=1; processing an image would be N=2, and processing a 3D volume would be N=3. Processing a 1024x1024 image would be handled this way: The global index space comprises a 2-dimensional space of 1024 by 1024 consisting of 1 kernel execution (or work-item) per pixel with a total of 1,048,576 executions. Within this index space, each work-item is assigned a unique global ID. The work-item for pixel x=30, y=22 would have global ID of (30,22). OpenCL also allows grouping of work-items together into work-groups, as shown in the Fig.3.3 Fig.3.4. The size of each work-group is defined by its own local index space. All work-items in the same work-group are executed together on the same device. The reason for executing on one device is to allow work-items to share local memory and synchronization. Global work-items are independent and cannot by synchronized. Synchronization is only allowed between the work-items in a work-

group. The following example shows a two-dimensional image with a global size of

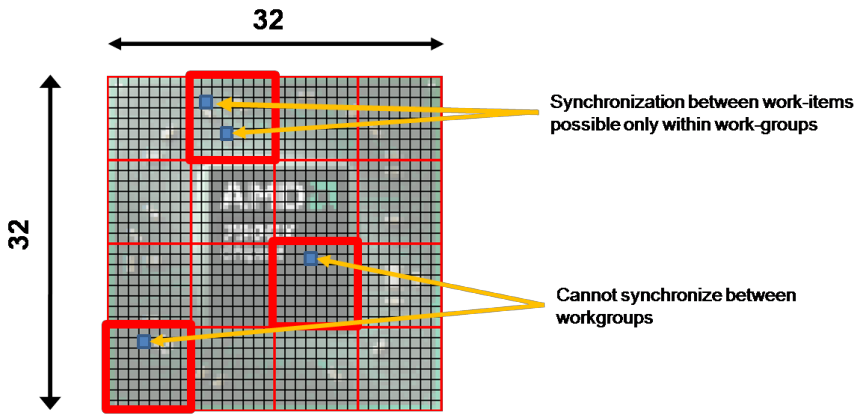

Figure 3.3: Grouping Work-items Into Work-groups[24][26]

1024 (32x32). The index space is divided into 16 work-groups. The highlighted work-group has an ID of (3,1) and a local size of 64 (8x8). The highlighted work-item in the work- group has a local ID of (4,2), but can also be addressed by its global ID of (28,10).[24][26]

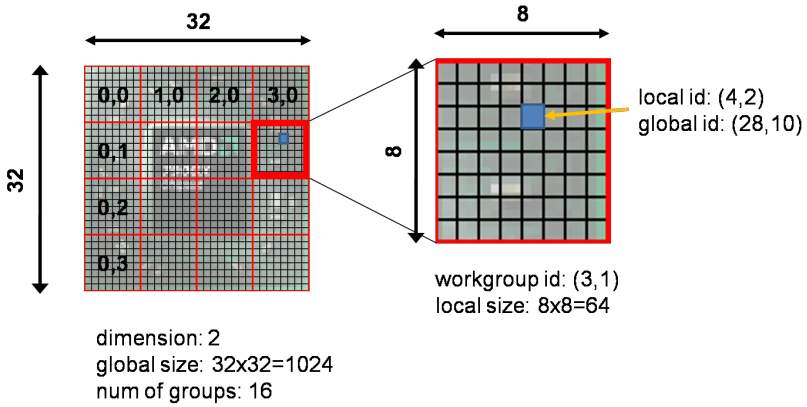

Figure 3.4: Work-group Example[24][26]

3.3.2.2 Host Program

The host program is responsible for setting up and managing the execution of kernels on the OpenCL device through the use of context. Using the OpenCL API, the host can create and manipulate the context by including the following resources:

- **Devices:**A set of OpenCL devices used by the host to execute kernels.
- **Program Objects:**The program source or program object that implements a kernel or collection of kernels.
- **Kernels:**The specific OpenCL functions that execute on the OpenCL device.
- **Memory Objects:**A set of memory buffers or memory maps common to the host and OpenCL devices.

After the context is created, command queues are created to manage execution of the kernels on the OpenCL devices that were associated with the context. Command queues accept three types of commands:

- Kernel execution commands run the kernel command on the OpenCL devices.
- Memory commands transfer memory objects between the memory space of the host and the memory space of the OpenCL devices.
- Synchronization commands define the order in which commands are executed.

Commands are placed into the command queue in-order and execute either in-order or out-of-order. In case of in-order mode, the commands are executed serially as they are placed onto the queue. In out-of-order mode, the order the commands execute is based on the synchronization constraints placed on the command.

3.3.3 The Memory Model

Since common memory address space is unavailable on the host and the OpenCL devices, the OpenCL memory model defines four regions of memory accessible to

work-items when executing a kernel.[24][26] The Fig.3.5 shows the regions of memory accessible by the host and the compute device:

Global memory is a memory region in which all work-items and work-groups

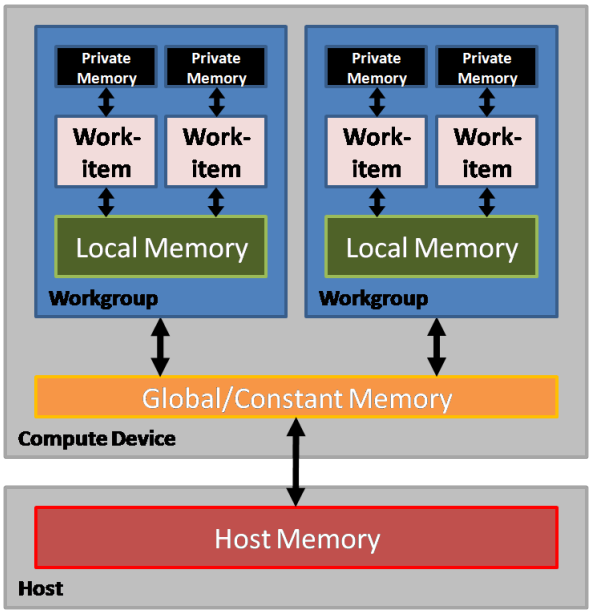

Figure 3.5: OpenCL Memory Model[24][26]

have read and write access on both the compute device and the host. This region of memory can be allocated only by the host during runtime.

Constant memory is a region of global memory that stays constant throughout the execution of the kernel. Work-items have only read access to this region. The host is permitted both read and write access.

Local memory is a region of memory used for data-sharing by work-items in a work-group. All work-items in the same work-group have both read and write access.

Private memory is a region that is accessible to only one work-item.

In most cases, host memory and compute device memory are independent of one another. Thus, memory management must be explicit to allow the sharing of data between the host and the compute device. This means that data must be explicitly moved from host memory to global memory to local memory and back. This process

works by enqueuing read/write commands in the command queue. The commands placed into the queue can either be blocking or non-blocking. Blocking means that the host memory command waits until the memory transaction is complete before continuing. Non-blocking means the host simply puts the command in the queue and continues, not waiting until the memory transaction is complete.[24][26]

3.4 Executing an OpenCL Program

The OpenCL framework is divided into a platform layer API and runtime API. The platform API allows applications to query for OpenCL devices and manage them through a context. The runtime API makes use of the context to manage the execution of kernels on OpenCL devices.[24][26]The basic steps involved in creating any OpenCL program are shown in Fig.3.6.

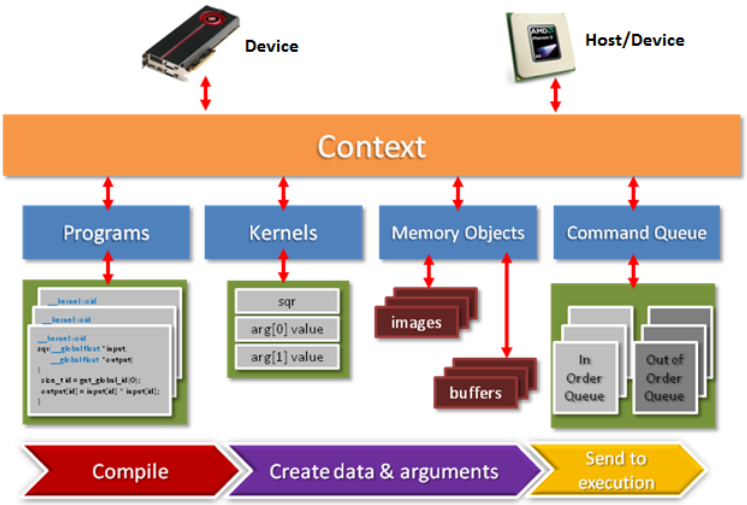

Figure 3.6: OpenCL Execution model[24][26]

The execution of OpenCL program is carried out in following steps:

1. Query the host system for OpenCL devices.
2. Create a context to associate the OpenCL devices.
3. Create programs that will run on one or more associated devices.
4. From the programs, select kernels to execute.
5. Create memory objects on the host or on the device.
6. Copy memory data to the device as needed.
7. Provide arguments for the kernels.
8. Submit the kernels to the command queue for execution.
9. Copy the results from the device to the host.

3.5 Multiplication of Large Square Matrices

Execution of OpenCL program is illustrated through matrix multiplication application. The task at hand is standard, i.e. to multiply two matrices. It is chosen primarily due to the fact that quite a lot of information on the subject can be found in different sources. Most of them, one way or another, offer more or less coordinated solutions. The further discussion will give step-by-step clarifications of OpenCL architecture, its memory model and programming through example of Square Matrix Multiplication .

Below is a matrix multiplication formula well-known in linear algebra, modified for computer calculations. The first index is the matrix row number, the second index is the column number. Every output matrix element is calculated by sequentially adding each successive product of elements in the first and second matrices to the

accumulated sum. Eventually, this accumulated sum is the calculated output matrix element:

$C_{i,j} = C_{i,j} + \sum_{k=o}^{P} A_{i,k} * B_{k,j}$

$0 \leq i \leq N$

$0 \leq j \leq M$

It can schematically be represented as shown in Fig.3.7:

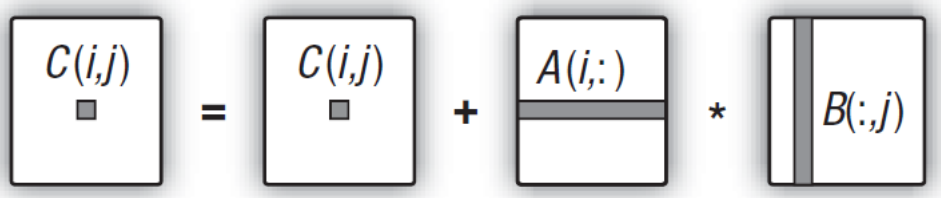

Figure 3.7: Matrix multiplication algorithm

3.6 Parallel Implementation

Since to create three linear buffers for the OpenCL kernel, it would be reasonable to rework the initial algorithm so that it is as similar to the kernel algorithm as possible. The code of the "non-parallel" program on a 'ingle core CPU" with linear buffers is provided together with the kernel code. The optimality of the code with two-dimensional arrays does not mean that its analog will also be optimal for linear buffers: all tests will have to be repeated. To avoid a possible matrix/buffer element addressing confusion, a Matrix (M rows by N columns) is laid out in global GPU memory as a linear buffer. One needs to calculate a linear shift of an element Matrix[row][column].

There is in fact no fixed order of laying out a matrix in GPU memory since it is determined by the logic of the problem alone. For example, elements of both matrices

could be laid out differently in buffers because as far as the matrix multiplication algorithm is concerned, matrices are asymmetrical, i.e. the rows of the first matrix are multiplied by the columns of the second matrix. Such rearrangement can greatly affect the calculation performance in sequential reading of matrix elements from global GPU memory in every iteration of the kernel.

The first implementation of the algorithm will feature matrices laid out in the same manner - in row-major order. The first row elements will be first to be placed into the buffer followed by all elements of the second row and so on. The formula of flattening a 2-dimensional representation of a matrix Matr[M(rows)][N(columns)] onto linear memory is as shown in Fig.3.8.

3.6.1 OpenCL Kernel

The OpenCL kernel for matrix multiplication looks as follows:

```
__kernel
void matrixMultiplication(__global float* A, __global float*
   B, __global float* C,  int widthA, int widthB )
{
        int i = get_global_id(0);
        int j = get_global_id(1);
        float value=0;
        for ( int k = 0; k < widthA; k++)
        {
                value = value + A[k + j * widthA] * B[k*
                   widthB + i];     }
        C[i + widthA * j] = value;
}
```

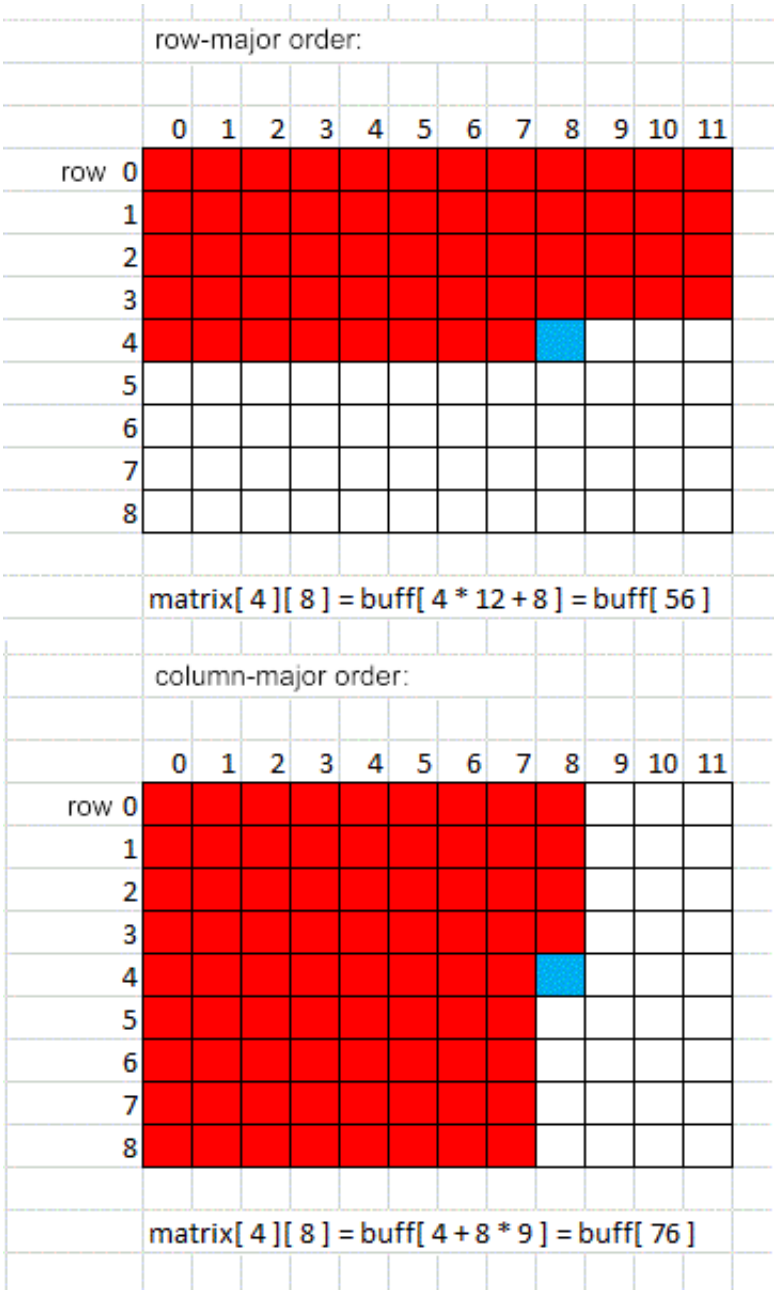

Figure 3.8: Algorithm for converting a two-dimensional index space into linear for laying the matrix out in the GPU buffer with Row-Major And Column Major

3.6.2 OpenCL Source code for C++

```
//Program to multiply two matrices using OpenCL in GPU

//#include "stdafx.h"
#include < stdio.h >
#include < stdlib.h >
#include < time.h >
#include < ctime >
#include <time.h>

#define widthA 512
#define heightA 512
#define widthB heightA
#define heightB 512
#define widthC widthA
#define heightC heightB

#ifdef __APPLE__
#include < OpenCL/opencl.h >
#else
#include < CL/cl.h >
#endif

#define MEM_SIZE (128)
#define MAX_SOURCE_SIZE (0x100000)

int main()
{
```

```
float * A = (float *)malloc(sizeof(float)*widthA*heightA);
float * B = (float *)malloc(sizeof(float)*widthB*heightB);
float * C = (float *)malloc(sizeof(float)*widthC*heightC);
float * Res = (float *)malloc(sizeof(float)*widthC*heightC)
   ;
float * D= (float *)malloc(sizeof(float)*widthC*heightC);

 FILE * fp1 = fopen("matAdata.csv", "w");
if (!fp1) {
  fprintf(stderr, "Failed to open matAdata.\n");
  exit(1);
 }
clock_t b1=clock();
float p=1;
for(int i = 0;i < widthA; i++)
{
                for(int j=0;j<heightA;j++)                {

                        *(A+i*heightA+j)=p;
                        fprintf(fp1, "%f,",*(A+i*heightA+j));
                        p++;
                }
                fprintf(fp1, "\n");
 }
 fclose(fp1);

 fp1 = fopen("matBdata.csv", "w");
 if (!fp1) {
  fprintf(stderr, "Failed to open matAdata.\n");
```

```
    exit(1);
   }
   float q=1;
        for(int i = 0;i < widthB; i++)
        {
                for(int j=0; j<heightB;j++)                {

                        *((B+i*heightB+j))=q;
                        fprintf(fp1, "%f," ,*(B+i*heightA+j));
                        q++;
                }
                fprintf(fp1, "\n");
        }
        fclose(fp1);
clock_t e1=clock();
double t1=((e1-b1)*1000)/CLOCKS_PER_SEC;;
printf("Time elapsed for initialisation:%f\n",t1);
  cl_device_id device_id = NULL;
  cl_context context = NULL;
  cl_command_queue command_queue = NULL;
  cl_mem memobjA = NULL;
  cl_mem memobjB = NULL;
  cl_mem memobjC = NULL;
  cl_mem rowA = NULL;
  cl_mem colC = NULL;
  cl_program program = NULL;
  cl_kernel kernel = NULL;
  cl_platform_id platform_id = NULL;
  cl_uint ret_num_devices;
```

```
cl_uint ret_num_platforms;
cl_int ret;

//char string[MEM_SIZE];

FILE *fp;
char fileName[] = "./hello.cl";
char *source_str;
size_t source_size;
int row = widthA;
int col = heightC;
/* Load the source code containing the kernel*/
fp = fopen(fileName, "r");
if (!fp) {
  fprintf(stderr, "Failed to load kernel.\n");
  exit(1);
}
source_str = (char*)malloc(MAX_SOURCE_SIZE);
source_size = fread( source_str, 1, MAX_SOURCE_SIZE, fp);
fclose( fp );

/* Get Platform and Device Info */
ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms)
   ;
ret = clGetDeviceIDs( platform_id, CL_DEVICE_TYPE_GPU, 1, &
   device_id, &ret_num_devices);

/* Create OpenCL context */
```

```
context = clCreateContext( NULL, 1, &device_id, NULL, NULL,
    &ret);

/* Create Command Queue */
command_queue = clCreateCommandQueue(context, device_id, 0,
    &ret);

/* Create Memory Buffer */
memobjA = clCreateBuffer(context, CL_MEM_READ_WRITE, widthA
    * heightA * sizeof(float), NULL, &ret);
memobjB = clCreateBuffer(context, CL_MEM_READ_WRITE, widthB
    * heightB * sizeof(float), NULL, &ret);
memobjC = clCreateBuffer(context, CL_MEM_READ_WRITE, widthC
    * heightC * sizeof(float), NULL, &ret);
rowA = clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(
   int), NULL, &ret);
colC = clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(
   int), NULL, &ret);

clock_t b2=clock();
// Copy the lists A and B to their respective memory
   buffers
  ret = clEnqueueWriteBuffer(command_queue,memobjA, CL_TRUE
     , 0,
          widthA * heightA * sizeof(int), A, 0, NULL, NULL);
  ret = clEnqueueWriteBuffer(command_queue, memobjB,
     CL_TRUE, 0,
           widthB * heightB * sizeof(int), B, 0, NULL, NULL)
              ;
```

```
        ret = clEnqueueWriteBuffer(command_queue, rowA,
            CL_TRUE, 0, sizeof(int), &row, 0, NULL, NULL);
        ret = clEnqueueWriteBuffer(command_queue, colC,
            CL_TRUE, 0, sizeof(int), &col, 0, NULL, NULL);

/* Create Kernel Program from the source */
program = clCreateProgramWithSource(context, 1, (const char
    **)&source_str,(const size_t *)&source_size, &ret);

/* Build Kernel Program */
ret = clBuildProgram(program, 1, &device_id, NULL, NULL,
    NULL);

/* Create OpenCL Kernel */
kernel = clCreateKernel(program, "matrixMultiplication", &
    ret);

/* Set OpenCL Kernel Arguments */
ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&
    memobjA);
ret = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&
    memobjB);
ret = clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *)&
    memobjC);
//ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&
    memobjA);
ret = clSetKernelArg(kernel, 3, sizeof(int), (void *)&row);
ret = clSetKernelArg(kernel, 4, sizeof(int), (void *)&col);
/* Execute OpenCL Kernel */
```

```
  //ret = clEnqueueTask(command_queue, kernel, 0, NULL,NULL);
  size_t globalThreads[2] = {widthA, heightB};
  size_t localThreads[2] = {8,8};

  clEnqueueNDRangeKernel(command_queue, kernel, 2, NULL,
     globalThreads, localThreads,NULL, 0, NULL);
  /* Copy results from the memory buffer */
  ret = clEnqueueReadBuffer(command_queue, memobjC, CL_TRUE,
     0,                                    widthA * heightC *
     sizeof(float),Res, 0, NULL, NULL);

  clock_t e2=clock();
double t2=((e2-b2)*1000)/CLOCKS_PER_SEC;;
printf("Time elapsed for GPU:%f\n",t2);
  fp1 = fopen("matGPURes.csv", "w");
  if (!fp1) {
    fprintf(stderr, "Failed to open matAdata.\n");
    exit(1);
  }

  printf("\nResult\n");
        for(int i = 0;i < widthA; i++)
        {
                for(int j=0;j < heightC; j++)
                {

                        fprintf(fp1, "%f,",*(Res+i*heightC+j)
                           );
```

```
                }
                fprintf(fp1, "\n");
        }
        fclose(fp1);

ret = clFlush(command_queue);
ret = clFinish(command_queue);
ret = clReleaseKernel(kernel);
ret = clReleaseProgram(program);
ret = clReleaseMemObject(memobjA);
ret = clReleaseMemObject(memobjB);
ret = clReleaseMemObject(memobjC);
ret = clReleaseCommandQueue(command_queue);
ret = clReleaseContext(context);

free(source_str);
//system("pause");

float sum=0.0;
clock_t b3=clock();
for(int i = 0;i < widthA; i++)
        {
                for(int j = 0; j < heightC; j++)
                {
                        sum = 0;
                        for(int k = 0; k < widthB; k++)
                        {
                                sum += A[i*col+k] * B[k*row+j
                                    ];
```

```
                        }
                    D[i*heightC+j] = sum;
                    }


            }
  clock_t e3=clock();
double t3=((e3-b3)*1000)/CLOCKS_PER_SEC;;
printf("Time elapsed for CPU:%f\n",t3);
    fp1 = fopen("matNormalMultiplicationRes.csv", "w");
  if (!fp1) {
    fprintf(stderr, "Failed to open matAdata.\n");
    exit(1);
  }

  printf("\nResult\n");
        for(int i = 0;i < widthA; i++)
        {
                for(int j=0;j < heightC; j++)
                {
                        fprintf(fp1, "%f,",*(D+i*heightC+j));


                }
                fprintf(fp1, "\n");
        }
   //system("pause");
  return 0;
}
```

3.6.3 Comparison of Speedup and Efficiency

In order to calculate speedup, we need to calculate following three time. In OpenCL terminology, host refers to the hardware which carries out sequential instructions and device refers to hardware which carries out parallel computations. If GPU is used as parallel computing hardware than it is known as device. Similarly if CPU is used as parallel computing hardware than it is known as both host and device. CPU-S stands for sequential time taken by the CPU for computation and communication time is time taken to transfer data from host variable to device buffer.It includes both transfer time i.e transfer from host to device and transfer from device to host. Execution time is a time taken by device for parallel computations.**Both execution time and communication time are affected by the temperature and latency of the device.**

In order to record time. time function in time.h header file is used. Code given below is example showing how time is recorded.

```
clock_t             c1=clock();
for(i=1;u<=1000;i++)
{     c[i]=a[i] + b[i]; }
clock_t             c2=clock();
double  c3=((c2-c1)*1000)/CLOCKS_PER_SEC;;
```

c3 gives time required for execution code taken as example.

```
clock_t       t4=clock();
clEnqueueWriteBuffer(command_queue, inputA, CL_TRUE, 0,
   sizeof(float) * DATA_SIZE, inputDataA, 0, NULL, NULL);
clock_t  t5=clock();
double  t6=((t5-t4)*1000)/CLOCKS_PER_SEC;;
```

t6 gives time required for loading data of buffer inputA into variable inputDataA.

Now speedup factor is a ratio of total sequential computation time to total parallel computation time.Prallel computation time is summation of execution time and communication time. Mathematically speedup factor can be defined as follows.

$$Speedupfactor = \frac{T_{Sequential}}{T_{Parallel}}$$

Table 3.1: Speedup For Paralle Square Matrix Multiplication Results

SIZE OF SQUARE MATRIX	GPU TIME(ms)	CPU TIME(ms)	Speedup
8	171	0	-
72	140	0	-
136	172	31	0.180
200	171	62	0.363
264	203	156	0.768
328	187	280	1.497
392	202	484	2.396
456	249	795	3.193
520	312	1295	4.151
584	345	2059	5.968
648	447	2933	6.562
712	546	4150	7.601
776	668	6209	9.295
840	781	7940	10.166
904	921	9969	10.824

Various sizes of matrices are considered for implementation of matrix multiplication. Device used for this matrix multiplication is ATI Mobility Radeon HD 4500/5100 Series. Table-3.1 shows the time required for parallel computations. GPU time, sequential computation CPU time and speedup. Parallel and sequential time required for multiplication of matrices of N × N order on CPU and GPU respectively are shown in Fig.3.9 and 3.10

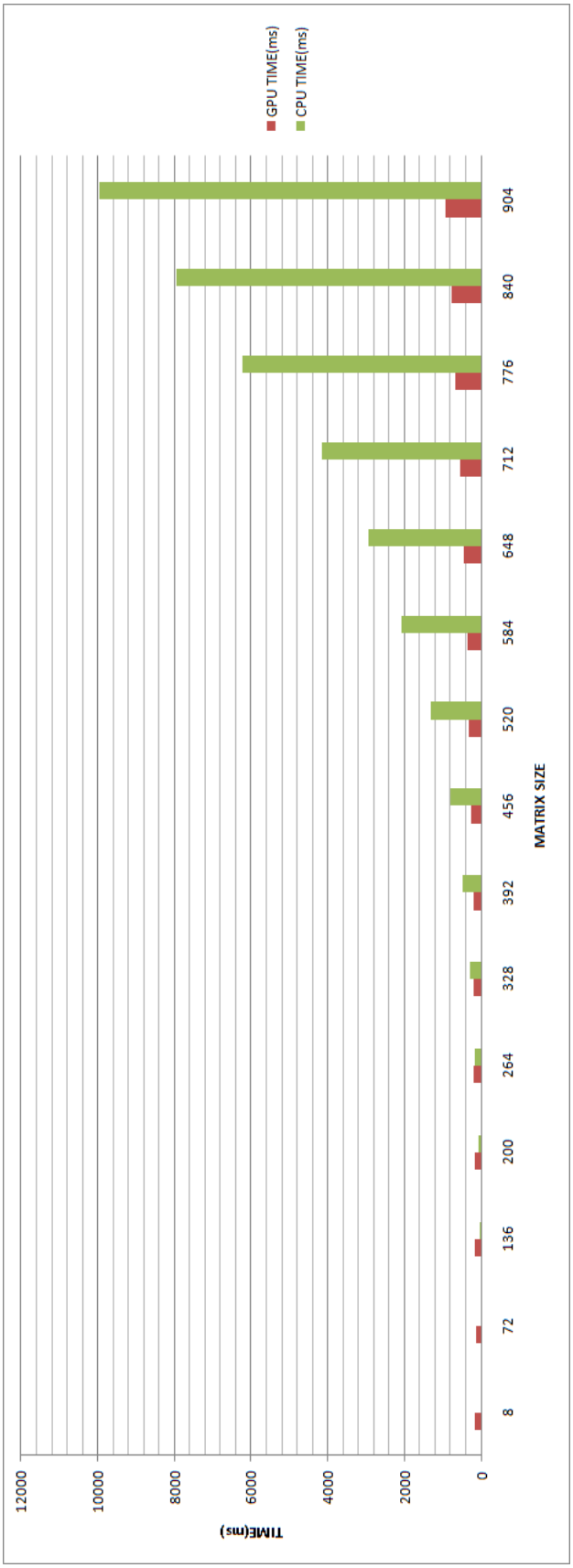

Figure 3.9: GPU And CPU Implementation of Square Matrix Multiplication

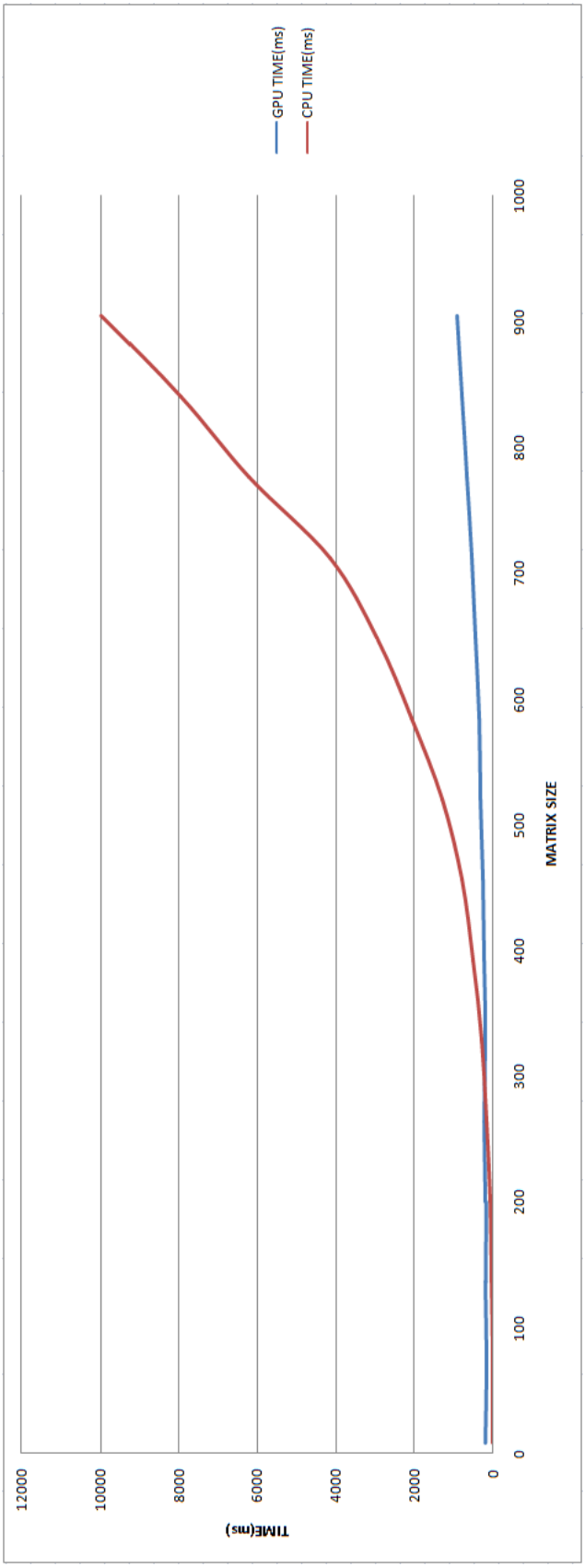

Figure 3.10: Comparison Of Computation Time For GPU And CPU Implementation of Square Matrix Multiplication

Fig.3.9 and 3.10 it is observed that for matrices of sizes upto 328×328, there is not much speed up. But for larger size of matrices there is significant speedup. From the results it is clear that parallel processing reduces the overll computing time.Parallel processing is much faster than sequential processing in case of square matrix multiplication problem considered here. It can be concluded that use of GPU for general Purpose Computing is useful in field of Structural Analysis.

3.7 Summary

In this chapter, introduction of OpenCL language along with its memory model and modules is explained in brief. It also gives an idea about OpenCL program kernel and its functioning. Multiplication of square matrices to illustrate application of OpenCL on GPU is presented in this chapter.

Chapter 4

Gaussian Elimination

4.1 General

Gaussian Elimination is widely used numerical method for the solution of simultaneous linear algebraic equations in which the unknowns are eliminated by combining the equations. The main aim of this method is to reduce a set of n equations in n unknowns to an equivalent triangular set (an equivalent set is a set having identical solution values) which is then easily solved by "back substitution". This method, hence, consists of two steps:

1. Triangularization
2. Back Substitution.

It is not practical to solve the linear algebraic equations by Cramer's Rule for n > 3. This method is quite uneconomical as compared to elimination methods and also difficult to automate. It does establish that there is a unique solution for all equations provided $|A| \neq 0$. A square matrix whose determinant is zero is known as a singular matrix. The solution of equations which have a singular or near singular coefficient matrix will require special consideration. Usually, computational time is proportional

to the number of mathematical operations i.e multiplications or additions. The number of multiplication in Cramer's Rule is $(n-1)(n+I)!$

Thus, the solution of ten simultaneous equations by determinants would require 3592512000 multiplications. With the multiplications performed on the computer at the rate of 2600/sec, atleast 38 hours is required to solve 10 equations by Cramer's Rule. To obtain a solution for 26 equations $3 \times (10)^{18}$ years would be required to obtain a solution. Yet in problems of engineering we use this method for solving equations less than three.Consider the set of equations below

$$E_1^0 = a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n = b_1 \tag{4.1}$$

$$E_2^0 = a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n = b_2 \tag{4.2}$$

... ...

$$E_i^0 = a_{i1}x_1 + a_{i2}x_2 + ... + a_{ij}x_j + ... + a_{in}x_n = b_i \tag{4.3}$$

... ...

$$E_n^0 = a_{n1}x_1 + a_{n2}x_2 + ... + a_{nn}x_n = b_n \tag{4.4}$$

Triangularization process

The first equation(4.1) E_1^0 of system is divided by the coefficient of x_1 in that equation(4.1) to obtain

$$E_1^{'} = \frac{E_1^0}{a_{11}} = x_1 + \frac{a_{12}}{a_{11}}x_2 + \frac{a_{13}}{a_{11}}x_3 + ... + \frac{a_{1n}}{a_{11}}x_n = \frac{b_1}{a_{11}} \tag{4.5}$$

Equation(4.5) is next multiplied by the co-efficient of x_1 in Equation(4.2) E_2^0 and resulting equation is subtracted from E_2^0 thus eliminating x_1 from E_2^0 as follows.

$$E_2^{'} = E_2^0 - a_{21}E_1^1 = (a_{22} - \frac{a_{21}a_{12}}{a_{11}})x_2 + (a_{23} - \frac{a_{21}a_{13}}{a_{11}})x_3 + (a_{2n} - \frac{a_{21}a_{1n}}{a_{11}})x_n = b_2 - \frac{a_{21}b_1}{a_{11}} \tag{4.6}$$

Similarly

$$E_i^{'} = E_i^0 - a_{i1}E_1^{'} \tag{4.7}$$

And so on

$$E_n^{'} = E_n^0 - a_{n1}E_1^{'} \tag{4.8}$$

The equation used to eliminate the unknowns in the equations which follow it is called the $'PivotEquation'$. In the pivot equation, the coe-efficient of unknown which is to be eliminated from subsequent equations is known as the pivot co-efficient (a_{11} in the preceding steps).Followig the above steps considering the second equation(4.2) as pivot equation and repeating same steps to eliminate x_2 from all equations.This entire procedure is repeated (n-1) times to get following form.

$$\begin{bmatrix} 1 & a_{12}^{'} & a_{13}^{'} & ... & a_{1n}^{'} \\ 0 & 1 & a_{23}^{'} & & a_{2n}^{'} \\ 0 & ... & 1 & ... & ... \\ ... & ... & ... & ... & ... \\ & & & & \\ 0 & 0 & 0 & ... & a_{nn}^{n-1} \end{bmatrix} X \begin{bmatrix} x_1 \\ x_2 \\ ... \\ ... \\ x_n \end{bmatrix} = \begin{bmatrix} b_1^{'} \\ b_2^{'} \\ ... \\ ... \\ b_n^{n-1} \end{bmatrix}$$

Back Substitution

After triangular set of equations has been obtained, that last equation in this equivalent set yields the value of x_n directly as

$$x_n = \frac{b_n^{n-1}}{a_{nn}^{n-1}} \tag{4.9}$$

This value is then back substituted in the next-to-last equations of the triangular set to obtain a value of x_{n-1} as

$$x_{n-1} = b_{n-1}^{n-1} - a_{n(n-1)}^{n-1} x_n andsoon. \tag{4.10}$$

Reviewing the procedure outlined in Triangularization, it can be seen that the elements of the reduced matrix-$A^{'}$ can be written directly from the original matrix-A, using the formula as

$$a_{ij}^{'} = a_{ij} - \frac{a_{ik}(a_{kj})}{a_{kk}} \tag{4.11}$$

where $k \leq j \leq m$ and $k+1 \leq i \leq n$

After obtaining the augmented matrix of the equivalent triangular set of equations, the x_i values are obtained by back substitution.
In order to program this in computer, we can write

$$a_{ij}^{k} = a_{ij}^{k-1} - \frac{a_{kj}^{k-1}(a_{ik}^{k-1})}{a_{kk}^{k-1}} \tag{4.12}$$

$$b_{ik} = \frac{b_i^{k-1} - a_{ik}^{k-1} - b_k^{k-1}}{a_{kk}^{k-1}} \tag{4.13}$$

where $k+1 \leq j \leq n$

$k+1 \leq i \leq n$

i=row number of matrix

j=column number of matrix

k=number identifying pivot row

n=number of rows in matrix

m=number of columns in matrix

4.2 Algorithm of Gaussian Elimination

1 Assume N equations and Elimination has been completed upto $(N-1)^{th}$ variable

2 The Elimination of then n^{th} variable

- $a_{nj} = a_{nj}/a_{nn}$

  $b_n = b_n/a_{nn}$

- For i=n+1, $a_{ij} = a_{ij} - a_{in}a_{nj}$ for j=n+1

  $b_i = b_i - a_{in}b_n$

3 Now Set n=n+1 and go back to 2

4 Back Substitution

Solve $X_N = b_N^{N-1}/a_{NN}$

$X_{N-1} = b_{N-1}^{N-2} - [a_{N-1}^{N-2}X_N]$

4.3 Sequential Implementation

The following code represents the sequential implementation of Gauss Elimination. It consists both Triangularization and Back Substitutions.

```
for(u=1;u<=(1*nj);u++)
{
    temp=kj[u][u];
        for(j=1;j<=(1*nj);j++)
            kj[u][j]=kj[u][j]/temp;
    ac[u]=ac[u]/temp;
        for(i=1;i<=(1*nj);i++)
        {
            if(i==u)continue;
```

```
        temp1=kj[i][u];
            for(j=1;j<=(1*nj);j++)
                        kj[i][j]=kj[i][j]-kj[u][j]*temp1;
        ac[i]=ac[i]-ac[u]*temp1;
         }
}
```

where [kj]=Matrix to be inversed and [ac]=Displacement Vector

After running this code, one gets directly the unknown vector {x} in form of vector {ac}.

4.4 Parallel Implementation

Parallel implementation requires host program and kernel function.Entire host program along with kernel function is given in Appendix-A. The following code represents kernel functions that executes on parallel hardware i.e CPU/GPU. Only Triangularization is done in parallel and Back Substitution is done sequentially.

1 Triangularization is divided into two kernel functions.

```
\__kernel void add(__global float *inputM, __global float
    *inputA, __global float *inputB,const int size,const
   int t)
{
    int globalId = get_global_id(0);
    if (globalId < size-1-t)
    {
        *(inputM + size * (globalId + t + 1)+t) = *(
           inputA + size * (globalId + t + 1) + t) / *(
           inputA + size * t + t);
```

```
    }
}
__kernel void add2(__global float *inputM, __global float
    *inputA, __global float *inputB,const int size,const
   int t)
{
    int globalIdx = get_global_id(0);
    int globalIdy = get_global_id(1);
    if (globalIdx < size-1-t && globalIdy < size-t)
    {
        inputA[size*(globalIdx+1+t)+(globalIdy+t)] -=
           inputM[size*(globalIdx+1+t)+t] * inputA[size*t
           +(globalIdy+t)];
        if(globalIdy == 0)
        {
            inputB[globalIdx+1+t] -= inputM[size*(
               globalIdx+1+t)+(globalIdy+t)]* inputB[t];
        }
    }
}
```

2 Back Substitution

```
for(i=0;i<size;i++)
    {
        finalVec[size-i-1]=inputDataB[size-i-1];
        for(j=0;j<i;j++)
        {
            finalVec[size-i-1]-=*(inputDataA+size*(size-i
               -1)+(size-j-1))*finalVec[size-j-1];
```

```
                    }

    finalVec[size-i-1]=finalVec[size-i-1]/*(inputDataA+
        size*(size-i-1)+(size-i-1));
    }
```

Equations in form of [A]{x}={B} using finite element analysis of axial bar using 3-node bar element where A=Square Stiffness Matrix, B=Load Vector and x=Displacement vector. For comparing computational efficiency of parallel code, speedup factor which is ratio of sequential execution time to parallel execution time is calculated for different number of linear equations ranging 101 to 10001. Different types of hardwares used are listed in Table-4.1 given below.

Table 4.1: Hardware used and their configurations

Sr. No.	Notation	Hardware Specification	Memory (RAM)
1	i3	Intel Core i3-3210 Processor (3M Cache,3.20 GHz)	4 GB
2	i5	Intel Core i5-3450 Processor (6M Cache,3.50GHz)	4 GB
3	i7	Intel Core i7-3450 Processor (6M Cache,3.50GHz)	32 GB
4	i7 - laptop	Intel Core i7-2630QM Processor(2.0GHz)	8 GB
5	i-7 laptop GPU	NVIDIA GeForce GT 525M	2 GB

Results of sequential and parallel implementation are measured in millisecond i.e ms. CPU-S stands for sequential time and CPU-P stands for parallel time. CPU-P is summation of execution time and communication time. Communication time depends on bandwidth of computer system considered. Bandwidth refers to amount of data transferred per unit time. Different computer systems have different bandwidth of data transfer.

Table-4.2 shows results of speedup for set of linear equation systems tested on Intel® Core™i3-3210 Processor(3M Cache,3.20 GHz). It is seen that for small set of equations i.e. 101 and 201 number of equations sequential computing time is equal to parallel computing time. For solving 301 and 401 number of equations, sequential computing time is more compared to parallel computing time. For solving number of equations from 101 to 601 there is no communication time which means that data transfer is done at maximum speed. For solving equations more than 701, sequential computing time is increasing considerably but parallel computing time is lesser. Maximum speed up of 838 for solving 10001 number of equations is obtained. Fig.4.1 is graphical representation of the results.

Table-4.3 shows results of speedup for set of linear equation systems tested on Intel® Core™i5-3450 Processor(6M Cache,3.50GHz). It is seen that for small set of equations i.e. 101 and 301number of equations sequential computing time is equal to parallel computing time. For solving number of equations more than 301,sequential computing time is more compared to parallel computing time. For solving number of equations from 101 to 301 there is no communication time which means that data transfer is done at optimum speed. For solving number of equations from 101 to 1301, there is no execution time for parallel computing which means that all computations are done in single fraction of second. Maximum speed up of 1100 for solving 10001 number of equations is obtained. Fig.4.2 is graphical representation of the results.

Table-4.4 shows results of speedup for set of linear equation systems tested on Intel® Core™i7-3450 Processor(6M Cache,3.50GHz). This hardware is having 32GB RAM in divided into 4 different slots each of 8GB.Due to multiple slots, it is seen that for small set of equations i.e. 301 number of equations there is communication time. Maximum speed up of 316 for solving 5001 number of equations is obtained. Fig.4.3 is graphical representation of the results.

Table 4.2: Performance Comparison using Intel® Core™i3-3210 Processor(3M Cache,3.20 GHz)

NO OF EQUATIONS	CPU-S (ms) i3	EXECUTION TIME(ms)	COMMUNICATION TIME(ms)	TOTAL TIME CPU-P(ms)	SPEEDUP
101	-	-	-	-	-
201	-	-	-	-	-
301	16	-	-	-	-
401	31	-	-	-	-
501	78	15	-	15	5.200
601	140	16	-	16	8.750
701	219	15	16	31	7.065
801	312	16	16	32	9.750
901	502	16	16	32	15.688
1001	718	15	16	31	23.161
1101	1030	15	15	30	34.333
1201	1357	16	16	32	42.406
1301	1841	16	15	31	59.387
1401	2359	18	15	33	71.485
1501	3025	16	31	47	64.362
1601	3915	16	31	47	83.298
1701	4540	16	31	47	96.596
1801	5595	31	31	62	90.242
1901	6286	47	16	63	99.778
2001	7292	31	32	63	115.746
2501	14414	31	31	62	232.484
3001	24679	47	46	93	265.366
3501	39000	62	62	124	314.516
4001	57969	78	78	156	371.596
4501	80652	79	108	187	431.294
5001	109975	93	124	217	506.797
10001	863695	250	780	1030	838.539

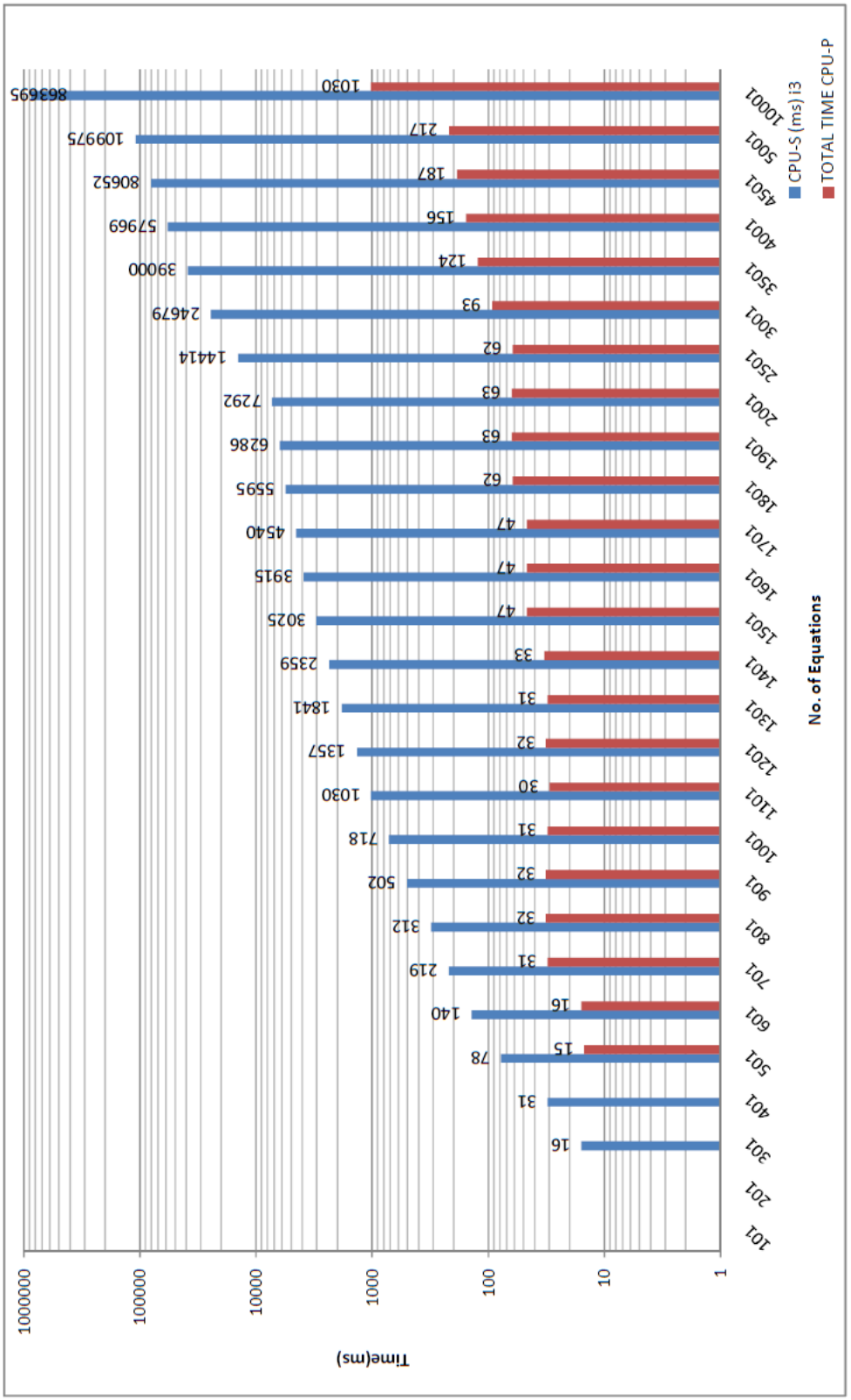

Figure 4.1: Performance Comparison using Intel® Core™i3-3210 Processor(3M Cache,3.20 GHz)

Table 4.3: Performance Comparison using Intel® Core™i5-3450 Processor(6M Cache,3.50GHz)

NO OF EQUATIONS	CPU-S (ms) i5	EXECUTION TIME (ms)	COMMUNICATION TIME(ms)	TOTAL TIME CPU-P(ms)	SPEEDUP
101	-	-	-	-	-
201	16	-	-	-	-
301	15	-	-	-	-
401	31	-	16	16	0.516
501	78	-	16	16	4.875
601	125	-	16	16	7.813
701	203	-	15	15	13.533
801	312	-	16	16	19.500
901	452	-	15	15	30.133
1001	624	-	16	16	39.000
1101	905	-	15	15	60.333
1201	1263	-	16	16	78.938
1301	1638	-	30	30	54.600
1401	2137	-	16	16	133.563
1501	2793	15	15	30	93.100
1601	3432	15	16	31	110.710
1701	4290	16	31	47	91.277
1801	5320	15	16	31	171.613
1901	6364	16	15	31	205.290
2001	7597	16	16	32	237.406
2501	16239	32	46	78	208.192
3001	28610	16	46	62	461.452
3501	44662	32	47	79	565.342
4001	65988	63	78	141	468.000
4501	91681	62	109	171	536.146
5001	128762	78	125	203	634.296
10001	1081534	172	811	983	1100.238

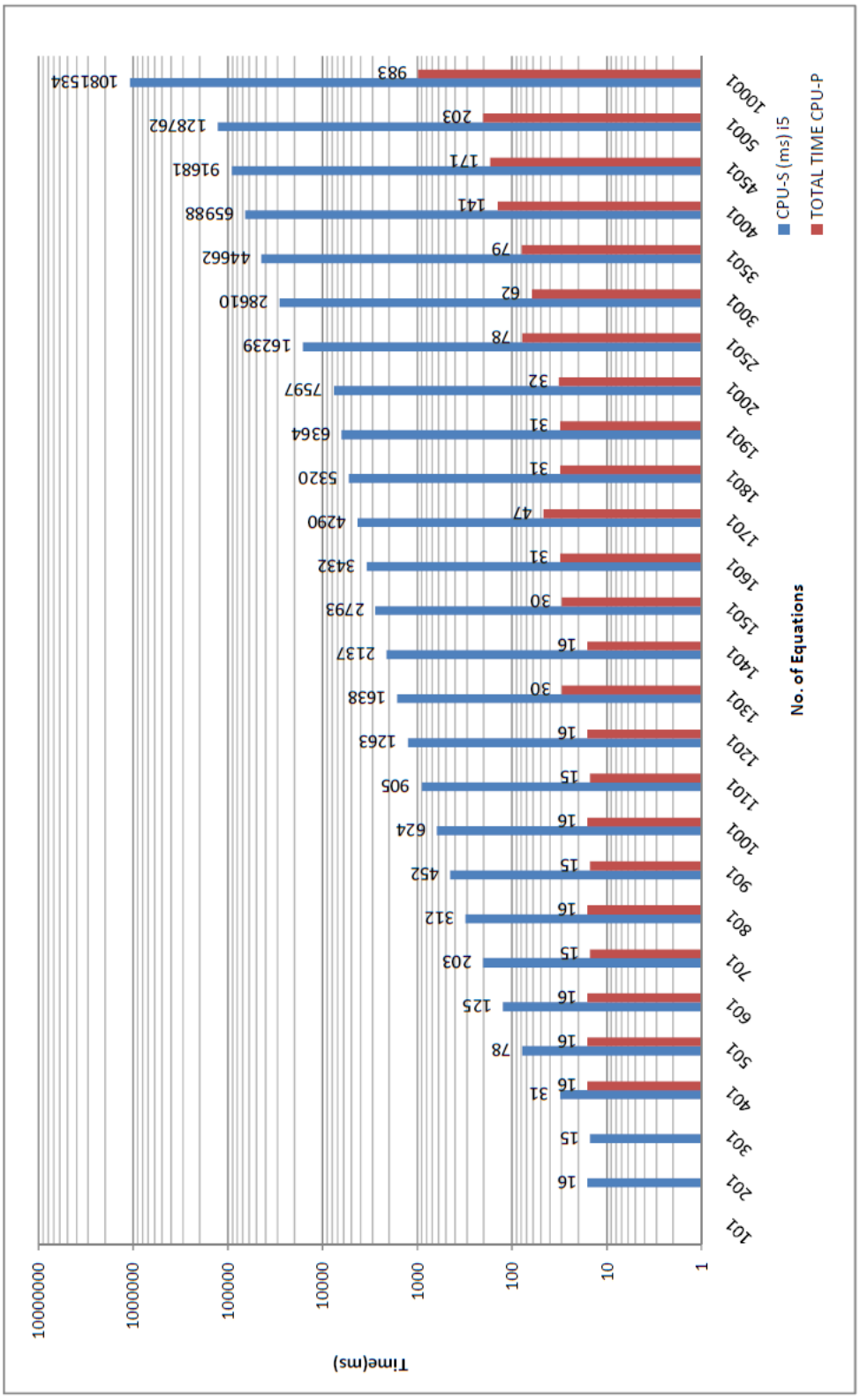

Figure 4.2: Performance Comparison using Intel® Core™i5-3450 Processor(6M Cache,3.50GHz)

Table 4.4: Performance Comparison using Intel® Core™i7-3450 Processor(6M Cache,3.50GHz)

NO OF EQUATIONS	CPU-S (ms) i7	EXECUTION TIME(ms)	COMMUNICATION TIME(ms)	TOTAL TIME CPU-P(ms)	SPEEDUP
101	-	15	-	-	-
201	31	16	-	16	1.938
301	93	16	15	31	3.000
401	203	-	31	31	6.548
501	405	32	15	47	8.617
601	687	47	15	62	11.081
701	1092	46	16	62	17.613
801	1700	63	15	78	21.795
901	2324	62	32	94	24.723
1001	3370	78	47	125	26.960
1101	4337	93	48	141	30.759
1201	5709	93	48	141	40.489
1301	7223	109	62	171	42.240
1401	8829	109	63	172	51.331
1501	11216	109	94	203	55.251
1601	13354	172	77	249	53.631
1701	16068	157	93	250	64.272
1801	19297	156	109	265	72.819
1901	21965	156	125	281	78.167
2001	25678	171	109	280	91.707
2501	50045	219	187	406	123.264
3001	86455	234	297	531	162.815
3501	136593	297	369	666	205.095
4001	203080	344	483	827	245.562
4501	289396	406	624	1030	280.967
5001	394883	468	781	1249	316.159

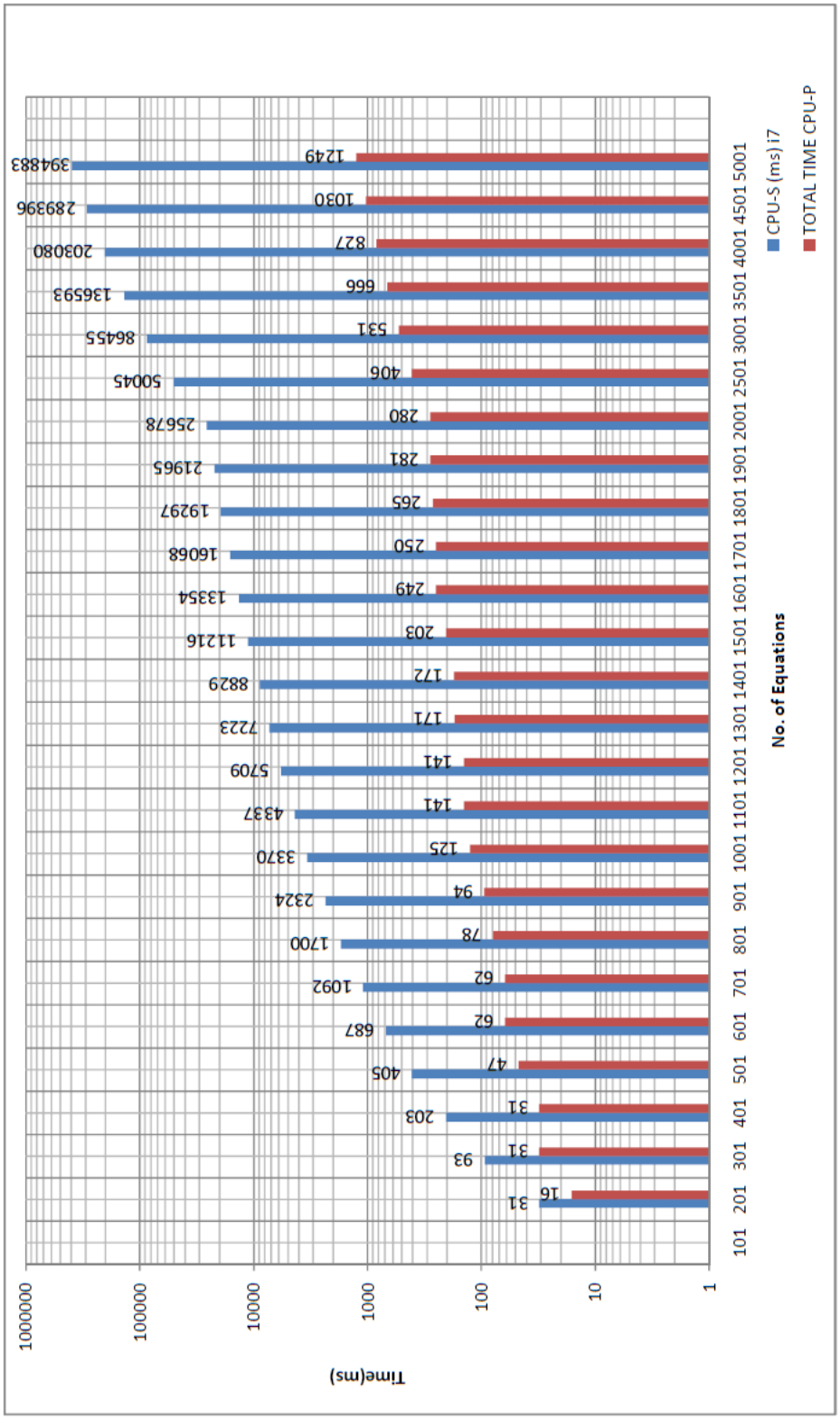

Figure 4.3: Performance Comparison using Intel® Core™i7-3450 Processor(6M Cache,3.50GHz)

Table-4.5 shows results of speedup for set of linear equation systems tested on Intel® Core™i7-2630QM Processor(2.0GHz). It is seen that for small set of equations i.e. 101 and 201 number of equations sequential computing time is equal to parallel computing time. For solving number of equations more than 301, sequential computing time is more compared to parallel computing time. For number of equations from 101 to 801, there is no communication time which means that data transfer is done at maximum speed. Since this hardware is on laptop, there is execution time right from 101 equations. For given hardware, maximum speed up of 1702 for solving 10001 number of equations is obtained. Fig.4.4 is graphical representation of the results.

Table-4.6 shows results of speedup for set of linear equation systems tested on NVIDIA GeForce GT 525M. It is seen that for small set of equations i.e. 101 and 201 number of equations sequential computing time is equal to parallel computing time. For number of equations more than 301, sequential computing time is more compared to parallel computing time. For solving number of equations from 101 to 501, there is no communication time which means that data transfer is done at optimum speed. Since this hardware is on laptop, there is execution time right from 301 number of equations onwards. For given hardware, maximum speed of 1702 for solving 10001 number of equations is obtained. Fig.4.5 is graphical representation of the results.

Table 4.5: Performance Comparison using Intel® Core™i7-2630QM Processor(2.0GHz)

NO OF EQUATIONS	CPU-S (ms) i7 Laptop	EXECUTION TIME(ms)	COMMUNICATION TIME(ms)	TOTAL TIME CPU-P(ms)	SPEEDUP
101	-	16	-	-	-
201	-	16	-	-	-
301	21	12	2	14	1.500
401	47	15	-	15	3.133
501	91	21	2	23	3.957
601	140	31	-	31	4.516
701	234	32	-	32	7.313
801	359	46	-	46	7.804
901	531	31	16	47	11.298
1001	733	47	15	62	11.823
1101	1029	47	15	62	16.597
1201	1435	47	16	63	22.778
1301	1935	47	15	62	31.210
1401	2449	62	18	80	30.613
1501	3073	46	32	78	39.397
1601	3698	78	29	107	34.561
1701	4461	62	16	78	57.192
1801	5359	73	12	85	63.047
1901	6130	78	16	94	65.213
2001	7036	94	15	109	64.550
2501	13307	94	15	109	122.083
3001	24929	110	30	140	178.064
3501	42556	140	47	187	227.572
4001	66221	172	47	219	302.379
4501	97843	250	78	328	298.302
5001	133582	281	94	375	356.219
10001	1222107	421	297	718	1702.099

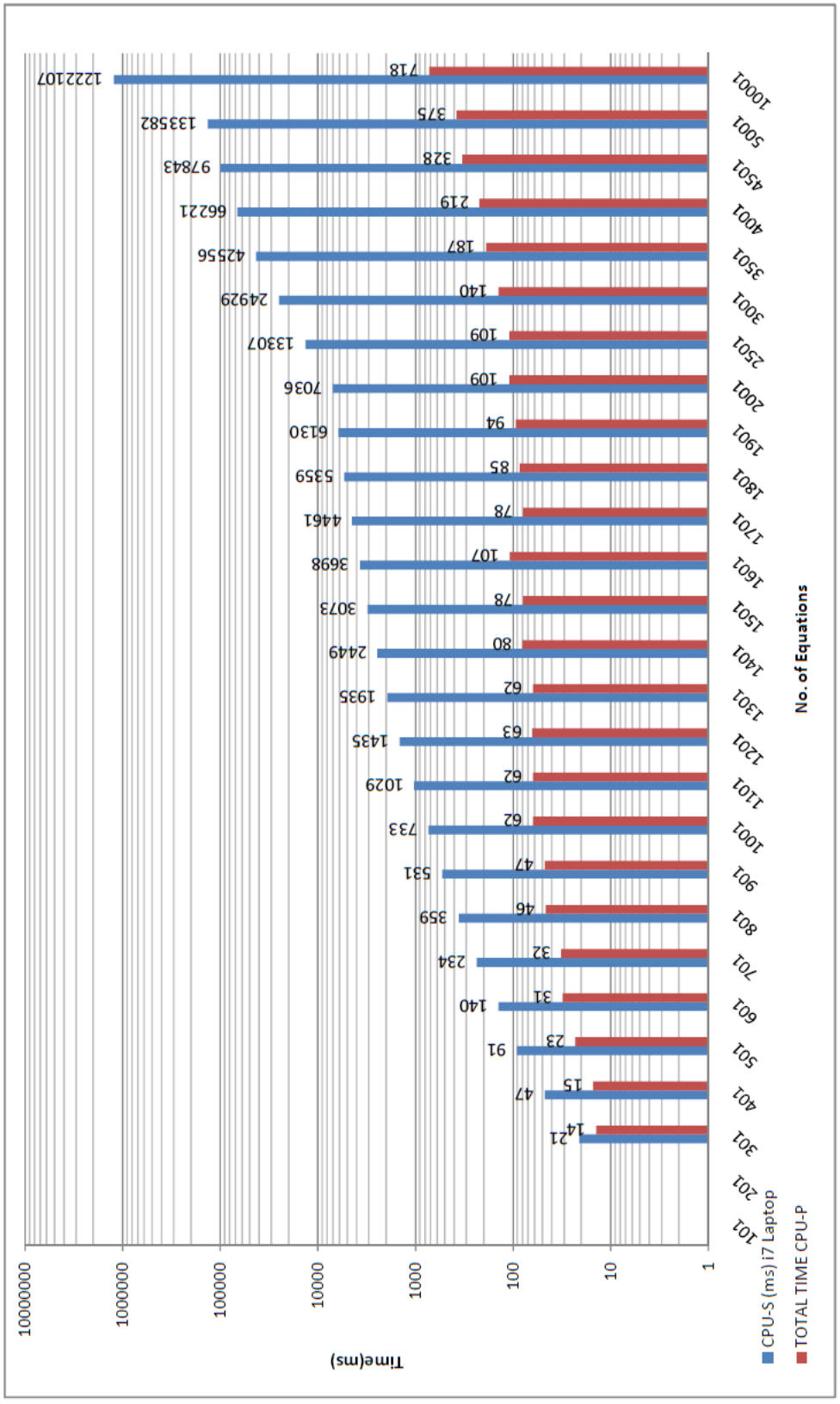

Figure 4.4: Performance Comparison using Intel® Core™i7-2630QM Processor(2.0GHz)

Table 4.6: Performance Comparison using NVIDIA GeForce GT 525M

NO OF EQUATIONS	CPU-S (ms) i7 Laptop	EXECUTION TIME(ms)	COMMUNICATION TIME(ms)	TOTAL TIME CPU-P(ms)	SPEEDUP
101	-	-	-	-	-
201	-	-	-	-	-
301	21	16	-	16	1.313
401	47	32	-	32	1.469
501	91	16	-	16	5.688
601	140	16	15	31	4.516
701	234	16	16	32	7.313
801	359	47	15	62	5.790
901	531	31	19	50	10.620
1001	733	32	15	47	15.596
1101	1029	47	18	65	15.831
1201	1435	47	16	63	22.778
1301	1935	47	19	66	29.318
1401	2449	46	21	67	36.552
1501	3073	63	15	78	39.397
1601	3698	78	15	93	39.763
1701	4461	62	16	78	57.192
1801	5359	63	15	78	68.705
1901	6130	94	15	109	56.239
2001	7036	78	16	94	74.851
2501	13307	124	32	156	85.301
3001	24929	140	31	171	145.784
3501	42556	157	46	203	209.635
4001	66221	187	63	250	264.884
4501	97843	234	63	297	329.438
5001	133582	281	63	344	388.320
10001	1222107	672	312	984	1241.979

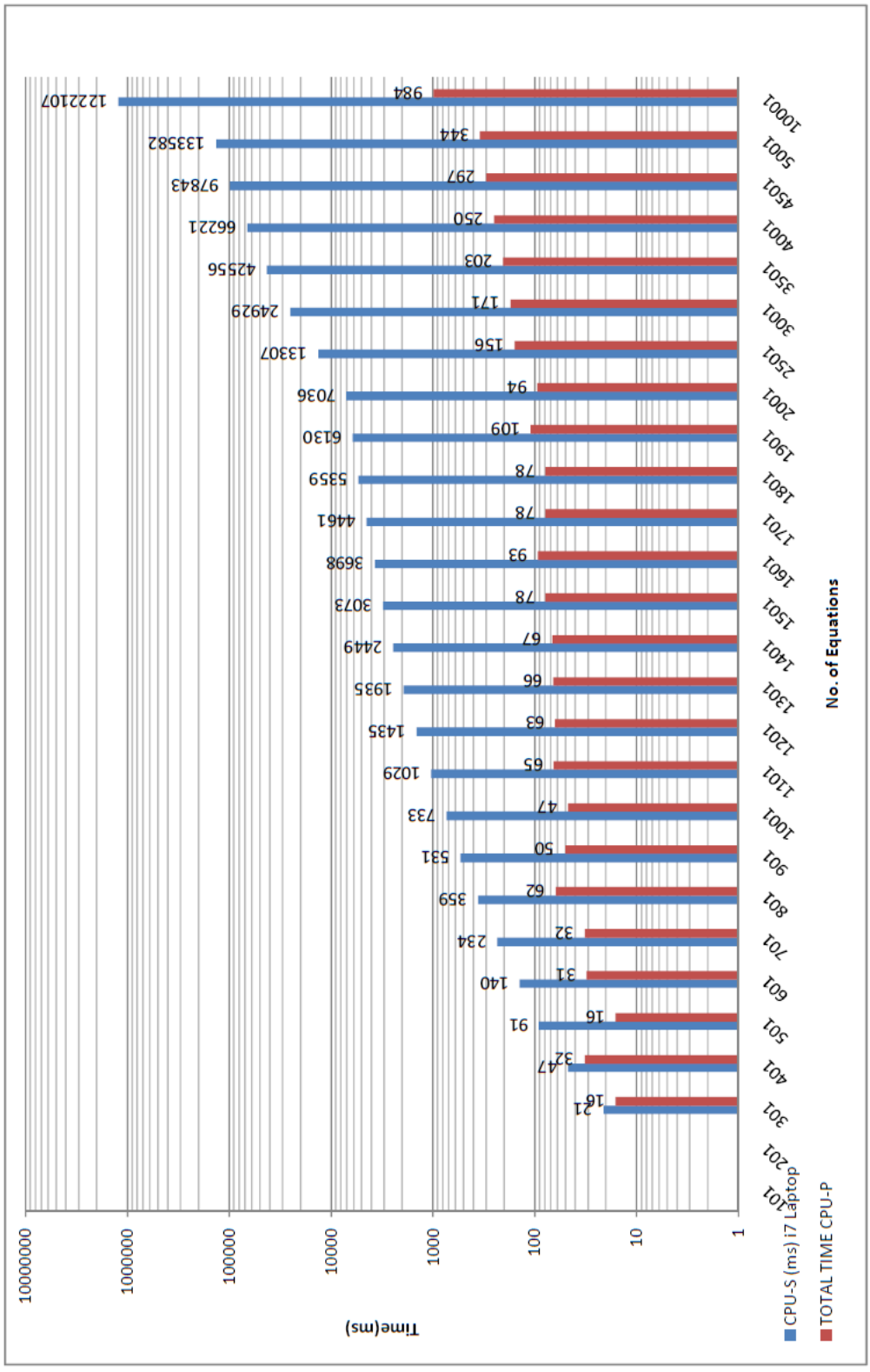

Figure 4.5: Performance Comparison using NVIDIA GeForce GT 525M

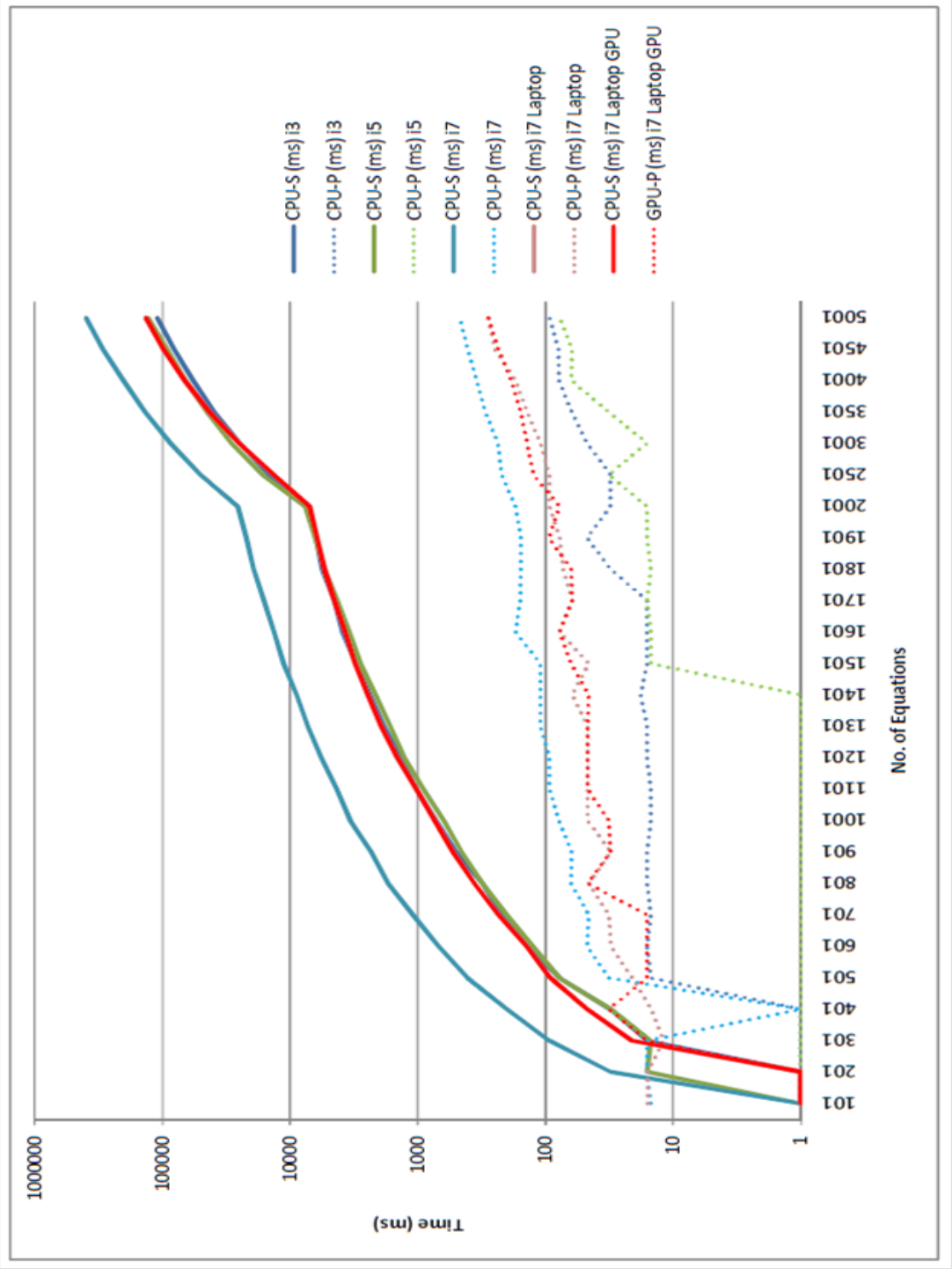

Figure 4.6: Comparison of Execution Time of Different Hardwares for Gauss Elimination

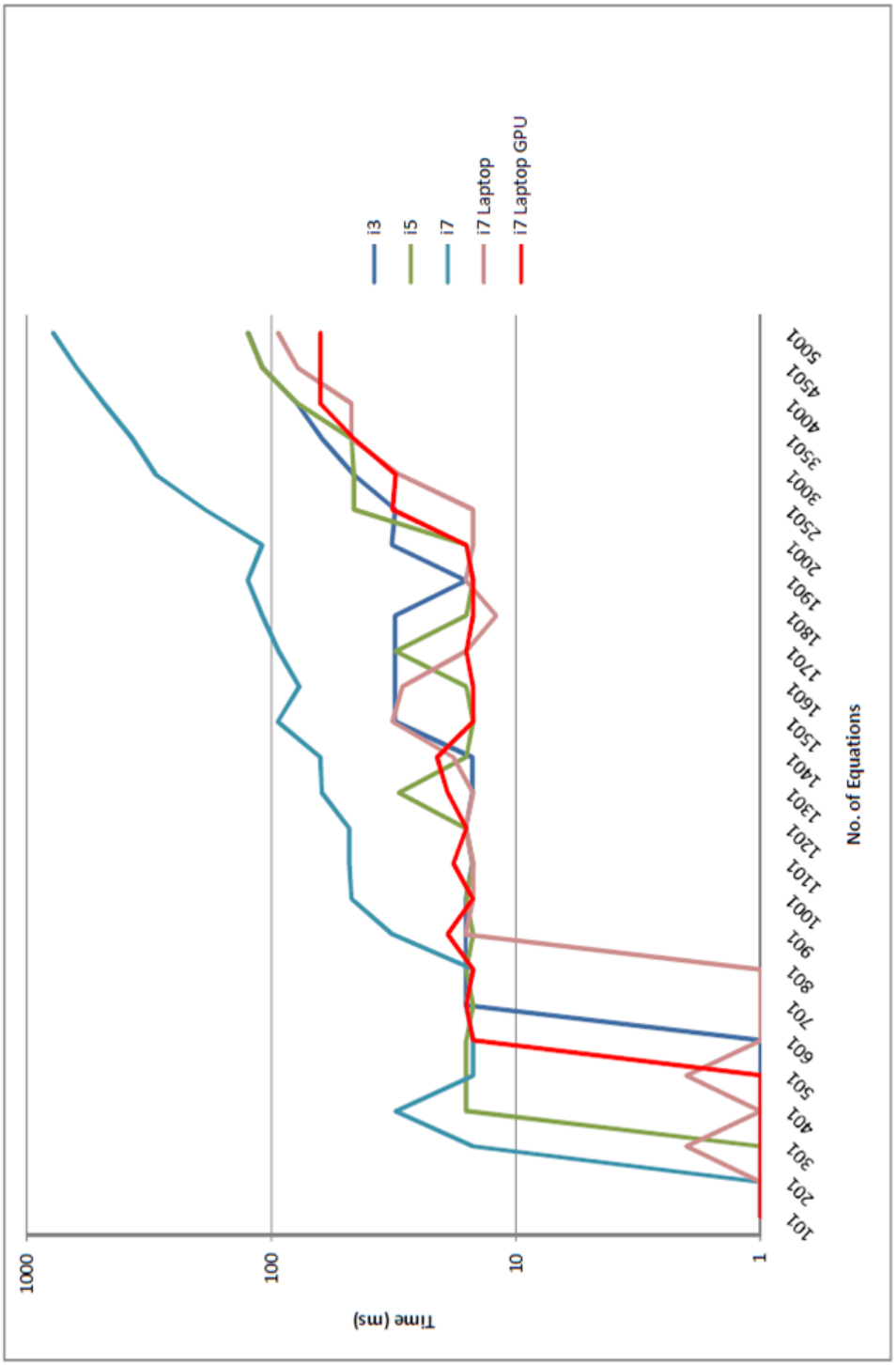

Figure 4.7: Comparison of Communication Time of Different Hardwares for Gauss Elimination

Figure 4.8: Comparison of Speedup factor of Different Hardwares for Gauss Elimination

Fig.4.6 shows comparison of execution time of both parallel and sequential code of Gauss Elimination. Solid line and dashed line represents results of sequential execution time and parallel execution time respectively. It is observed that parallel execution time lesser than sequential execution time for all the types of hardwares used in parametric study. Parallel execution time of Intel® Core™i5-3450 Processor is least followed by Intel® Core™i3 Processor. Parallel execution time of Intel® Core™i7-2630QM Processor and NVIDIA GeForce GT 525M GPU are nearly same.

Fig.4.7 shows comparison of communication time of all 5 computer systems used for parametric study. Intel® Core™i7-3450 Processor system is taking maximum communication time. This is due to multiple slots of 32 GB RAM divided into 4 slots each of 8 GB. Intel® Core™i7-2630QM Processor is taking least communication time among all 5 computer systems upto 2501 number of equations but after that NVIDIA GeForce GT 525M GPU is taking least communication time for larger number of equations.

Fig.4.8 shows comparison of speedup factor of all 5 computer systems considered. Intel® Core™i7-2630QM Processor gives maximum speedup factor of 1702.099 among all 5 computer systems followed by NVIDIA GeForce GT 525M GPU with speedup factor of 1241.979. For parallel implementation of Gauss Elimination, Intel® Core™i7-2630QM gives optimum performance among 5 computer systems.

4.5 Summary

In this chapter, Gaussian Elimination method for solving linear equations i.e. [A]{x}={B} where [A] is square matrix, with the help of an algorithm is presented. Its sequential and parallel implementations on multi-core CPUs and GPUs is presented. Based on different number of linear equation systems tested on different computer systems, results of execution time, communication time and speedup factor are presented.

Chapter 5

Half-Band Matrix Solver

5.1 General

Generally in analysis of various types of structures, stiffness matrices are square and symmetric consisting of many zero elements. So, storing stiffness matrix in square form will require large memory and large number of computations. If matrix is stored in Half-Band form, amount of memory required and number of computations will be reduced drastically.

Consider the symmetric matrix shown below. There are some zero elements.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & & & \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} & & \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} & a_{37} & \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} & a_{47} & a_{48} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} & a_{56} & a_{57} & a_{58} \\ & a_{62} & a_{63} & a_{64} & a_{65} & a_{66} & a_{67} & a_{68} \\ & & a_{73} & a_{74} & a_{75} & a_{76} & a_{77} & a_{78} \\ & & & a_{84} & a_{85} & a_{86} & a_{87} & a_{88} \end{bmatrix}$$

It is enough to store elements on the upper portion of the main diagonal as shown. As far as the first row is concerned, one has to store all the five elements. But in second row one need not to store a_{21} since $a_{21} = a_{12}$ which is already stored. Similarly, in the third row we store elements a_{33} to a_{37} and the storage scheme is shown in matrix below

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{22} & a_{23} & a_{24} & a_{25} & a_{26} \\ a_{33} & a_{34} & a_{35} & a_{36} & a_{37} \\ a_{44} & a_{45} & a_{46} & a_{47} & a_{48} \\ a_{55} & a_{56} & a_{57} & a_{58} & \\ a_{66} & a_{67} & a_{68} & & \\ a_{77} & a_{78} & & & \\ a_{88} & & & & \end{bmatrix}$$

By using this procedure insted of storing 8×8=64 elements, in band matrix only 8×5=40 elements are stored. This is the greatest advantage in saving the storage memory.To solve the banded-matrix modified Gauss Elimination solver is used. The band- matrix is stored as shown in Figure5.1 below

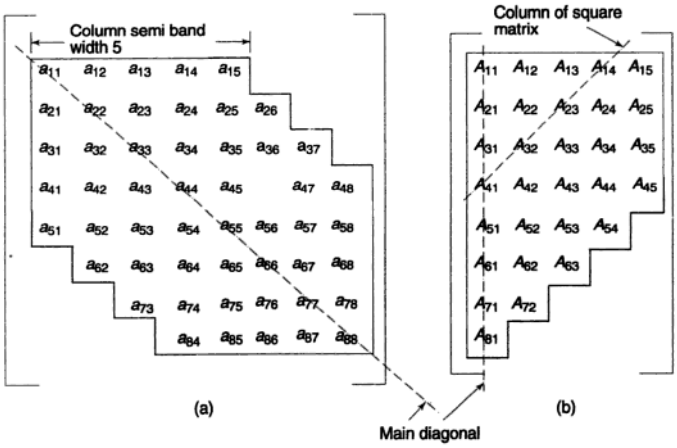

Figure 5.1: Band Matrix for sympatric square matrix (a) Square Matrix (b) Band Matrix[35]

5.2 Algorithm of Half-Band Solver

1 $c = a_{12}/a_{11}$

2 Modify second row elements as follows

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

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

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

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

3 Modify right hand side b_2 as

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

4 Now change value of changing value of a_{12} as follows

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

After following above four steps we get following matrix.

$$\begin{bmatrix} a_{11}^{'} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21}^{'} & a_{22}^{'} & a_{23}^{'} & a_{24} & a_{25} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ a_{51} & a_{52} & a_{53} & a_{54} & \\ a_{61} & a_{62} & a_{63} & & \\ a_{71} & a_{72} & & & \\ a_{81} & & & & \end{bmatrix}$$

Taking the second equation, one may follow the same steps as 1 to 4 as before. This is to be repeated for $(NE-1)$ times.For N^{th} equation and modifying the a_{NL}^{th} element $a_{IJ}^{'} = a_{IJ} - \frac{a_{NL}a_{NK}}{a_{N1}}$

$b'_I = b_I - \frac{a_{NL} b_N}{a_{N1}}$

5.3 Sequential Implementation

Based on algorithm discussed previously, sequential code for Half-Band solver is as follows.

a Factorization

```
NL=nra-nb+1;
NM=nra-1;
MR=nb;
for(N=1;N<=NM;N++)
{       BN=ac[N];
        ac[N]=BN/kj[N][1];
        if(N>NL) MR=nra-N+1;
        for(L=2;L<=MR;L++)
        {       C=kj[N][L]/kj[N][1];
                i=N+L-1;
                j=0;
                for(K=L;K<=MR;K++)
                {       j=j+1;
                        kj[i][j]=kj[i][j]-C*kj[N][K];
                }
                ac[i]=ac[i]-C*BN;
                kj[N][L]=C;
        }
}
```

b Back Substitution

```
i=nra;
ac[nra]=ac[nra]/kj[nra][1];
for(N=1;N<=NM;N++)
{
        i=i-1;
        if(N<nb)
                MR=N+1;
        for(j=2;j<=MR;j++)
        {       k=i+j-1;
                ac[i]=ac[i]- kj[i][j]*ac[k];
        }
}
```

5.4 Parallel Implementation

Parallel implementation requires host program and kernel function.Entire host program along with kernel function is given in Appendix-B. The following code represents kernel functions that executes on parallel hardware i.e CPU/GPU.

a Factorization is divided into five kernel functions.

```
__kernel void add(__global float *inputM, __global float
   *inputA, __global float *inputB,const int size,const
   int t,const int MRg)
{
    int globalId = get_global_id(0);
      if (globalId < MRg)
    {
        *(inputM + + size * t + globalId  + 1) = *(
          inputA + size * t + globalId  + 1) / *(
```

```
                inputA + size * t  );
        }
    }

    __kernel void add2(__global float *inputM, __global float
        *inputA, __global float *inputB,const int size,const
       int t,const int MRg)
    {
        int globalIdx = get_global_id(0);
            int globalIdy = get_global_id(1);
            if (globalIdx < MRg && globalIdy < (MRg-globalIdx
               ))
            {
               inputA[size*(globalIdx+1+t)+(globalIdy)] -=
                  inputM[+ size * t + globalIdx  + 1] *
                  inputA[size*t+(globalIdy)+1+globalIdx];

            }
    }

    __kernel void add3(__global float *inputM, __global float
        *inputA, __global float *inputB,const int size,const
       int t,const int MRg)
    {
        int globalId = get_global_id(0);
            if (globalId < MRg)
            {
               inputB[globalId+1+t] -= inputM[size * t +
                  globalId  + 1] * inputB[t];
```

```
        }
    }

    __kernel void add4(__global float *inputM, __global float
        *inputA, __global float *inputB,const int size,const
       int t,const int MRg)
    {
        int globalId = get_global_id(0);
            if (globalId == 1)
            {
               inputB[t*globalId] = inputB[t*globalId] /
                  inputA[size*t*globalId];
        }
    }

    __kernel void add5(__global float *inputM, __global float
        *inputA, __global float *inputB,const int size,const
       int t,const int MRg)
    {
        int globalId = get_global_id(0);
            if (globalId < MRg)
            {
               *(inputA ++ size * t + globalId  + 1) = *(
                  inputM + size * t + globalId  + 1);
        }
    }
```

b Back Substitution

```
i=nra-1;
```

```
inputDataB[nra-1]=inputDataB[nra-1]/inputDataA[(nra-1)*nb
    ];
for(Ng=0;Ng<=NMg;Ng++)
{       i=i-1;
        if(Ng<(nb-1))
                MRg=Ng+1;
        for(j=1;j<=MRg;j++)
        {
                k=i+j;
                inputDataB[i]=inputDataB[i]- inputDataA[i
                    *nb + j]*inputDataB[k];
        }
}
```

5.5 Plane Frame Analysis

In the case of plane frame, all the members lie in the same plane and are interconnected by rigid joints. The internal forces at a cross-section of a plane frame member consist of bending moment, shear force and an axial force. The significant deformations in the plane frame are only flexural and axial. Typical plane frame considered for analysis is shown in Fig.5.2.The Global Stiffness matrix is stored in Half-Band manner. Subsequently half-banded matrix is inverted using Cholesky factorization. Global stiffness matrix is rectangular matrix in which number of rows is equals to numbers of Degrees of Freedom and number of columns equals to half-bandwidth. Thus matrix size turns out to be DoF×nb where, DoF= Degrees of Freedom and nb=half-bandwidth. For solution of equations, Half-Band matrix solver is used. Speedup factor called Tsequntial/Tparallel is also calculated.

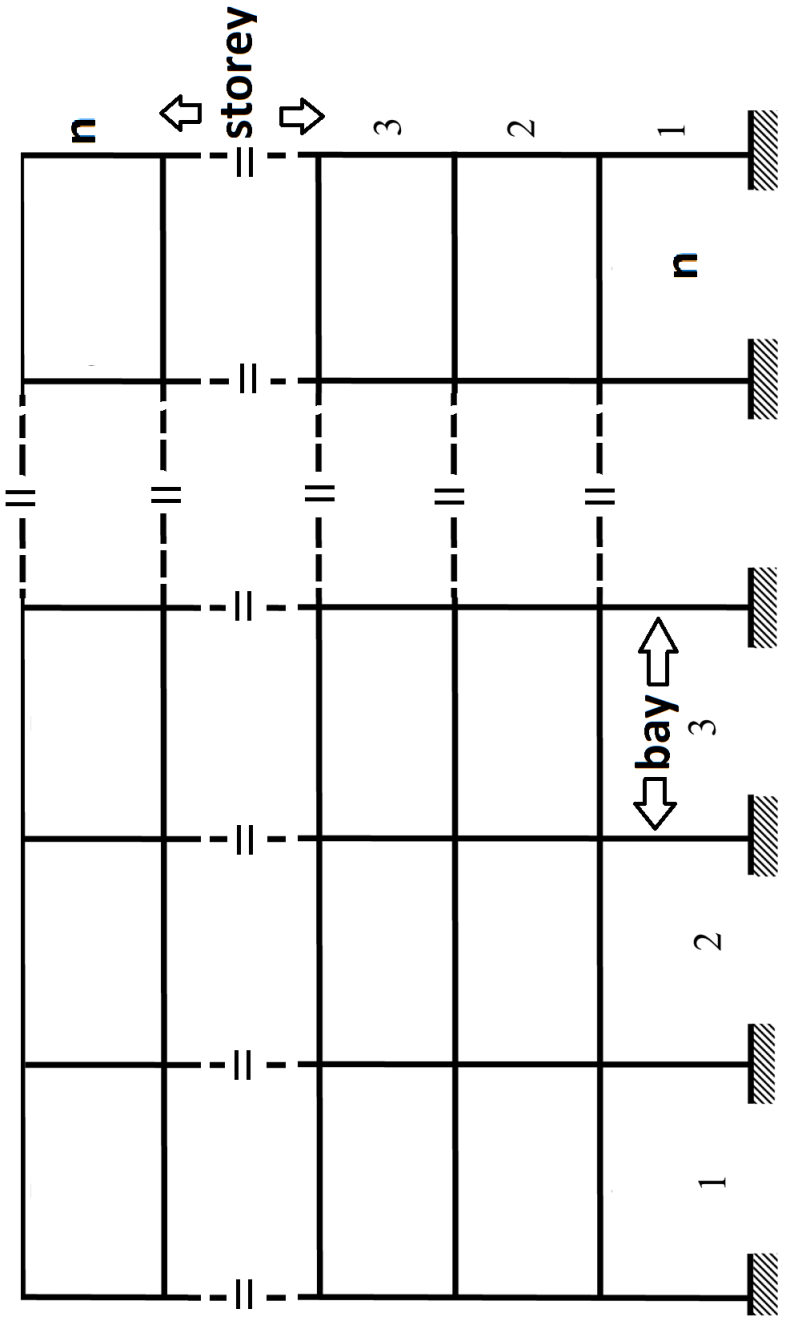

Figure 5.2: Plane Frame Schematic Diagram

Plane Frame member Stiffness Matrix for member axes is shown below:

$$\begin{bmatrix} \frac{AE}{L} & 0 & 0 & \frac{-AE}{L} & 0 & 0 \\ 0 & \frac{12EI}{L^3} & \frac{6EI}{L^2} & 0 & \frac{-12EI}{L^3} & \frac{6EI}{L^2} \\ 0 & \frac{6EI}{L^2} & \frac{4EI}{L} & 0 & \frac{-6EI}{L^2} & \frac{2EI}{L} \\ \frac{-AE}{L} & 0 & 0 & \frac{AE}{L} & 0 & 0 \\ 0 & \frac{-12EI}{L^3} & \frac{-6EI}{L^2} & 0 & \frac{12EI}{L^3} & \frac{-6EI}{L^2} \\ 0 & \frac{6EI}{L^2} & \frac{2EI}{L} & 0 & \frac{-6EI}{L^2} & \frac{4EI}{L} \end{bmatrix}$$

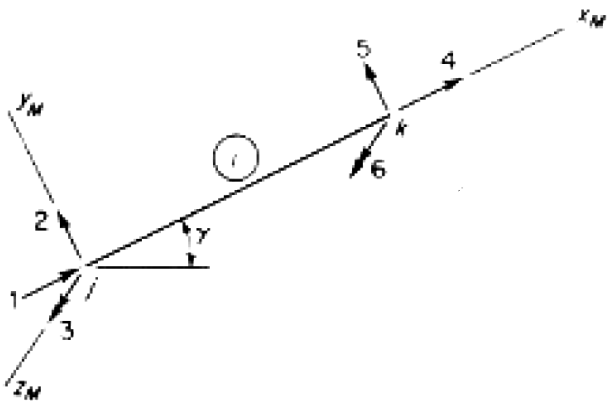

Figure 5.3: Plane Frame Member axes and degrees of freedom

Results of sequential and parallel implementation are measured in millisecond i.e ms. CPU-S stands for sequential time and CPU-P stands for parallel time. CPU-P is summation of execution time and communication time. Communication time depends on bandwidth of computer system considered. Bandwidth refers to amount of data transferred per unit time. Different computer systems have different bandwidth of data transfer.

Table-5.1 shows results of speedup for Analysis of Plane Frame with different number of bays and storeys tested on Intel® Core™i3-3210 Processor(3M Cache,3.20 GHz). It is observed that for small frame speedup factor is less than unity. Maximum speedup factor 1.782 is obtained in case of $200bay \times 200storey$. Given hardware gives optimum results from plane frame having 120600 degrees of freedom. Fig.5.4 is graphical representation of the results.

Table 5.1: Performance Comparison of Plane Frame Analysis using Intel® Core™i3-3210 Processor(3M Cache,3.20 GHz)

NO OF BAYS & STOREYS	MATRIX SIZE	CPU-S (ms) i3	EXECUTION TIME(ms)	COMMUNICATION TIME(ms)	TOTAL TIME CPU-P(ms)	SPEEDUP
50X50	7650 x 156	390	312	343	655	0.595
100X100	30300 x 306	5709	1249	3260	4509	1.266
150X150	67950 x 456	28314	2902	13400	16302	1.737
200X200	120600 x 606	92540	10686	41246	51932	1.782
250X250	188250 x 756	223922	75738	493429	569167	0.393

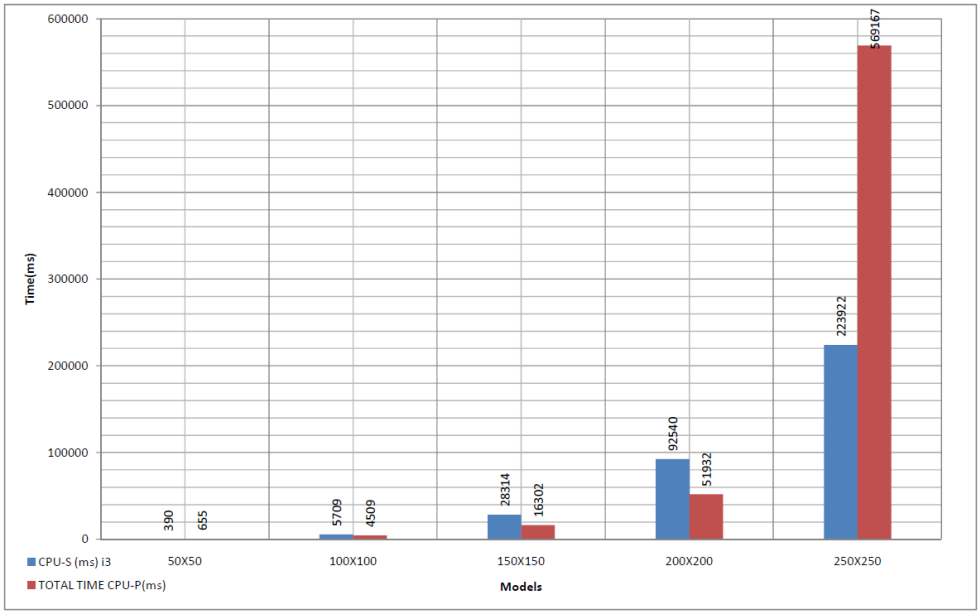

Figure 5.4: Performance Comparison of Plane Frame Analysis using Intel® Core™i3-3210 Processor(3M Cache,3.20 GHz)

Table-5.2 shows results of speedup for Analysis of Plane Frame with different number of bays and storeys tested on Intel® Core^{TM}i5-3450 Processor(6M Cache,3.50GHz). It is observed that for small frame speedup factor is less than unity. Maximum speedup factor 2.847 is obtained in case of $150bay \times 150storey$. Given hardware gives optimum results from plane frame having 67950 degrees of freedom.Fig.5.5 is graphical representation of the results.

Table 5.2: Performance Comparison of Plane Frame Analysis using Intel® Core^{TM}i5-3450 Processor(6M Cache,3.50GHz)

NO OF BAYS & STOREYS	MATRIX SIZE	CPU-S (ms) i5	EXECUTION TIME(ms)	COMMUNICATION TIME(ms)	TOTAL TIME CPU-P(ms)	SPEEDUP
50X50	7650 x 156	406	188	264	452	0.898
100X100	30300 x 306	5897	780	2138	2918	2.021
150X150	67950 x 456	29219	1685	8579	10264	2.847
200X200	120600 x 606	94053	9750	28190	37940	2.479
250X250	188250 x 756	228665	42401	279006	321407	0.711

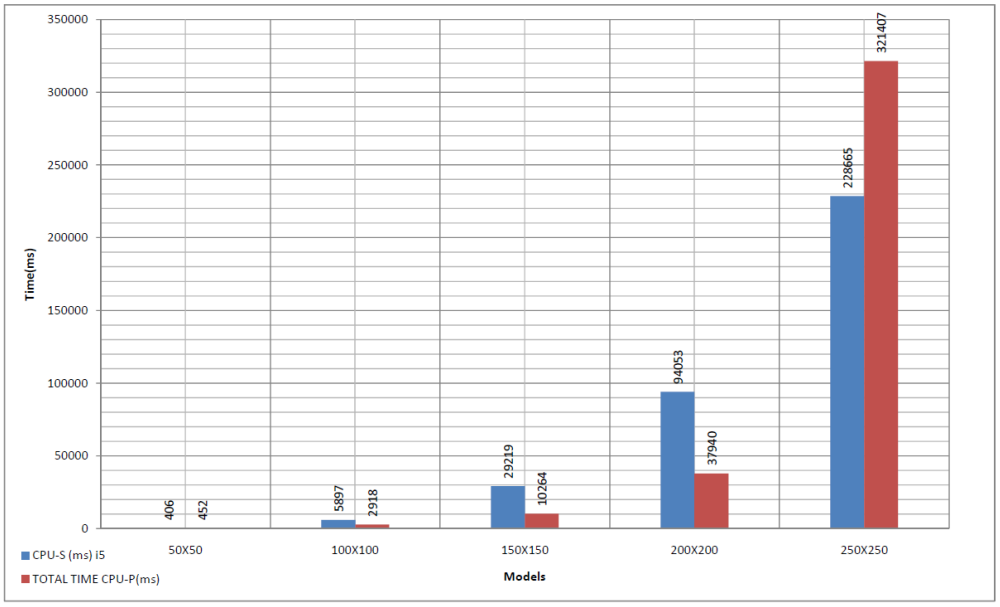

Figure 5.5: Performance Comparison of Plane Frame Analysis using Intel® Core^{TM}i5-3450 Processor(6M Cache,3.50GHz)

Table-5.3 shows results of speedup for Analysis of Plane Frame with different number of bays and storeys tested on Intel® Core^{TM}i7-3450 Processor(6M Cache,3.50GHz). It is observed that for small frame speedup factor is less than unity. Maximum speedup factor 3.910 is obtained in case of $250bay \times 250storey$. Given hardware gives optimum results from plane frame having 188250 degrees of freedom.Fig.5.6 is graphical representation of the results.

Table 5.3: Performance Comparison using Intel® Core^{TM}i7-3450 Processor(6M Cache,3.50GHz)

NO OF BAYS & STOREYS	MATRIX SIZE	CPU-S (ms) i7	EXECUTION TIME(ms)	COMMUNICATION TIME(ms)	TOTAL TIME CPU-P(ms)	SPEEDUP
50X50	7650 x 156	328	1466	38	1504	0.218
100X100	30300 x 306	4875	5725	287	6012	0.811
150X150	67950 x 456	23861	12395	957	13352	1.787
200X200	120600 x 606	77340	21724	2242	23966	3.227
250X250	188250 x 756	187690	43630	4370	48000	3.910

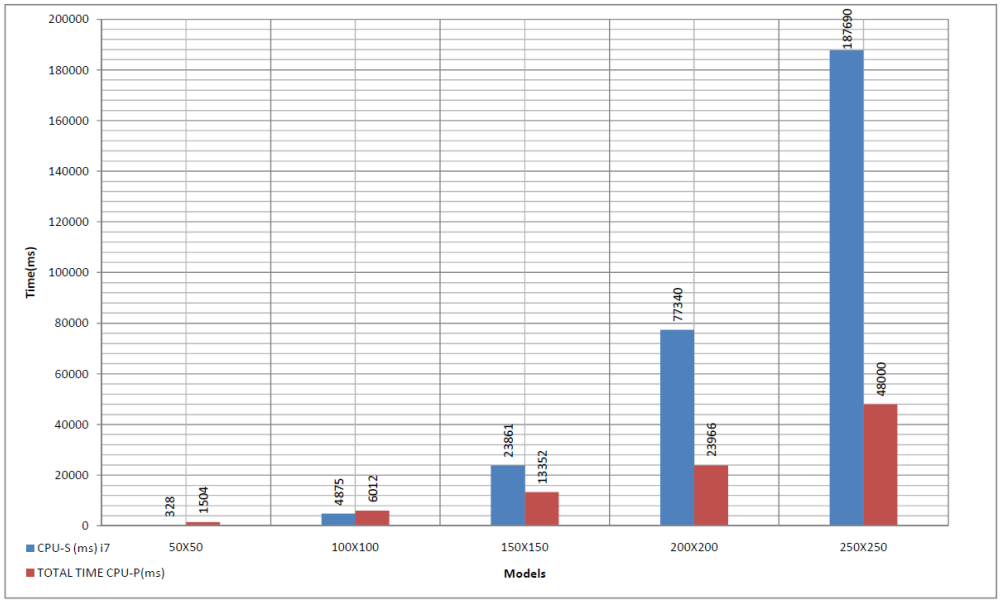

Figure 5.6: Performance Comparison of Plane Frame Analysis using Intel® Core^{TM}i7-3450 Processor(6M Cache,3.50GHz)

Table-5.4 shows results of speedup for Analysis of Plane Frame with different number of bays and storeys tested on Intel® Core™i7-2630QM Processor(2.0GHz). It is observed that for small frame speedup factor is less than unity. Maximum speedup factor 2.017 is obtained in case of $100bay \times 100storey$. Given hardware gives optimum results from plane frame having 30300 degrees of freedom.Fig.5.7 is graphical representation of the results.

Table 5.4: Performance Comparison using Intel® Core™i7-2630QM Processor(2.0GHz)

NO OF BAYS & STOREYS	MATRIX SIZE	CPU-S (ms) i7 Laptop	EXECUTION TIME(ms)	COMMUNICATION TIME(ms)	TOTAL TIME CPU-P(ms)	SPEEDUP
50X50	7650 x 156	718	983	15	998	0.719
100X100	30300 x 306	10795	5116	235	5351	2.017
150X150	67950 x 456	53727	28767	561	29328	1.832
200X200	120600 x 606	168668	101369	1075	102444	1.646
250X250	188250 x 756	411107	264749	1716	266465	1.543

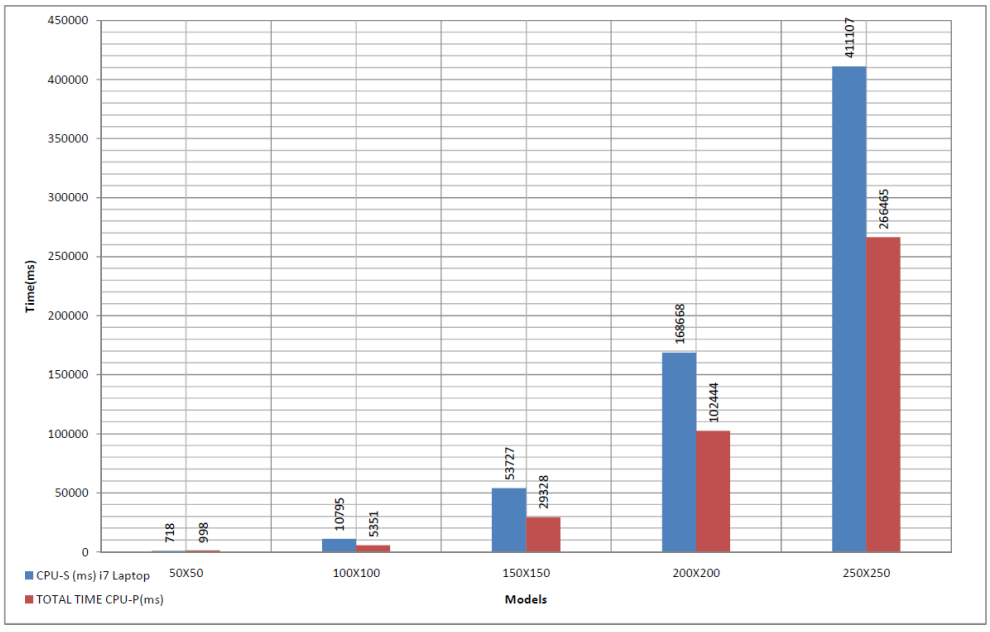

Figure 5.7: Performance Comparison using Intel® Core™i7-2630QM Processor(2.0GHz)

Table-5.5 shows results of speedup for Analysis of Plane Frame with different number of bays and storeys tested on Intel® Core™i7-2630QM Processor(2.0GHz). It is observed that for small frame speedup factor is less than unity. Maximum speedup factor 2.023 is obtained in case of 100bay × 100storey. Given hardware gives optimum results from plane frame having 30300 degrees of freedom.Fig.5.8 is graphical representation of the results.

Table 5.5: Performance Comparison using NVIDIA GeForce GT 525M

NO OF BAYS & STOREYS	MATRIX SIZE	CPU-S (ms) i7 Laptop	EXECUTION TIME(ms)	COMMUNICATION TIME(ms)	TOTAL TIME GPU-P(ms)	SPEEDUP
50X50	7650 x 156	718	702	-	702	1.023
100X100	30300 x 306	10795	5117	218	5335	2.023
150X150	67950 x 456	53727	28798	530	29328	1.832
200X200	120600 x 606	168668	101448	1091	102539	1.645
250X250	188250 x 756	411107	264827	1731	266558	1.542

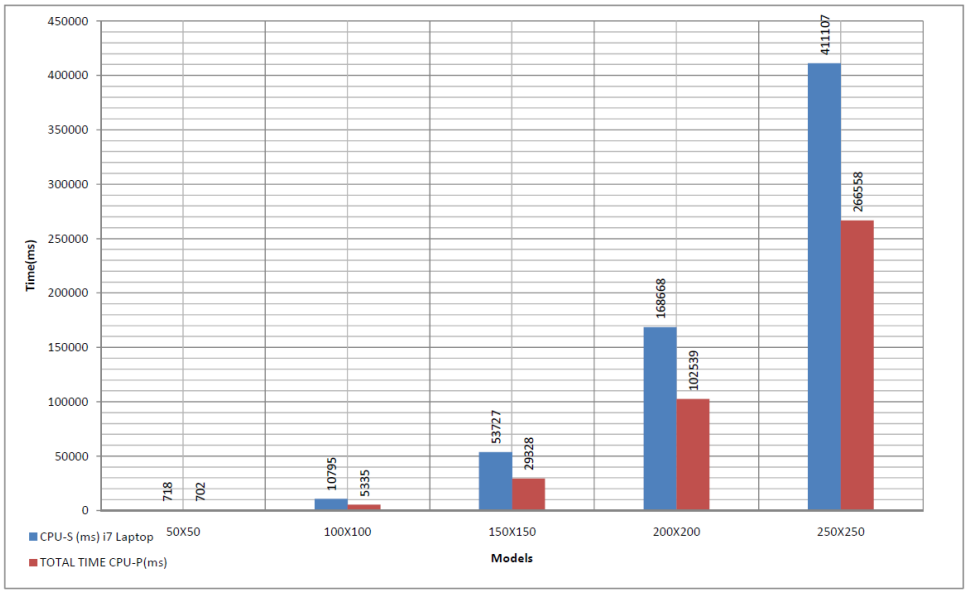

Figure 5.8: Performance Comparison using NVIDIA GeForce GT 525M

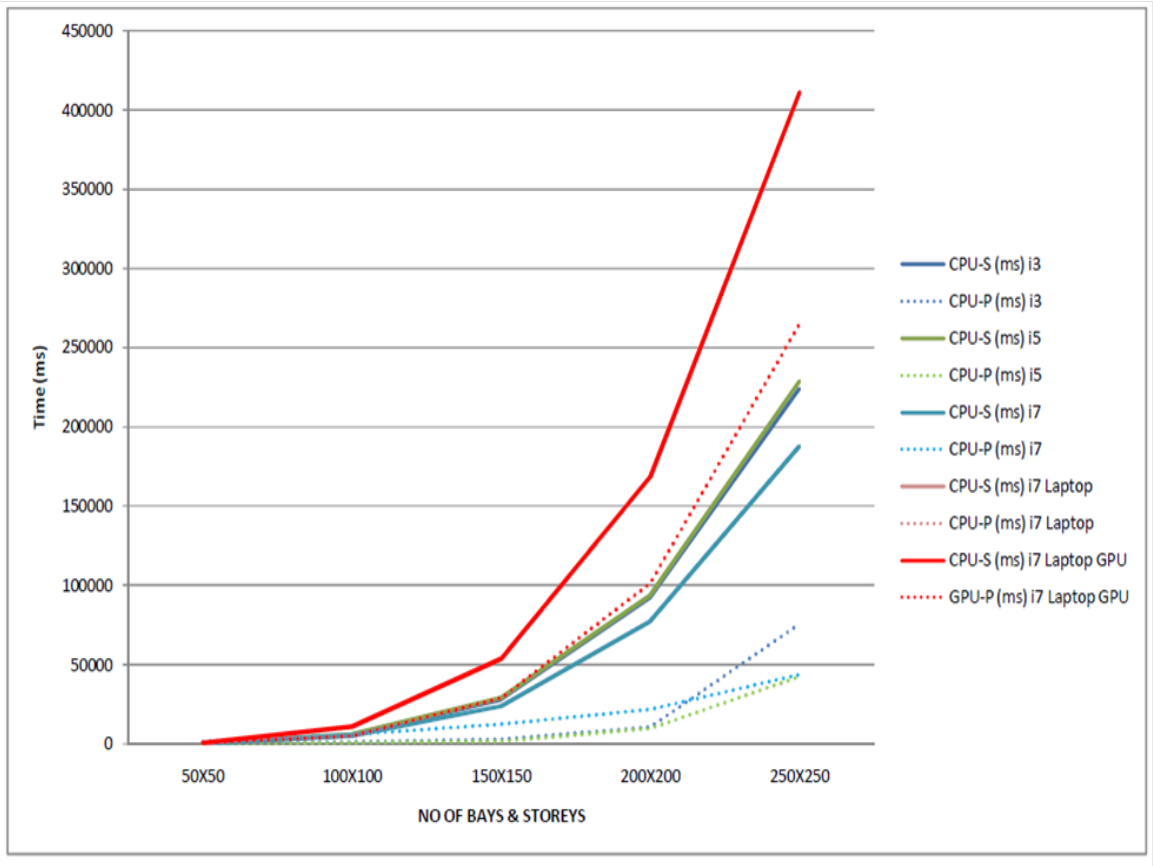

Figure 5.9: Comparison of Execution Time of Different Hardwares for Plane Frame Analysis

Fig.5.9 shows comparison of execution time of both parallel and sequential code of Half-Band Solver. Solid line and dashed line represents results of sequential execution time and parallel execution time respectively. It is observed that parallel execution time lesser than sequential execution time for all the types of hardwares used in parametric study. Parallel execution time of Intel® Core^{TM}i5-3450 Processor is least followed by Intel® Core^{TM}i7-3450 Processor.

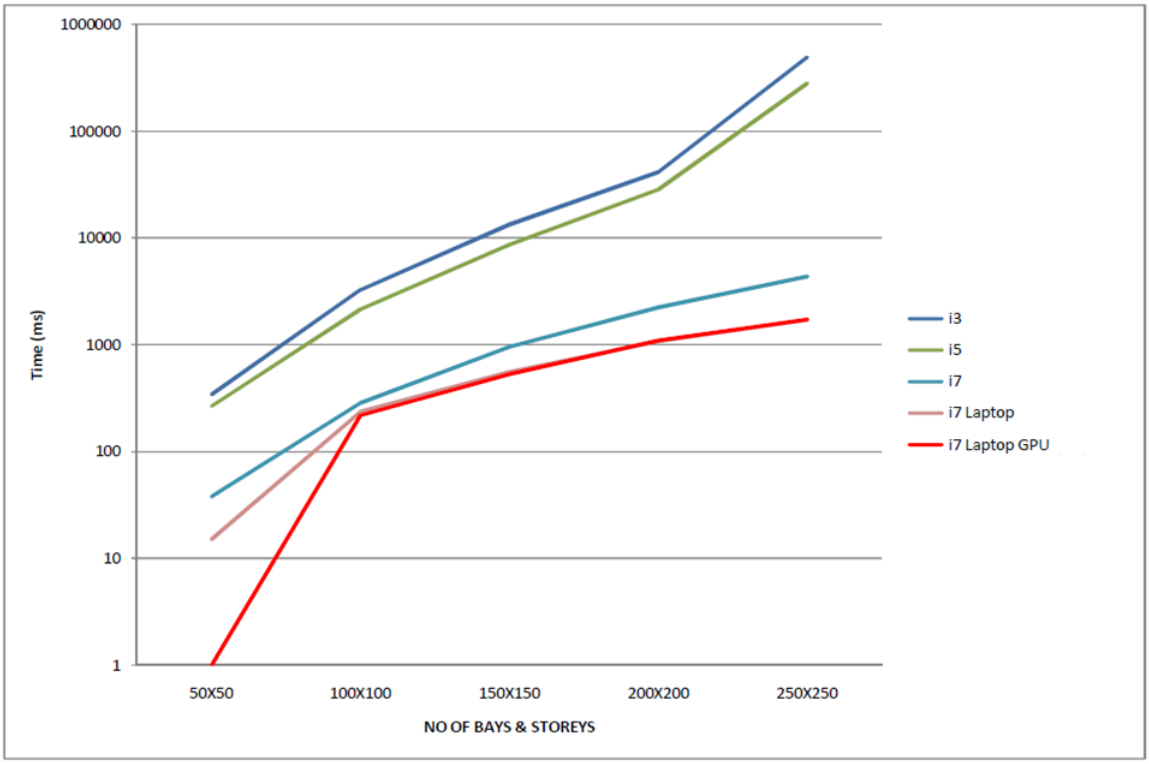

Figure 5.10: Comparison of Communication Time of Different Hardwares for Plane Frame Analysis

Fig.5.10 shows comparison of communication time of all 5 computer systems used for parametric study. Intel® Core^{TM}i3-3210 Processor system is taking maximum communication time. This is due to large large amount of data transfer. Intel® Core^{TM}i7-2630QM Processor and NVIDIA GeForce GT 525M GPU are taking least communication time among all 5 computer systems.

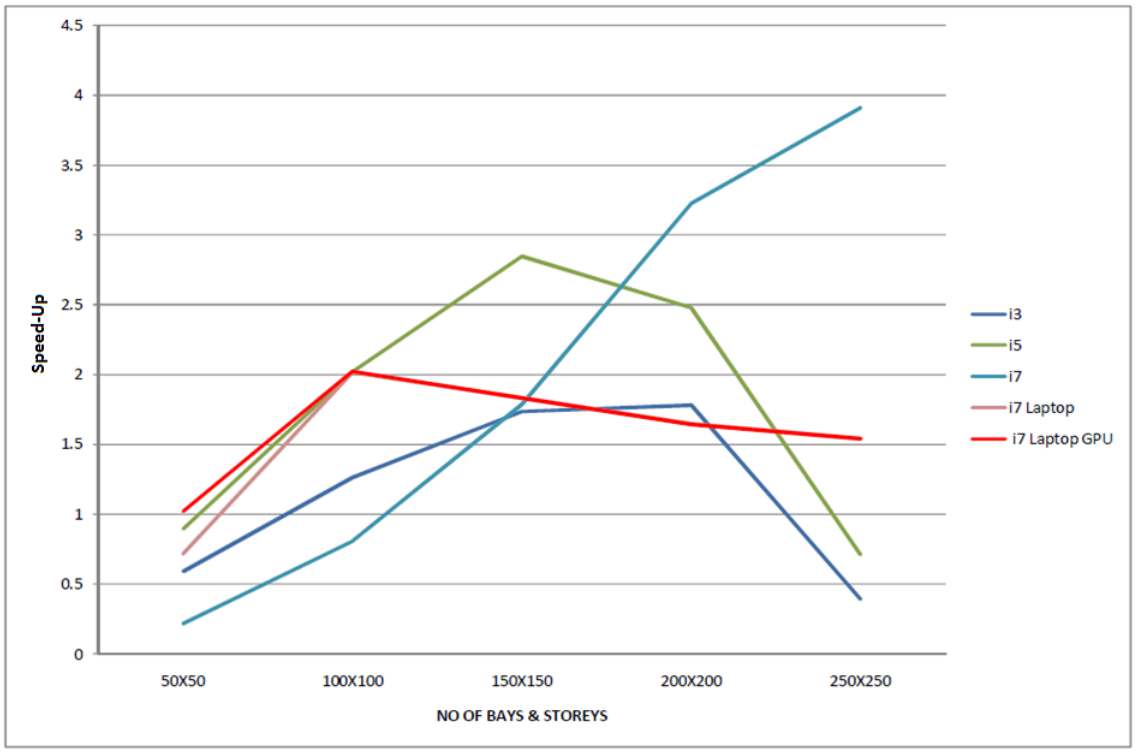

Figure 5.11: Comparison of Speedup factor of Different Hardwares for Plane Frame Analysis

Fig.5.11 shows comparison of speedup factor of all 5 computer systems considered. Intel® Core™i7-3450 Processor gives maximum speedup factor of 3.910 among all 5 computer systems for plane frame having size 250bay × 250storey followed by Intel® Core™i5-3450 Processor with speedup factor of 2.847 for plane frame having size 150bay × 150storey. For parallel implementation of Half-Band Solver for plane frame analysis , Intel® Core™i7-3450 Processor gives optimum performance among 5 computer systems.

5.6 Space Frame Analysis

In the case of space frame, all the members lie in the 3 different planes and are interconnected by rigid joints. The internal forces at a cross-section of a space frame member consist of 3-bending moments, 2-shear forces and an axial force.Three different space frame models, $5 \times 5 \times 5$, $10 \times 10 \times 10$ and $20 \times 20 \times 20$ ($bays - x \times bays - y \times storeys - z$) are considered. Fig.5.14 shows typical space frame diagram.

$$
\begin{bmatrix}
\frac{EA}{L} & & & & & & & & & & & \\
0 & \frac{12EI_z}{L^3} & & & & & & & & & & \\
0 & 0 & \frac{12EI_y}{L^3} & & & & & \textbf{symmetric} & & & & \\
0 & 0 & 0 & \frac{GJ}{L} & & & & & & & & \\
0 & 0 & \frac{-6EI_y}{L^2} & 0 & \frac{4EI_y}{L} & & & & & & & \\
0 & \frac{6EI_z}{L^2} & 0 & 0 & 0 & \frac{4EI_z}{L} & & & & & & \\
\frac{-EA}{L} & 0 & 0 & 0 & 0 & 0 & \frac{EA}{L} & & & & & \\
0 & \frac{-12EI_z}{L^3} & 0 & 0 & 0 & \frac{-6EI_y}{L^2} & 0 & \frac{12EI_z}{L^3} & & & & \\
0 & 0 & \frac{-12EI_y}{L^3} & 0 & \frac{6EI_y}{L^2} & 0 & 0 & 0 & \frac{12EI_y}{L^3} & & & \\
0 & 0 & 0 & \frac{-GJ}{L} & 0 & 0 & 0 & 0 & 0 & \frac{GJ}{L} & & \\
0 & 0 & \frac{-6EI_y}{L^2} & 0 & \frac{2EI_y}{L} & 0 & 0 & 0 & \frac{6EI_y}{L^2} & 0 & \frac{4EI_y}{L} & \\
0 & \frac{6EI_z}{L^2} & 0 & 0 & 0 & \frac{2EI_z}{L} & 0 & \frac{-6EI_y}{L^2} & 0 & 0 & 0 & \frac{4EI_z}{L}
\end{bmatrix}
$$

Figure 5.12: Space Frame Member Stiffness Matrix

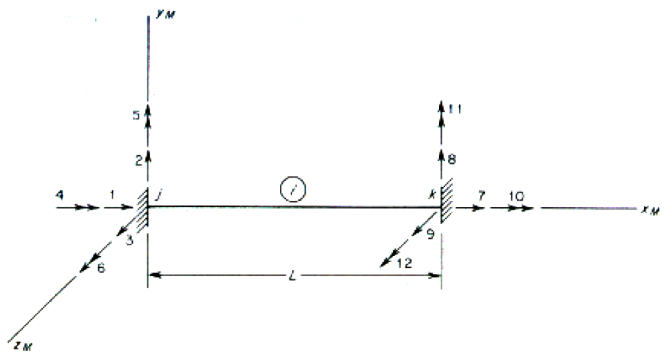

Figure 5.13: Space Frame Member axes and degrees of freedom

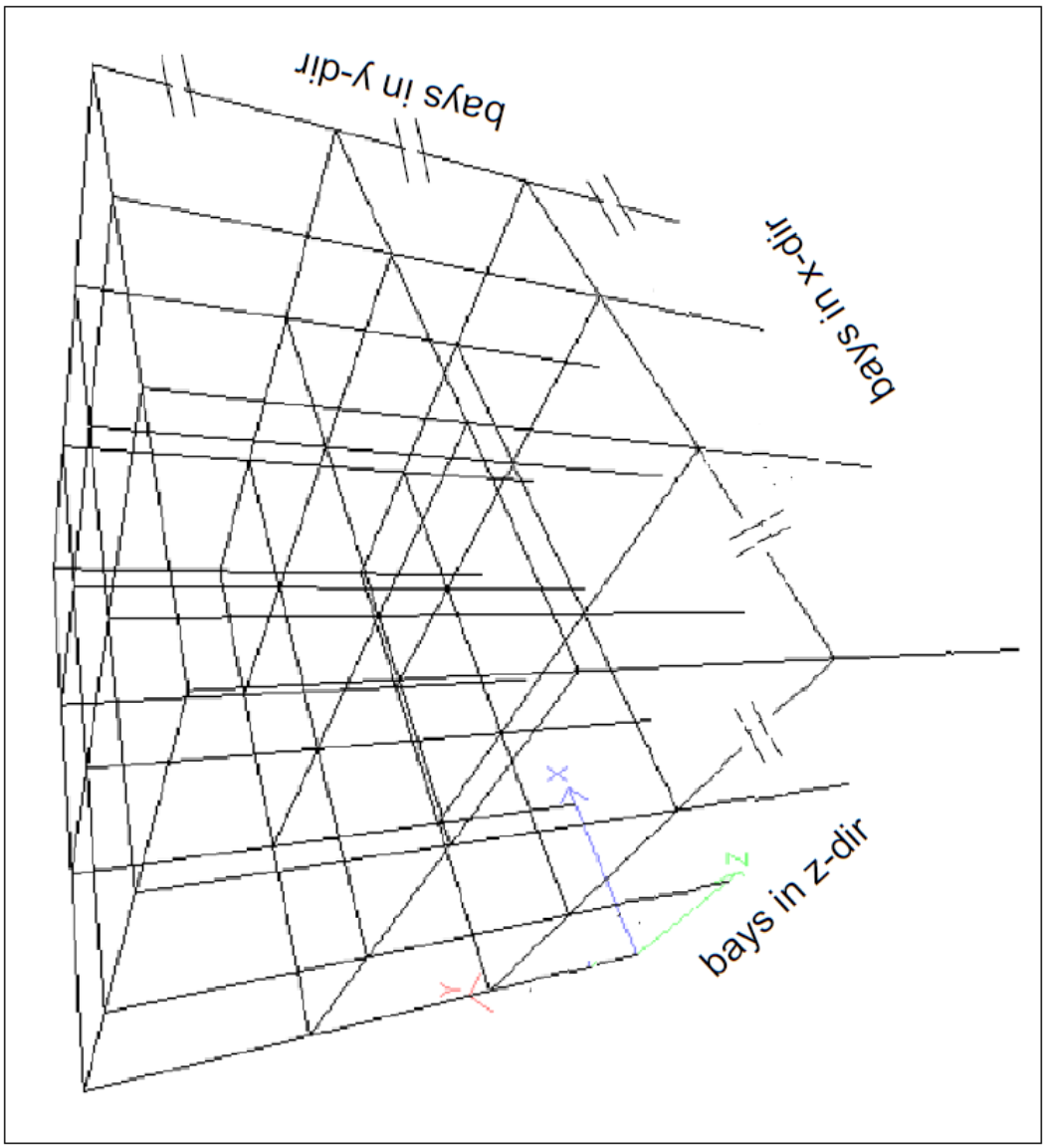

Figure 5.14: Space Frame Schematic Diagram

Table-5.6 shows results of speedup for Analysis of Space Frame with different number of bays and storeys tested on Intel® Core™i3-3210 Processor(3M Cache,3.20 GHz). It is observed that for small space frame speedup factor is less than unity. Maximum speedup factor 1.492 is obtained in case of $10 \times 10 \times 10$ space frame model.Fig.5.15 is graphical representation of the results.

Table 5.6: Performance Comparison using Intel® Core™i3-3210 Processor(3M Cache,3.20 GHz)

NO OF BAYS & STOREYS	MATRIX SIZE	CPU-S (ms) i3	EXECUTION TIME(ms)	COMMUNICATION TIME(ms)	TOTAL TIME CPU-P(ms)	SPEEDUP
5X5X5	1080 x 222	93	47	63	110	0.845
10X10X10	7260 x 732	7192	342	4477	4819	1.492
20X20X20	52920 x 2652	731922	10249	855802	866051	0.845

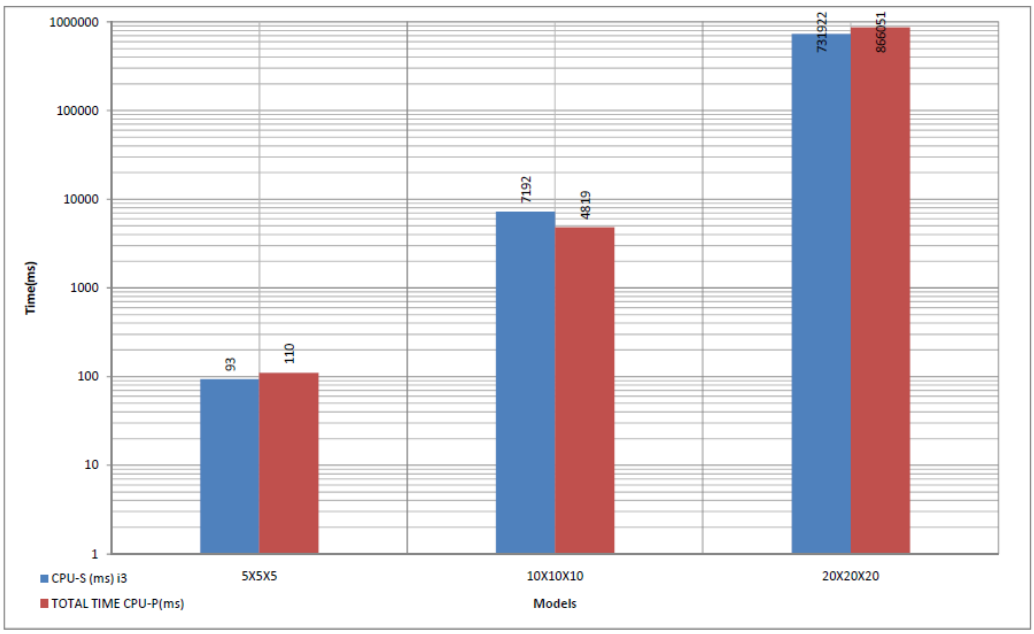

Figure 5.15: Performance Comparison using Intel® Core™i3-3210 Processor(3M Cache,3.20 GHz)

Table-5.7 shows results of speedup for Analysis of Space Frame with different number of bays and storeys tested on Intel® Core^{TM}i5-3450 Processor(6M Cache,3.50GHz). It is observed that for small space frame speedup factor is less than unity. Maximum speedup factor 3.4 is obtained in case of $10 \times 10 \times 10$ space frame model. Fig.5.16 is graphical representation of the results.

Table 5.7: Performance Comparison using Intel® Core^{TM}i5-3450 Processor(6M Cache,3.50GHz)

NO OF BAYS & STOREYS	MATRIX SIZE	CPU-S (ms) i5	EXECUTION TIME(ms)	COMMUNICATION TIME(ms)	TOTAL TIME CPU-P(ms)	SPEEDUP
5X5X5	1080 x 222	94	32	46	78	1.205
10X10X10	7260 x 732	7425	187	1997	2184	3.400
20X20X20	52920 x 2652	749518	6662	491713	498375	1.504

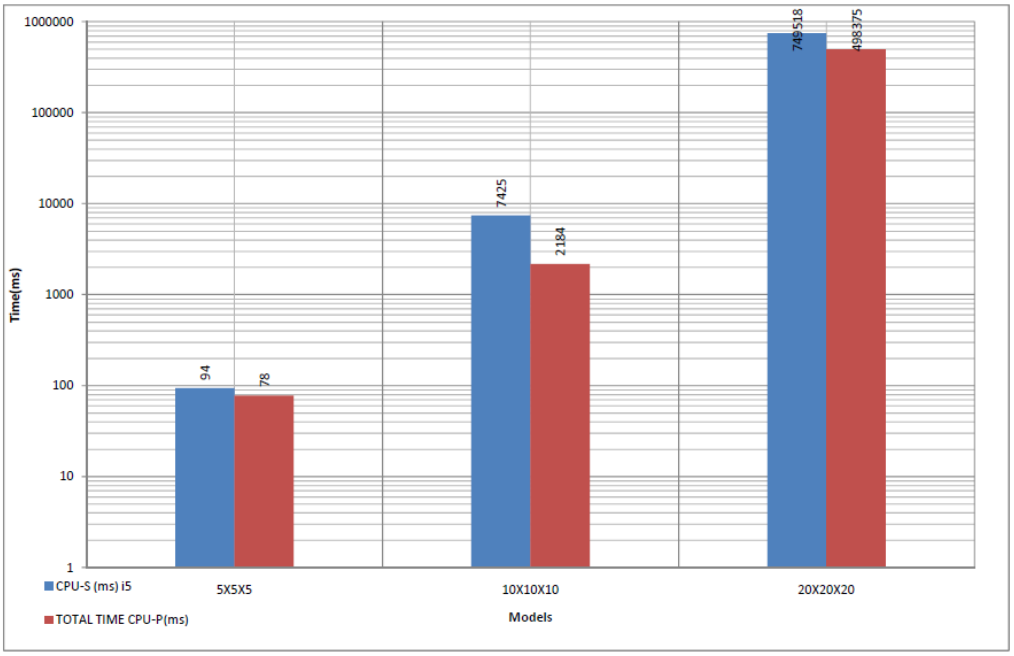

Figure 5.16: Performance Comparison using Intel® Core^{TM}i5-3450 Processor(6M Cache,3.50GHz)

Table-5.8 shows results of speedup for Analysis of Space Frame with different number of bays and storeys tested on Intel® Core^{TM}i7-3450 Processor(6M Cache,3.50GHz). It is observed that for small space frame speedup factor is less than unity. Maximum speedup factor 5.096 is obtained in case of $20 \times 20 \times 20$ space frame model.Fig.5.17 is graphical representation of the results.

Table 5.8: Performance Comparison using Intel® Core^{TM}i7-3450 Processor(6M Cache,3.50GHz)

NO OF BAYS & STOREYS	MATRIX SIZE	CPU-S (ms) i7	EXECUTION TIME(ms)	COMMUNICATION TIME(ms)	TOTAL TIME CPU-P(ms)	SPEEDUP
5X5X5	1080 x 222	83	201	9	210	0.395
10X10X10	7260 x 732	6247	1619	166	1785	3.500
20X20X20	52920 x 2652	624549	118248	4305	122553	5.096

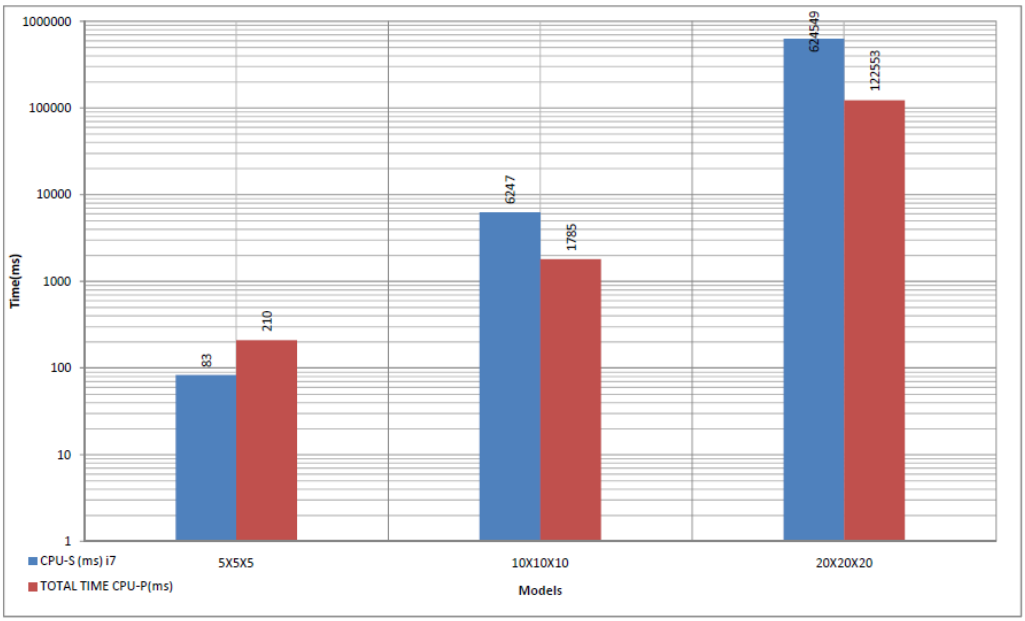

Figure 5.17: Performance Comparison using Intel® Core^{TM}i7-3450 Processor(6M Cache,3.50GHz)

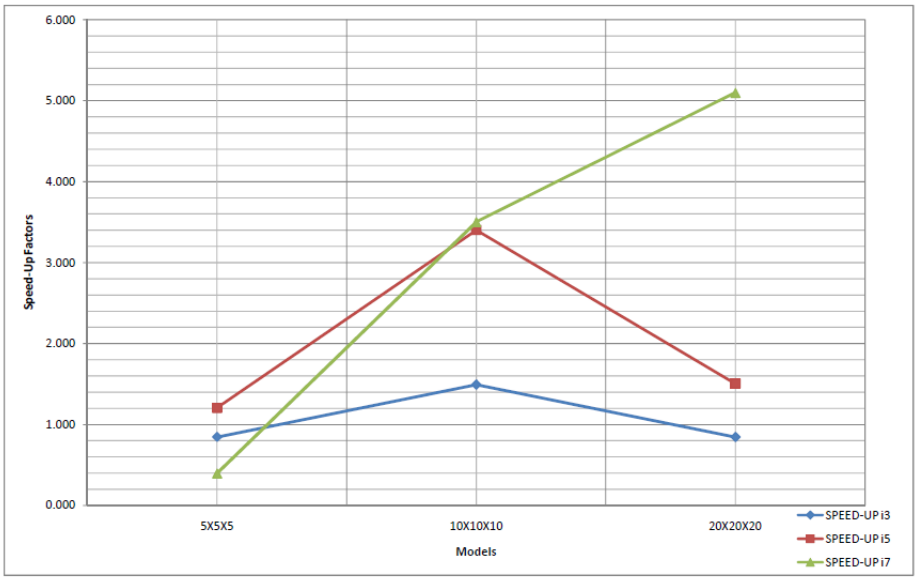

Figure 5.18: Speedup Factor Comparison

Fig.5.18 shows speedup factor comparison for all 3 models of Space Frame using 3 different processors. It is observed that Intel® Core^{TM}i7-3450 Processor(6M Cache,3.50GHz) gives maximum speedup factor.

5.7 Summary

In this chapter, Half-Band Storage method is described with the help of an example. Half-Band solver for solving linear equations i.e. [A]{x}={B} where [A] is rectangular matrix, with the help of an algorithm is presented. Its sequential and parallel implementations on multi-core CPUs and GPUs is presented. Different size of plane frame and space frame are considered for sequential and parallel implementation. Results of execution time, communication time and speedup factor are presented.

Chapter 6

Summary and Conclusion

6.1 Summary

Solution of linear equation system in form of [A]{x}={B}, is highly compute intensive in structural analysis program. It is implemented on high performance computing platforms like multi-core processors and graphics processing units in present study. In structural analysis, Matrix [A] representing Stiffness matrix, can be stored in square form or half-band form. As stiffness matrix is square, symmetric consisting of many zero elements, half-band storage will require less memory and reduces computations. In present study Gauss Elimination and modified Cholesky factorization methods are used for solving linear equations having Matrix [A] in square form and half-band form. OpenCL and C++ programming language are used for parallel and sequential implementation over variety of high performance computing platform.

For parallel implementation of Gaussian Elimination solver, linear equations system representing equilibrium equations of finite element problem is used. Equations in form of [A]{x}={B} are generated from finite element analysis of axial bar using 3-node bar element where A=Square Stiffness Matrix, B=Load Vector and x=Displacement vector. For solution of equations Matrix-[A] is inverted using se-

quential and parallel implementation of Gaussian Elimination. Sequential program is developed using C++ and parallel program is developed using OpenCL language. For comparing computational efficiency of parallel code, speedup factor which is ratio of sequential execution time to parallel execution time is calculated for different number of linear equations ranging 101 to 10001. Parallel execution time includes processing time and communication time. As data is transferred between various memories, communication time increases total computational time. Code is executed on different CPUs and GPUs for parametric study.

For parallel implementation of Half-Band solver, which is based on modified cholesky method, Direct Stiffness Method program of Plane Frame and Space Frame are used for generating set of linear equation system. Here stiffness matrix is stored in banded form to reduce memory requirements. Programs for sequential and parallel solution of banded equations are developed using C++ and OpenCL languages. Problems of varying size from 7650 Degrees of Freedom to 1,88,250 Degrees of Freedom are solved using sequential and parallel Half-Band solver. The computational efficiency of parallel code is studied based on speedup factor. Further to understand the efficiency of program on different hardware platform, the parallel code is executed on multi-core CPUs like Intel® Core^{TM}i3, i5, i7 processors with different specifications and NVIDIA GPU.

6.2 Conclusion

Based on present study, following conclusions are derived:

- Parallel processing reduces computing time for solving linear equation system in form of [A]{x}={B} significantly.
- Various types of CPUs and GPUs have different computing capacities depending on number of cores present in them.

- Performance of the parallel implementation depends on the type of numerical problem taken and data-dependencies.

- Gauss Elimination parallel implementation tested on different CPUs and GPUs reveals that it gives better efficiency.

- Use of CPU as parallel processing hardware results in significant reduction in communication time. On other hand, use of GPU as parallel processing hardware results in significant reduction in execution time due to large number of computing cores present.

- Parallel Implementation of Gauss Elimination

  - Intel® Core^{TM}i5-3450 Processor exhibits least execution time of 78ms for solving 5001 number of linear equations system.

  - Computer system with Intel® Core^{TM}i7-3450 Processor exhibits maximum communication time for all set of linear equation system due to multiple slot 32GB RAM.

  - Maximum speed-up factor of 1702 is achieved by system having Intel® Core^{TM}i7-2630QM Processor for solving 10001 number of linear equations.

  - Intel® Core^{TM}i7-2630QM Processor (2.0GHz) is best suited for the parallel implementation of problem considered in this study.

- Parallel Implementation of Half-Band solver

  - Intel® Core^{TM}i5-3450 Processor exhibits least execution time of 42401ms for solving plane frame of 250bay$\times$250 storey.

  - Maximum speedup factor of 3.9 is achieved by system having Intel® Core^{TM}i7-3450 Processor for solving plane frame of 250bay$\times$250 storey with 188250 Degrees of Freedom and 756 half-bandwidth.

- Maximum speedup factor of 3.5 is achieved by Intel® Core™i7-3450 Processor for solving space frame of 20×20×20(bays-x×bays-y×storey) with 52920 Degrees of Freedom and 2652 half-bandwidth.

6.3 Future Scope of Work

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

- Parallelization of complete structural analysis problem rather than only concentrating on equation solution.
- Various structural engineering problems like FEM analysis using numerical integration, non-linear dynamic analysis.
- Development of efficient algorithm to suit different hardware platforms.
- A heterogenous program which uses both GPU and CPU simultaneously for carrying out different instruction on different data.

Appendix A

Gauss Elimination host program

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

#ifdef __APPLE__
#include <OpenCL/opencl.h>
#else
#include <CL/cl.h>
#endif

#define cal(zz,qq) ((qq *) calloc(zz,sizeof(qq)))

using namespace std;

const char *ProgramSource =
```

```
"__kernel void add(__global float *inputM, __global float *
   inputA, __global float *inputB,const int size,const int t)
   \n"\
"{\n"\
"    int globalId = get_global_id(0);\n"\
"        if (globalId < size-1-t)\n"\
"        {\n"\
"                        *(inputM + size * (globalId + t + 1)+
   t) = *(inputA + size * (globalId + t + 1) + t) / *(inputA
   + size * t + t);\n"\
"    }\n"\

"}\n";

const char *ProgramSource2 =
"__kernel void add2(__global float *inputM, __global float *
   inputA, __global float *inputB,const int size,const int t)
   \n"\
"{\n"\
"            int globalIdx = get_global_id(0);\n"\
"            int globalIdy = get_global_id(1);\n"\
"        if (globalIdx < size-1-t && globalIdy < size-t)\n
   "\
"            {\n"\
"                                    inputA[size*(
   globalIdx+1+t)+(globalIdy+t)] -= inputM[size*(globalIdx+1+
   t)+t] * inputA[size*t+(globalIdy+t)];\n"\
"        if(globalIdy == 0)\n"\
"                    {\n"\
```

```
"                                                inputB[globalIdx+1+t]
    -= inputM[size*(globalIdx+1+t)+(globalIdy+t)] * inputB[t
    ];\n"\
"                                }\n"\
"                        }\n"\
"}\n";

int main()
{

        int ne,nj,i,nrj,j,b,im[5],n,u;
    float temp,temp1;



    FILE *f1,*f2;
    system("cls");
        f1=fopen("input.txt","r");
    f2=fopen("out.txt","w");
        FILE *ft=fopen("time.txt","w");

    fscanf(f1,"%d",&ne);
    nj=(2*ne)+1;
        fprintf(ft,"Number of Equations=%d\n",nj);

        clock_t c4=clock();
        float * ac = (float *)malloc(sizeof(float)*(nj+1));
        float **kj = (float **) calloc(nj+1,sizeof(float *));
        for(i=1;i<=nj;i++)
                kj[i] = (float *) calloc(nj+1,sizeof(float));
```

```
        clock_t c5=clock();
        double c6=((c5-c4)*1000)/CLOCKS_PER_SEC;;
        //cl starts


cl_context context;
cl_context_properties properties[3];
cl_kernel kernel,kernel2;
cl_command_queue command_queue;
cl_program program,program2;
cl_int err;
cl_uint num_of_platforms=0;
cl_platform_id platform_id;
cl_device_id device_id;
cl_uint num_of_devices=0;
cl_mem inputA, inputB,inputM;
cl_int ret;


size_t global,global2[2],local2[2];
int DATA_SIZE=nj*nj;
float *inputDataA= (float *)malloc(sizeof(float)*DATA_SIZE);
float *finalVec= (float *)malloc(sizeof(float)*nj);
float *inputDataB= (float *)malloc(sizeof(float)*nj);
float *inputDataM= (float *)malloc(sizeof(float)*DATA_SIZE);

FILE *f3=fopen("out.csv","w");
```

```
fprintf(f3,"Before kernel matrix\n");
for(i=0;i<1*nj;i++)
    {
        for(j=0;j<1*nj;j++)
                {
                        fscanf(f1,"%f",&inputDataA[j + nj*i
                            ]);
                        kj[i+1][j+1]=inputDataA[j + nj*i ];
                        inputDataM[j + nj*i ] = 0;
                        //fprintf(f3,"%f,",inputDataA[j + nj*
                            i ]);

                }
            //fprintf(f3,"\n");
    }
/*fprintf(f2,"Global Stiffness matrix\n");
    for(i=1;i<=1*nj;i++)
    {
        for(j=1;j<=1*nj;j++)
            fprintf(f2,"S%d%d=%f      ",i,j,kj[i][j]);
            fprintf(f2,"\n");
    }*/
for(i=0;i<1*nj;i++)
    {
                fscanf(f1,"%f",&inputDataB[i]);
                ac[i+1]=inputDataB[i];

        }
/*fprintf(f2,"AC\n");
```

```
    for(i=1;i<=1*nj;i++)
    {
        fprintf(f2,"AC-%d=%f\n",i,ac[i]);
    }*/
// retreive a list of platforms avaible
if (clGetPlatformIDs(1, &platform_id, &num_of_platforms)!=
   CL_SUCCESS)
{
printf("Unable to get platform_id\n");
return 1;
}

// try to get a supported GPU device
if (clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_CPU, 1, &
   device_id, &num_of_devices) != CL_SUCCESS)
{
printf("Unable to get device_id\n");
return 1;
}

// context properties list - must be terminated with 0
properties[0]= CL_CONTEXT_PLATFORM;
properties[1]= (cl_context_properties) platform_id;
properties[2]= 0;

// create a context with the GPU device
context = clCreateContext(properties,1,&device_id,NULL,NULL,&
   err);
```

```
// create command queue using the context and device
command_queue = clCreateCommandQueue(context, device_id, 0, &
    err);

// create a program from the kernel source code
program = clCreateProgramWithSource(context,1,(const char **)
    &ProgramSource, NULL, &err);

// compile the program
if (clBuildProgram(program, 0, NULL, NULL, NULL, NULL) !=
    CL_SUCCESS)
{
printf("Error building program\n");
return 1;
}

// specify which kernel from the program to execute
kernel = clCreateKernel(program, "add", &err);

// create buffers for the input and ouput

inputA = clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(
    float) * DATA_SIZE, NULL, NULL);
inputM = clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(
    float) * DATA_SIZE, NULL, NULL);
inputB = clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(
    float) * nj, NULL, NULL);
```

```
// load data into the input buffer
clock_t g5=clock();
clEnqueueWriteBuffer(command_queue, inputA, CL_TRUE, 0,
    sizeof(float) * DATA_SIZE, inputDataA, 0, NULL, NULL);
clEnqueueWriteBuffer(command_queue, inputM, CL_TRUE, 0,
    sizeof(float) * DATA_SIZE, inputDataM, 0, NULL, NULL);
clEnqueueWriteBuffer(command_queue, inputB, CL_TRUE, 0,
    sizeof(float) * nj, inputDataB, 0, NULL, NULL);
clock_t g6=clock();
        double t3=((g6-g5)*1000)/CLOCKS_PER_SEC;;
 //part-2 starts
// create command queue using the context and device
command_queue = clCreateCommandQueue(context, device_id, 0, &
    err);

// create a program from the kernel source code
program2 = clCreateProgramWithSource(context,1,(const char
    **) &ProgramSource2, NULL, &err);

// compile the program
if (clBuildProgram(program2, 0, NULL, NULL, NULL, NULL) !=
    CL_SUCCESS)
{
printf("Error building program-2\n");
return 1;
}
```

```
// specify which kernel from the program to execute
kernel2 = clCreateKernel(program2, "add2", &err);
int size=nj;

size_t globalWorksizeFan1[1];
    size_t globalWorksizeFan2[2];

        globalWorksizeFan1[0] = 8;
        globalWorksizeFan2[0] = 8;
        globalWorksizeFan2[1] = 8;

// part-2 ends
clock_t g1=clock();
for (int t=0; t<(size-1); t++) {
        // kernel args
        cl_int argchk;
        argchk  = clSetKernelArg(kernel, 0, sizeof(cl_mem), (
           void *)&inputM);
        argchk |= clSetKernelArg(kernel, 1, sizeof(cl_mem), (
           void *)&inputA);
        argchk |= clSetKernelArg(kernel, 2, sizeof(cl_mem), (
           void *)&inputB);
        argchk |= clSetKernelArg(kernel, 3, sizeof(int), (
           void *)&size);
        argchk |= clSetKernelArg(kernel, 4, sizeof(int), (
           void *)&t);
```

```
        // launch kernel
        clEnqueueNDRangeKernel(command_queue,kernel, 1, 0,
            globalWorksizeFan1,NULL,0, NULL, NULL);



                // kernel args
                argchk  = clSetKernelArg(kernel2, 0, sizeof(
                    cl_mem), (void *)&inputM);
        argchk |= clSetKernelArg(kernel2, 1, sizeof(cl_mem),
            (void *)&inputA);
        argchk |= clSetKernelArg(kernel2, 2, sizeof(cl_mem),
            (void *)&inputB);
        argchk |= clSetKernelArg(kernel2, 3, sizeof(int), (
            void *)&size);
        argchk |= clSetKernelArg(kernel2, 4, sizeof(int), (
            void *)&t);




        // launch kernel
        clEnqueueNDRangeKernel(command_queue,  kernel2, 2, 0,
            globalWorksizeFan2,NULL,0, NULL, NULL);


        }

clock_t g2=clock();
        double t1=((g2-g1)*1000)/CLOCKS_PER_SEC;;
```

```
clock_t g3=clock();
// copy the results from out of the output buffer
clEnqueueReadBuffer(command_queue, inputA, CL_TRUE, 0, sizeof
   (float) *DATA_SIZE, inputDataA, 0, NULL, NULL);
//clEnqueueReadBuffer(command_queue, inputM, CL_TRUE, 0,
   sizeof(float) *DATA_SIZE, inputDataM, 0, NULL, NULL);
clEnqueueReadBuffer(command_queue, inputB, CL_TRUE, 0, sizeof
   (float) *nj, inputDataB, 0, NULL, NULL);
clock_t g4=clock();
        double t2=((g4-g3)*1000)/CLOCKS_PER_SEC;;

// print the results
fprintf(f3,"After kernel matrix\n");
for(i=0;i<1*nj;i++)
    {
        for(j=0;j<1*nj;j++)
                {

                //      fprintf(f3,"%f,",inputDataA[j + nj*i
                   ]);


                }
            //fprintf(f3,"\n");
    }
// cleanup - release OpenCL resources
clReleaseMemObject(inputA);
clReleaseMemObject(inputB);
clReleaseMemObject(inputM);
clReleaseProgram(program);
```

```
clReleaseKernel(kernel);
clReleaseCommandQueue(command_queue);
clReleaseContext(context);

clock_t g7=clock();
for(i=0;i<size;i++)
{
                finalVec[size-i-1]=inputDataB[size-i-1];
                for(j=0;j<i;j++)
                {
                        finalVec[size-i-1]-=*(inputDataA+size
                            *(size-i-1)+(size-j-1)) * finalVec
                            [size-j-1];
                }
                finalVec[size-i-1]=finalVec[size-i-1]/ *(
                    inputDataA+size*(size-i-1)+(size-i-1));
        }
clock_t g8=clock();
        double t4=((g8-g7)*1000)/CLOCKS_PER_SEC;;
        double t5=t1+t2+t3+t4;
        fprintf(ft,"Time elapsed for GPU:%f\n",t5);
        fprintf(ft,"Time elapsed for execution:%f\n",t1+t4);


//cl ends

fprintf(f2,"SOLUTION OF EQUATION\n");
clock_t c1=clock();
    for(u=1;u<=(1*nj);u++)
```

```
{
    temp=kj[u][u];
    //fprintf(f2,"temp=%f",temp);
    for(j=1;j<=(1*nj);j++)
            kj[u][j]=kj[u][j]/temp;
            ac[u]=ac[u]/temp;
            //fprintf(f2,"S%d%d=%f      ",u,j,sj[u][j]);
            //fprintf(f2,"AC-%d=%f\n",u,ac[u]);
            for(i=1;i<=(1*nj);i++)
            {
                    if(i==u)continue;
                    temp1=kj[i][u];
                    for(j=1;j<=(1*nj);j++)

                            kj[i][j]=kj[i][j]-kj[u][j]*
                               temp1;
                            ac[i]=ac[i]-ac[u]*temp1;


            }
}
clock_t c2=clock();
    double c3=((c2-c1)*1000)/CLOCKS_PER_SEC;;
    double c7=c3+c6;
    fprintf(ft,"Time elapsed for CPU:%f\n",c7);
    fprintf(ft,"Time elapsed for communication:%f\n",t2+
       t3);
    /*

    fprintf(f2,"Modified Global Stiffness matrix\n");
```

```
    for(i=1;i<=1*nj;i++)
    {
        for(j=1;j<=1*nj;j++)
            fprintf(f2,"S%d%d=%f      ",i,j,kj[i][j]);
            fprintf(f2,"\n");
    }*/

        /*fprintf(f3,"\nModified Global Stiffness matrix\n");
    for(i=1;i<=1*nj;i++)
    {
        for(j=1;j<=1*nj;j++)
            fprintf(f3,"%f,",kj[i][j]);
            fprintf(f3,"\n");
    }*/
        fprintf(f2,"Modified AC\n");
    for(i=1;i<=1*nj;i++)
    {
        //fprintf(f2,"AC-%d=%f      GAC-%d=%f\n",i,ac[i],i,
            finalVec[i-1]);
    }

        free(ac);
        free(kj);
        free(inputDataA);
        free(inputDataB);
        free(inputDataM);
}
```

Appendix B

Half-Band Solver host program

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

#ifdef __APPLE__
#include <OpenCL/opencl.h>
#else
#include <CL/cl.h>
#endif

//#define DATA_SIZE 100
#define cal(zz,qq) ((qq *) calloc(zz,sizeof(qq)))

using namespace std;

const char *ProgramSource =
```

```
"__kernel void add(__global float *inputM, __global float *
   inputA, __global float *inputB,const int size,const int t,
   const int MRg)\n"\
"{\n"\
"    int globalId = get_global_id(0);\n"\
"        if (globalId < MRg)\n"\
"        {\n"\
"                        *(inputM + + size * t + globalId  +
   1) = *(inputA + size * t + globalId  + 1) / *(inputA +
   size * t  );\n"\
"    }\n"\

"}\n";

const char *ProgramSource2 =
"__kernel void add2(__global float *inputM, __global float *
   inputA, __global float *inputB,const int size,const int t,
   const int MRg)\n"\
"{\n"\
"               int globalIdx = get_global_id(0);\n"\
"               int globalIdy = get_global_id(1);\n"\
"           if (globalIdx < MRg && globalIdy < (MRg-globalIdx
   ))\n"\
"               {\n"\
"                                       inputA[size*(
   globalIdx+1+t)+(globalIdy)] -= inputM[+ size * t +
   globalIdx  + 1] * inputA[size*t+(globalIdy)+1+globalIdx];\
   n"\
```

```
"                                                    //inputB[globalIdx+1+
   t] -= inputM[size*(globalIdx+1+t)+t] * inputB[t];\n"\
"                                                    //inputB[t] = inputB[
   t] / inputA[size*t];\n"\
"                          //*(inputA + size * t + globalIdx  +
   1)=*(inputM + size * (globalIdx + t + 1)+t);\n"\
"                }\n"\
"}\n";

const char *ProgramSource3 =
"__kernel void add3(__global float *inputM, __global float *
   inputA, __global float *inputB,const int size,const int t,
   const int MRg)\n"\
"{\n"\
"    int globalId = get_global_id(0);\n"\
"        if (globalId < MRg)\n"\
"        {\n"\
"                          inputB[globalId+1+t] -= inputM[size *
    t + globalId  + 1] * inputB[t];\n"\
"                          //inputB[t] = inputB[t] / inputA[size
   *t];\n"\
"                          //*(inputA + + size * t + globalId  +
    1) = *(inputM + size * t + globalId  + 1);\n"\
"    }\n"\

"}\n";

const char *ProgramSource4 =
```

```
"__kernel void add4(__global float *inputM, __global float *
   inputA, __global float *inputB,const int size,const int t,
   const int MRg)\n"\
"{\n"\
"    int globalId = get_global_id(0);\n"\
"        if (globalId == 1)\n"\
"        {\n"\
"                        inputB[t*globalId] = inputB[t*
   globalId] / inputA[size*t*globalId];\n"\
"    }\n"\

"}\n";

const char *ProgramSource5 =
"__kernel void add5(__global float *inputM, __global float *
   inputA, __global float *inputB,const int size,const int t,
   const int MRg)\n"\
"{\n"\
"    int globalId = get_global_id(0);\n"\
"        if (globalId < MRg)\n"\
"        {\n"\
"                        //inputB[globalId+1+t] -= inputM[size
    * t + globalId  + 1] * inputB[t];\n"\
"                        //inputB[t] = inputB[t] / inputA[size
   *t];\n"\
"                        *(inputA ++ size * t + globalId  +
   1) = *(inputM + size * t + globalId  + 1);\n"\
"    }\n"\
```

```
"}\n";

int main()
{
    int nra,nb,i,j;
    //float x;
    //float kj[120][120];

        //float **kj=malloc(1000*sizeof(int*));
    FILE *f1,*f2,*f4,*f5,*f6,*fp;
    system("cls");
        f1=fopen("in.txt","r");
    f2=fopen("out.txt","w");

        f4=fopen("rec.txt","r");
        f5=fopen("dis.txt","r");
        f6=fopen("flow.txt","w");
        fp=fopen("paraflow.txt","w");
    fscanf(f4,"%d",&nra);
        fscanf(f4,"%d",&nb);

    clock_t c1=clock();
        float **kj = (float **) calloc(nra+1,sizeof(float *))
           ;
        for(i=1;i<=nra;i++)
                kj[i] = (float *) calloc(nb+1,sizeof(float));
        clock_t c2=clock();
        double tc1=((c2-c1)*1000)/CLOCKS_PER_SEC;;
```

```
        //float **kjp = (float **) calloc(nra+1,sizeof(float
            *));
        //for(i=0;i<=nra;i++)
                //kjp[i] = (float *) calloc(nb+1,sizeof(float
                    ));

    printf("NRA=%d\n NB=%d\n",nra,nb);
        //fprintf(f2,"Global Stiffness matrix\n");
    for(i=1;i<=nra;i++)
    {
        for(j=1;j<=nb;j++)
                {
                        //kjp[i-1][j-1]=kj[i][j]=0;
                        kj[i][j]=0;
                        fscanf(f4,"%f",&kj[i][j]);
                        //kjp[i-1][j-1]=kj[i][j];
                        //fprintf(f2,"S%d%d=%f      ",i,j,kj[i
                            ][j]);
                }
                //fprintf(f2,"\n");
    }
        clock_t c3=clock();
        float * ac = (float *)malloc(sizeof(float)*(nra+1));
        clock_t c4=clock();
        double tc2=((c4-c3)*1000)/CLOCKS_PER_SEC;;
        //float * acp = (float *)malloc(sizeof(float)*(nra+1)
            );

    //fprintf(f2,"Modified AC\n");
```

```
    for(i=1;i<=nra;i++)
    {
                //acp[i-1]=ac[i]=0;
                ac[i]=0;
                fscanf(f5,"%f",&ac[i]);
                //acp[i-1]=ac[i];
                //fprintf(f2,"AC-%d=%f\n",i,ac[i]);
    }

        //cl starts


cl_context context;
cl_context_properties properties[3];
cl_kernel kernel,kernel2,kernel3,kernel4,kernel5;
cl_command_queue command_queue;
cl_program program,program2,program3,program4,program5;
cl_int err;
cl_uint num_of_platforms=0;
cl_platform_id platform_id;
cl_device_id device_id;
cl_uint num_of_devices=0;
cl_mem inputA, inputB,inputM;
cl_int ret;

size_t global,global2[2],local2[2];
int DATA_SIZE=nra*nb;
float *inputDataA= (float *)malloc(sizeof(float)*DATA_SIZE);
float *inputDataB= (float *)malloc(sizeof(float)*nra);
```

```
float *inputDataM= (float *)malloc(sizeof(float)*DATA_SIZE);

FILE *f31=fopen("out.csv","w");
//fprintf(f31,"\n");
//fprintf(f31,"Before kernel matrix,");
//fprintf(f31,"\n");
for(i=0;i<nra;i++)
    {
        for(j=0;j<nb;j++)
                {
                        inputDataA[j + nb*i ] = kj[i+1][j+1];
                        inputDataM[j+ nb*i ]=0;
                        //fprintf(f31,"%f,",inputDataA[j + nb
                            *i ]);

                }
            //fprintf(f31,"\n");
    }
/*
//fprintf(f31,"\n");
//fprintf(f31,"before ban KJ\n");
//fprintf(f31,"\n");
for(i=0;i<1*nra;i++)
    {
        for(j=0;j<1*nb;j++)
                {

                        fprintf(f31,"%f,",kj[i+1][j+1]);
```

```
                }
            fprintf(f31,"\n");
    }

        */
//fprintf(f2,"AC\n");
    for(i=1;i<=nra;i++)
    {

                inputDataB[i-1]=0;
                inputDataB[i-1] = ac[i];
                //fprintf(f2,"AC-%d=%f\n",i,inputDataB[i-1]);
    }


//new CL

        // retreive a list of platforms avaible
if (clGetPlatformIDs(1, &platform_id, &num_of_platforms)!=
   CL_SUCCESS)
{
printf("Unable to get platform_id\n");
return 1;
}

// try to get a supported GPU device
if (clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_CPU, 1, &
   device_id, &num_of_devices) != CL_SUCCESS)
{
```

```
printf("Unable to get device_id\n");
return 1;
}

// context properties list - must be terminated with 0
properties[0]= CL_CONTEXT_PLATFORM;
properties[1]= (cl_context_properties) platform_id;
properties[2]= 0;

// create a context with the GPU device
context = clCreateContext(properties,1,&device_id,NULL,NULL,&
    err);

// create command queue using the context and device
command_queue = clCreateCommandQueue(context, device_id, 0, &
    err);

// create a program from the kernel source code
program = clCreateProgramWithSource(context,1,(const char **)
    &ProgramSource, NULL, &err);

// compile the program
if (clBuildProgram(program, 0, NULL, NULL, NULL, NULL) !=
   CL_SUCCESS)
{
printf("Error building program\n");
return 1;
}
```

```
// specify which kernel from the program to execute
kernel = clCreateKernel(program, "add", &err);
clock_t g1=clock();
// create buffers for the input and ouput

inputA = clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(
    float) * DATA_SIZE, NULL, NULL);
inputM = clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(
    float) * DATA_SIZE, NULL, NULL);
inputB = clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(
    float) * nra, NULL, NULL);



// load data into the input buffer
clEnqueueWriteBuffer(command_queue, inputA, CL_TRUE, 0,
    sizeof(float) * DATA_SIZE, inputDataA, 0, NULL, NULL);
clEnqueueWriteBuffer(command_queue, inputM, CL_TRUE, 0,
    sizeof(float) * DATA_SIZE, inputDataM, 0, NULL, NULL);
clEnqueueWriteBuffer(command_queue, inputB, CL_TRUE, 0,
    sizeof(float) * nra, inputDataB, 0, NULL, NULL);
clock_t g2=clock();
        double tg1=((g2-g1)*1000)/CLOCKS_PER_SEC;;

 //part-2 starts
// create command queue using the context and device
command_queue = clCreateCommandQueue(context, device_id, 0, &
    err);
```

```
// create a program from the kernel source code
program2 = clCreateProgramWithSource(context,1,(const char
    **) &ProgramSource2, NULL, &err);

// compile the program
if (clBuildProgram(program2, 0, NULL, NULL, NULL, NULL) !=
    CL_SUCCESS)
{
printf("Error building program-2\n");
return 1;
}

// specify which kernel from the program to execute
kernel2 = clCreateKernel(program2, "add2", &err);

// part-2 ends

//part-3 starts
// create command queue using the context and device
command_queue = clCreateCommandQueue(context, device_id, 0, &
    err);

// create a program from the kernel source code
program3 = clCreateProgramWithSource(context,1,(const char
    **) &ProgramSource3, NULL, &err);

// compile the program
```

```
if (clBuildProgram(program3, 0, NULL, NULL, NULL, NULL) !=
    CL_SUCCESS)
{
printf("Error building program-3\n");
return 1;
}

// specify which kernel from the program to execute
kernel3 = clCreateKernel(program3, "add3", &err);

// part-3 ends

//part-4 starts
// create command queue using the context and device
command_queue = clCreateCommandQueue(context, device_id, 0, &
    err);

// create a program from the kernel source code
program4 = clCreateProgramWithSource(context,1,(const char
    **) &ProgramSource4, NULL, &err);

// compile the program
if (clBuildProgram(program4, 0, NULL, NULL, NULL, NULL) !=
    CL_SUCCESS)
{
printf("Error building program-4\n");
return 1;
}
```

```
// specify which kernel from the program to execute
kernel4 = clCreateKernel(program4, "add4", &err);

// part-4 ends

//part-5 starts
// create command queue using the context and device
command_queue = clCreateCommandQueue(context, device_id, 0, &
    err);

// create a program from the kernel source code
program5 = clCreateProgramWithSource(context,1,(const char
    **) &ProgramSource5, NULL, &err);

// compile the program
if (clBuildProgram(program5, 0, NULL, NULL, NULL, NULL) !=
    CL_SUCCESS)
{
printf("Error building program-5\n");
return 1;
}

// specify which kernel from the program to execute
kernel5 = clCreateKernel(program5, "add5", &err);

// part-5 ends
int size=nb;
```

```
size_t globalWorksizeFan1[1];
size_t globalWorksizeFan21[1];
    size_t globalWorksizeFan2[2];

        globalWorksizeFan1[0] = nb-1     ;
        globalWorksizeFan21[0] = 2;
        globalWorksizeFan2[0] = nb-1;
        globalWorksizeFan2[1] = nb-1;



int NLg,NMg,MRg,Ng,Lg,Kg,kg;
//float BN,C;
clock_t g3=clock();
NLg=(nra-1)-(nb-1)+1;
NMg=(nra-1)-1;
MRg=nb-1;

        for (int t=0; t<=(nra-2); t++)
        {
        if(t>NLg) MRg=nra -t;
                // kernel args
        cl_int argchk;
        argchk  = clSetKernelArg(kernel, 0, sizeof(cl_mem), (
           void *)&inputM);
        argchk |= clSetKernelArg(kernel, 1, sizeof(cl_mem), (
           void *)&inputA);
        argchk |= clSetKernelArg(kernel, 2, sizeof(cl_mem), (
           void *)&inputB);
```

```
argchk |= clSetKernelArg(kernel, 3, sizeof(int), (
    void *)&size);
argchk |= clSetKernelArg(kernel, 4, sizeof(int), (
    void *)&t);
        argchk |= clSetKernelArg(kernel, 5, sizeof(
            int), (void *)&MRg);




// launch kernel
clEnqueueNDRangeKernel(command_queue,kernel, 1, 0,
    globalWorksizeFan1,NULL,0, NULL, NULL);



        // kernel args
        argchk  = clSetKernelArg(kernel2, 0, sizeof(
            cl_mem), (void *)&inputM);
argchk |= clSetKernelArg(kernel2, 1, sizeof(cl_mem),
    (void *)&inputA);
argchk |= clSetKernelArg(kernel2, 2, sizeof(cl_mem),
    (void *)&inputB);
argchk |= clSetKernelArg(kernel2, 3, sizeof(int), (
    void *)&size);
argchk |= clSetKernelArg(kernel2, 4, sizeof(int), (
    void *)&t);
        argchk |= clSetKernelArg(kernel2, 5, sizeof(
            int), (void *)&MRg);
```

```
// launch kernel
clEnqueueNDRangeKernel(command_queue, kernel2, 2, 0,
    globalWorksizeFan2,NULL,0, NULL, NULL);


        argchk = clSetKernelArg(kernel3, 0, sizeof(
            cl_mem), (void *)&inputM);
argchk |= clSetKernelArg(kernel3, 1, sizeof(cl_mem),
    (void *)&inputA);
argchk |= clSetKernelArg(kernel3, 2, sizeof(cl_mem),
    (void *)&inputB);
argchk |= clSetKernelArg(kernel3, 3, sizeof(int), (
    void *)&size);
argchk |= clSetKernelArg(kernel3, 4, sizeof(int), (
    void *)&t);
        argchk |= clSetKernelArg(kernel3, 5, sizeof(
            int), (void *)&MRg);



// launch kernel
clEnqueueNDRangeKernel(command_queue,kernel3, 1, 0,
    globalWorksizeFan1,NULL,0, NULL, NULL);
```

```
        argchk  = clSetKernelArg(kernel4, 0, sizeof(
            cl_mem), (void *)&inputM);
argchk |= clSetKernelArg(kernel4, 1, sizeof(cl_mem),
    (void *)&inputA);
argchk |= clSetKernelArg(kernel4, 2, sizeof(cl_mem),
    (void *)&inputB);
argchk |= clSetKernelArg(kernel4, 3, sizeof(int), (
    void *)&size);
argchk |= clSetKernelArg(kernel4, 4, sizeof(int), (
    void *)&t);
        argchk |= clSetKernelArg(kernel4, 5, sizeof(
            int), (void *)&MRg);




// launch kernel
clEnqueueNDRangeKernel(command_queue,kernel4, 1, 0,
    globalWorksizeFan21,NULL,0, NULL, NULL);



        argchk  = clSetKernelArg(kernel5, 0, sizeof(
            cl_mem), (void *)&inputM);
argchk |= clSetKernelArg(kernel5, 1, sizeof(cl_mem),
    (void *)&inputA);
argchk |= clSetKernelArg(kernel5, 2, sizeof(cl_mem),
    (void *)&inputB);
argchk |= clSetKernelArg(kernel5, 3, sizeof(int), (
    void *)&size);
```

```
        argchk |= clSetKernelArg(kernel5, 4, sizeof(int), (
            void *)&t);
                argchk |= clSetKernelArg(kernel5, 5, sizeof(
                    int), (void *)&MRg);




        // launch kernel
        clEnqueueNDRangeKernel(command_queue,kernel5, 1, 0,
            globalWorksizeFan1,NULL,0, NULL, NULL);

    }

clock_t g4=clock();
        double tg2=((g4-g3)*1000)/CLOCKS_PER_SEC;;

clock_t g5=clock();
// copy the results from out of the output buffer
clEnqueueReadBuffer(command_queue, inputA, CL_TRUE, 0, sizeof
    (float) *DATA_SIZE, inputDataA, 0, NULL, NULL);
//clEnqueueReadBuffer(command_queue, inputM, CL_TRUE, 0,
    sizeof(float) *nra, inputDataM, 0, NULL, NULL);
clEnqueueReadBuffer(command_queue, inputB, CL_TRUE, 0, sizeof
    (float) *nra, inputDataB, 0, NULL, NULL);
clock_t g6=clock();
        double tg3=((g6-g5)*1000)/CLOCKS_PER_SEC;;
//fprintf(f31,"\n\nafterGPU\n");
/*for(i=0;i<1*nra;i++)
```

```
    {
        for(j=0;j<1*nb;j++)
                {

                        fprintf(f31,"%f,",inputDataA[j + nb*i
                            ]);

                }
            fprintf(f31,"\n");
    }
        */
//end new CL

int NL,NM,MR,N,L,K,k;
float BN,C;
clock_t c5=clock();
NL=nra-nb+1;
NM=nra-1;
MR=nb;

for(N=1;N<=NM;N++)
{
        BN=ac[N];
        ac[N]=BN/kj[N][1];
        if(N>NL) MR=nra-N+1;
        for(L=2;L<=MR;L++)
        {
                //if(kj[N][L]==0)continue;
                C=kj[N][L]/kj[N][1];
```

```
                    i=N + L -1;
                    j=0;
                    for(K=L;K<=MR;K++)
                    {
                            j=j+1;
                            kj[i][j]=kj[i][j]- C*kj[N][K];
                    }
                    ac[i]=ac[i]-C*BN;
                    kj[N][L]=C;
            }

}
clock_t c6=clock();
            double tc3=((c6-c5)*1000)/CLOCKS_PER_SEC;;
/*
fprintf(f31,"\n");
fprintf(f31,"after ban KJ\n");
fprintf(f31,"\n");
for(i=0;i<1*nra;i++)
    {
        for(j=0;j<1*nb;j++)
                    {

                            fprintf(f31,"%f,",kj[i+1][j+1]);

                    }
                fprintf(f31,"\n");
    }
```

```
        */
/*fprintf(f2,"Modified AC before backsub\n");
    for(i=1;i<=1*nra;i++)
    {
        fprintf(f2,"AC-%d=%f      ACP-%d=%f\n",i,ac[i],i,acp[i
            -1]);

    }

        /*
fprintf(f2,"Modified AC\n");
    for(i=1;i<=1*nra;i++)
    {
        fprintf(f2,"AC-%d=%f                GAC-%d=%f\n",i,ac[i],
            i,inputDataB[i-1]);
    }*/

clock_t c7=clock();

i=nra;
ac[nra]=ac[nra]/kj[nra][1];
for(N=1;N<=NM;N++)
{
        i=i-1;
        if(N<nb)
                MR=N+1;
        for(j=2;j<=MR;j++)
        {
                k=i+j-1;
```

```
                ac[i]=ac[i]- kj[i][j]*ac[k];
        }

}
clock_t c8=clock();
        double tc4=((c8-c7)*1000)/CLOCKS_PER_SEC;;

//GPU banfac
/*
int NLg,NMg,MRg,Ng,Lg,Kg,kg;
//float BN,C;
NLg=(nra-1)-(nb-1)+1;
NMg=(nra-1)-1;
MRg=nb-1;

for(Ng=0;Ng<=NMg;Ng++)
{
        BN=inputDataB[Ng];
        inputDataB[Ng]=BN/inputDataA[Ng*nb];
        if(Ng>NLg) MRg=nra -Ng;
        for(Lg=1;Lg<=MRg;Lg++)
        {
                if(inputDataA[Ng*nb + Lg]==0)continue;
                C=inputDataA[Ng*nb + Lg]/inputDataA[Ng*nb];
                i=Ng + Lg ;
                j=0;
                for(Kg=Lg;Kg<=MRg;Kg++)
                {
```

```
                        inputDataA[i*nb + j]=inputDataA[i*nb
                            + j]- C*inputDataA[Ng*nb + Kg];
                        j=j+1;
                }
                inputDataB[i]=inputDataB[i]-C*BN;
                inputDataA[Ng*nb + Lg]=C;
        }

}
 */

// cleanup - release OpenCL resources
clReleaseMemObject(inputA);
clReleaseMemObject(inputB);
clReleaseMemObject(inputM);
clReleaseProgram(program);
clReleaseKernel(kernel);
clReleaseProgram(program2);
clReleaseKernel(kernel2);
clReleaseProgram(program3);
clReleaseKernel(kernel3);
clReleaseProgram(program4);
clReleaseKernel(kernel4);
clReleaseProgram(program5);
clReleaseKernel(kernel5);
clReleaseCommandQueue(command_queue);
clReleaseContext(context);

//GPU Backsub
```

```
clock_t g7=clock();

i=nra-1;
inputDataB[nra-1]=inputDataB[nra-1]/inputDataA[(nra-1)*nb];
for(Ng=0;Ng<=NMg;Ng++)
{
        i=i-1;
        if(Ng<(nb-1))
                MRg=Ng+1;
        for(j=1;j<=MRg;j++)
        {
                k=i+j;
                inputDataB[i]=inputDataB[i]- inputDataA[i*nb
                    + j]*inputDataB[k];
        }
}

clock_t g8=clock();
        double tg4=((g8-g7)*1000)/CLOCKS_PER_SEC;;

// print the results




fprintf(f2,"Modified AC\n");
    for(i=1;i<=1*nra;i++)
    {
```

```
            //fprintf(f2,"AC-%d=%f                GAC-%d=%f\n",i,ac[i],
                i,inputDataB[i-1]);
        }
            fprintf(f2,"AC-%d=%f                GAC-%d=%f\n",nra,ac[
                nra],nra,inputDataB[nra-1]);

            fprintf(f2,"\nNRA=%d\n",nra);
            fprintf(f2,"\nNB=%d\n",nb);
            fprintf(f2,"\nGPU Execution time=%f\n",tg2+tg4);
            fprintf(f2,"\nGPU Communication time=%f\n",tg1+tg3);
            fprintf(f2,"\nCPU time=%f\n",tc1+tc2+tc3+tc4);
            fprintf(f2,"\nGPU time=%f\n",tg1+tg2+tg3+tg4);
}
```

Appendix C

List of Paper Published/Communicated

1 Application of Parallel Computing In Structural Engineering, National Conference on Emerging Trends in Technology Engineering & Management(NCEETM),Indus University, Ahmedabad

2 Heterogeneous Linear Equation Solver Using Graphics Processing Unit (GPU), Advances in Civil Engineering and Chemistry of Innovative Materials (ACECIM-14), SRM University, Chennai

3 Heterogeneous Linear Equation Solver Using Graphics Processing Unit (GPU And Central Processing Unit (CPU), International Civil Engineering Symposium (ICES-14),VIT University, Vellore

4 Accelerated Plane Frame Analysis Using Parallel Computing, Recent Advances In Civil And Structural Engineering (Racse-'14), ADIT and BVM

References

[1] E. D. Sotelino, Parallel Processing Techniques in Structural Engineering Applications, Journal of Structural Engineering, Vol. 129, No. 12, December 1, 2003,ASCE

[2] Jerome F. Hajjarz and John F. Abel, Parallel Processing Of Nonlinear Dynamic Analysis Of Steel Frame Structures Using Domain Decomposition, Proceedings of Ninth World Conference on Earthquake Engineering August 2-9, 1988, Tokyo-Kyoto, JAPAN (Vol.V)

[3] T. Bahcecioclu and O. Kurc, Nonlinear dynamic finite element analysis with GPU, 14th International Conference on Computing in Civil and Building Engineering. by International Society for Computing in Civil and Building Engineering,June-2012

[4] V. Kandasamy and M. Konig, Parallel finite element mesh generator using multiple GPUs, 14th International Conference on Computing in Civil and Building Engineering. by International Society for Computing in Civil and Building Engineering,June-2012

[5] Filip Kruzel and Krzysztof Banas,Vectorized OpenCL implementation of numerical integration for higher order finite elements, Computers and Mathematics with Applications, August-2013

[6] Depeng Yang, Junqing Sun, JunKu Lee, Getao Liang, David D. Jenkins, Gregory D. Peterson, and Husheng Li, Performance Comparison of Cholesky Decomposition on GPUs and FPGAs, 2012 Symposium on Application Accelerators in High-Performance Computing,National Center for Supercomputing Applications at the University of Illinois at Urbana-Champaign

[7] Lixiang Wang, Shihai Li, Guoxin Zhang, Zhaosong Ma and Lei Zhang, A GPU-based Parallel Procedure for Nonlinear Analysis of Complex Structures Using a Coupled FEM/DEM Approach, Mathematical Problems in Engineering, Volume 2013 (2013)

[8] Shang-Hsien Hsieh, Yuan-Sen Yang and Po-Yao Hsu , Integration of General Sparse Matrix and Parallel Computing Technologies for Large-Scale Structural

Analysis ,Computer-Aided Civil and Infrastructure Engineering 17, Blackwell Publishing, 350 Main Street, Malden, MA 02148, USA and 108 Cowley Road, Oxford OX4 1JF, UK.

[9] Xiao Qian, Wang Chengguo, Guo Ge, The Research of Parallel Computing for Large-scale Finite Element Model of WheelIlRail Rolling Contact,IEEE

[10] Xuanhua Fan, Rui-an Wu, Pu Chen, Zuogui Ning, Jian Li, Parallel Computing of Large Eigenvalue Problems for Engineering Structures, 2011 International Conference on Future Computer Sciences and Application,IEEE

[11] Chaojiang Fu, Parallel Computing For Finite Element Structural Analysis On Workstation Cluster, 2008 International Conference on Computer Science and Software Engineering, IEEE

[12] Chaojiang Fu, Parallel Computing For Finite Element Structural Modal Analysis On Workstation Cluster , International Conference on Information Science and Engineering (ICISE2009) , IEEE

[13] Yoon Kah Leow, Ali Akoglu, Ibrahim Guven, Erdogan Madenci, High Performance Linear Equation Solver Using NVIDIA GPUs, NASA/ESA Conference on Adaptive Hardware and Systems (AHS-2011)

[14] Girish Sharma , Abhishek Agarwala , Baidurya Bhattacharya ,A fast parallel Gauss Jordan algorithm for matrix inversion using CUDA, Computers and Structures 128 (2013) 3137

[15] Ravi Reddy,Alexey Lastovetsky,Pedro Alonso, Parallel Solvers for Dense Linear Systems for Heterogeneous Computational clusters, 978-1-4244-3750-4/09, 2009,IEEE

[16] T. P. Stefanski, S. Benkler, N. Chavannes, N. Kuster,Parallel Implementation of the Finite-Difference Time- Domain Method in Open Computing Language,978-1-4244-7368-7/10/2010,IEEE

[17] Jian-She Wang, Nathan Ida, Parallel Algorithms For Direct Solution Of Large Systems Of Equations, CH2649-2/89/0000/0231 1988, IEEE

[18] V. Mani, B. Dattaguru, N. Balakrishnan and T.S. Ramamurthy, Parallel Gaussian Elimination For Banded Matrix, A Computational Model, Conference on Computer and Communication Systems, September 1990, Hong Kong, IEEE

[19] S.F.McGinn, R.E.Shaw, Parallel Gaussian Elimination Using OpenMP and MPI, Proceedings of the 16th Annual International Symposium on High Performance Computing Systems and Applications (HPCS.02) 0-7695-1626-2/02 2002, IEEE

[20] Hang Liu, Jung-Hee Seo, Rajat Mittal, H. Howie Huang, GPU-Accelerated Scalable Solver for Banded Linear Systems, 978-1-4799-0898-1/13/ 2013, IEEE

[21] Zhihui Zhang, Qinghai Miao, Ying Wang, CUDA-Based Jacobi's Iterative Method, 978-0-7695-3930-0/09 2009, IEEE

[22] Parallel computing - Wikipedia `http://en.wikipedia.org/wiki/Parallel_computing`

[23] Graphics processing unit - Wikipedia `http://en.wikipedia.org/wiki/Graphics_processing_unit`

[24] Introduction to OpenCL Programming-Training Guide - AMD

[25] General-purpose computing on graphics processing units - Wikipedia `http://en.wikipedia.org/wiki/General-purpose_computing_on_graphics_processing_units`

[26] OpenCL Specifications by `http://www.khronos.org`

[27] Heterogeneous Computing with OpenCL by Benedict Gaster, Lee Howes, David R. Kaeli, Perhaad Mistry, Dana Schaa

[28] OpenCL Zone - AMD Developer Central, `http://developer.amd.com/resources/heterogeneous-computing/opencl-zone`

[29] Earth Simulator - Wikipedia `http://en.wikipedia.org/wiki/Earth_Simulator`

[30] Blue Gene - Wikipedia `http://en.wikipedia.org/wiki/Blue_Gene`

[31] Nvidia Tesla - Wikipedia `http://en.wikipedia.org/wiki/Nvidia_Tesla`

[32] Supercomputer - Wikipedia `http://en.wikipedia.org/wiki/Supercomputer`

[33] Introduction to High Performance Computing, M. D. Jones, Ph.D., Center for Computational Research University at Buffalo State University of New York

[34] Computer cluster - Wikipedi `http://en.wikipedia.org/wiki/Computer_cluster`

[35] Numerical Methods in Science and Engineering, By S. Rajasekaran