SOLUTION OF SPARSE SYSTEMS ON GPU ARCHITECTURE

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

ANDAÇ LÜLEÇ

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
CIVIL ENGINEERING

JUNE 2011

Approval of the thesis:

SOLUTION OF SPARSE SYSTEMS ON GPU ARCHITECTURE

Submitted by **ANDAÇ LÜLEÇ** in partial fulfillment of the requirements for the degree of **Master of Science in Civil Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen _____
Dean, Graduate School of **Natural and Applied Sciences**

Prof. Dr. Güney Özcebe _____
Head of Department, **Civil Engineering**

Asst. Prof. Dr. Özgür Kurç _____
Supervisor, **Civil Engineering Dept., METU**

Examining Committee Members:

Asst. Prof. Dr. Afşin Sarıtaş _____
Civil Engineering Dept., METU

Asst. Prof. Dr. Özgür Kurç _____
Civil Engineering Dept., METU

Asst. Prof. Dr. Ayşegül Askan Gündoğan _____
Civil Engineering Dept., METU

M.Sc Onur Pekcan _____
Civil Engineering Dept., METU

Asst. Prof. Dr. Alptekin Temizel _____
Informatics Institute, METU

Date: Jun 24, 2011

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name: Andaç LÜLEÇ

Signature

ABSTRACT

SOLUTION OF SPARSE SYSTEMS ON GPU ARCHITECTURE

Lüleç, Andaç
M.Sc., Department of Civil Engineering
Supervisor: Asst. Prof. Dr. Özgür Kurç

June 2011, 93 pages

The solution of the linear system of equations is one of the core aspects of Finite Element Analysis (FEA) software. Since large amount of arithmetic operations are required for the solution of the system obtained by FEA, the influence of the solution of linear equations on the performance of the software is very significant.

In recent years, the increasing demand for performance in the game industry caused significant improvements on the performances of Graphical Processing Units (GPU). With their massive floating point operations capability, they became attractive sources of performance for the general purpose programmers. Because of this reason, GPUs are chosen as the target hardware to develop an efficient parallel direct solver for the solution of the linear equations obtained from FEA.

Keywords: GPGPU, Sparse Solver, multifrontal, multiple front

ÖZ

SEYREK SİSTEMLERİN GPU KULLANILARAK ÇÖZÜMLENMESİ

Lüleç, Andaç
Yüksek Lisans, İnşaat Mühendisliği Bölümü
Tez Yöneticisi: Yrd. Doç. Dr. Özgür Kurç

Haziran 2011, 93 sayfa

Doğrusal denklemlerin çözümü, sonlu elemanlar metodunun kullanıldığı analizlerin önemli bir bölümünü oluşturmaktadır. Bu bölümde yapılan aritmetik işlemlerin çokluğu sebebiyle, doğrusal denklemlerin çözüm başarımı, sonlu elemanlar metodunu kullanan yazılımların toplam başarımını önemli ölçüde etkilemektedir.

Son yıllarda bilgisayar oyun endüstrisinin gelişmesi, ekran kartlarının hesaplama güçlerinde önemli bir yükselişe sebep olmuştur. Ekran kartlarında bulunan işlemcilerin sahip oldukları bu güç, günümüzde genel amaçlı program tasarlayıcıların da dikkatini çekmektedir. Bu sebeple sonlu elemanlar metodunun kullanıldığı analizlerle elde edilen doğrusal denklemlerin çözülebilmesi amacıyla, bu çalışmada ekran kartlarındaki işlemciler ile çalışan aşamalı ve çoklu aşamalı çözüm algoritmalarının kullanıldığı seyrek çözücüler geliştirilmiştir.

Anahtar Kelimeler: GPGPU, Seyrek Çözücü, Çoklu Aşamalı Çözüm, Aşamalı Çözüm

To My Family

ACKNOWLEDGMENT

I wish to express my deepest gratitude to my supervisor Asst. Prof. Dr. Özgür Kurç for his guidance, encouragement, criticism, support and patience during this research.

I would like to thank my dear roommates Alper Aldemir, İsmail Ozan Demirel, Emre Özkök, Uğur Akpınar, Emrah Erşan Erdoğan, Taylan Solmaz and Efe Gökçe Kurt who make this research much more enjoyable with their friendship.

I would like to thank my colleagues Tunç Bahçecioğlu and Semih Özmen for their support and inspiring discussions during this study.

I would like to thank my dearest friend Dinçay Akçören, for his companionship during writing this work.

I would like to thank my beloved Zeynep, for her patience for my absence while I am working on this study and her love which gave me strength to finish this dissertation.

I wish to thank my family, especially my mother, Neşe Lüleç. Without their encouragement and support this study could not be completed.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURES

LIST OF SYMBOLS/ACRONYMS

ALU: Arithmetic Logical Unit

COO: Coordinate List sparse matrix storage algorithm

CPU: Central Processing Unit

CSC: Compressed Sparse Column sparse matrix storage algorithm

CSR: Compressed Sparse Row sparse matrix storage algorithm

CUDA: Compute Unified Device Architecture

D: Diagonal matrix obtained from LDLT decomposition

DirectCompute: Microsoft Direct Compute

DOF: Degree of Freedom

FEA: Finite Element Analysis

FPU: Floating Point Unit

GPGPU: General Purpose computing on Graphics Processing Units

GPU: Graphical Processing Unit

L: Unit lower triangular matrix

MF: Multifrontal Algorithm

MF2: Multifrontal Algorithm with assembly tree formation algorithm

MPF: Multiple Front Algorithm

MPI: Message Passing Interface

MPICH2: Message Passing Interface Chameleon 2

MUMPS: a Multifrontal Massively Parallel sparse direct Solver

OpenCL: Open Computing Language

OpenMP: Open Multi Programming

PC: Personal Computer

PVM: Parallel Virtual Machine

RAM: Random Access Memory

SFU: Special Function Units

SM: Streaming Multiprocessor

SP: Streaming Processor

SPA: Streaming Processor Array

SPOOLES: SParse Object Oriented Linear Equations Solver

TPC: Texture/Processor Cluster

WSMP: Watson Sparse Matrix Package

[] :Matrix

{} :Vector

n :number of degree of freedoms

nrhs :number of right hand side vectors

[K]: stiffness matrix

{d}: displacment vector

{F}: Force vector

CHAPTER 1

INTRODUCTION

1.1 Problem Definition

In recent years, production of multi core processors caused a significant change in the software development paradigm. Nowadays, almost every personal computer has multi-core processors which offer users a significant increase in computational power. The benefits of this increase took effect not only in our daily lives but also in science and engineering. The computationally hard problems can be solved easily and the computation time decreased significantly. Since CPU technology is limited with temperature and it is hard to increase transistors in one core, processor manufacturers are planning to increase the number of cores to satisfy the demand for performance. This fact challenges programmers to move towards multi-core programming.

As the computational power increases with the use of multi core processors in the central processing units (CPU), the recent improvements in game industry enforced graphical processing units (GPU) designed to complete massive floating operations simultaneously for a video frame in a game. Today, most of the modern graphic cards are manufactured with the hundreds of processing units on them. Since the GPUs have such a computational power with cheaper prices relative to multi-core CPUs, they have become attractive sources of high performance for not only graphical computations but also software used for solution of problems in mechanics, fluid dynamics, finance and etc. According to the recent developments, with the

concept of general purpose computing on graphics processing units (GPGPU), programmers started to use GPUs to handle problems requiring the large number of floating point operations such as solution of linear systems.

Solution of large number of linear equations plays a major role in finite element analysis (FEA) software. Since large amount of floating point operations must be completed, solution of linear equations is a significant factor affecting the performance of the FEA software. Although there are high performance solvers available, most of these solvers were developed for parallel CPU architectures. Since GPUs offer developers higher performance with cheaper prices and ease to manufacture/assembly of the system, use of GPUs in the solution of sparse systems seems to be profitable by means of performance and budget. Because of this reason, in this study, a parallel general purpose direct sparse solver running on GPU was implemented to provide the mentioned advantages of GPU architecture and increase the performance of the FEA software.

1.2 Background

1.2.1 Parallel Architectures

The word “parallelism”, meaning use of more than one processors to deal with a single problem, was first used by two IBM researchers, John Cocke and Daniel Slotnick in 1958 [1-2]. For 30 years, after the first personal computer (PC) was developed, a lot of improvements were achieved in the technologies for PCs and workstations. Most of the desktop computers have CPUs with clock speeds changing between 1 GHz and 4 GHz, running approximately 1000 times faster their 30 year old ancestor [3]. These improvements in processor speeds, however, were able to continue until 2003. Since then, due to high energy consumption and heat-dissipation problems, processor manufacturers has changed their models to ones those include

more than one processing units, referred to as processor cores [4-5]. Although it was expected that this change would decrease the execution time of programs, the performance of applications was below expectations, since these programs were sequential. On the contrary, parallel programs benefit from the improvement about multi processors. This caused a trend about parallel program development and referred to as concurrency revolution [4-5]. Although parallel programming is not a new term and applications have been written as parallel codes for a long time, the use of such applications was not very common. These programs were run on expensive, large scale computers. With the recent improvements, every microprocessor became parallel computers now and every programmer can benefit from the performance of multi core processors running concurrently [4].

The demand for performance of CPU has been satisfied with the multi core CPUs recently [6-7]. Moreover the semiconductor industry decided on two main manufacturing options since 2003 [4, 8] i.e. multi core and many core processors. All processors in such systems use a single random access memory (RAM), this systems are classified as shared memory architecture. Intel Core i7 microprocessors are good examples of multi-core machines designed for increasing the performance of sequential codes. Rather, the many core devices such as recent GPUs were designed for increasing the performance of parallel codes. By 2009, while the peak performance of an Intel Quad-Core CPU for floating point operations was around 100 gigaflops, computing speed of 1 teraflops was reached by an AMD GPU [4]. The difference between the peak performances was mainly due to the difference in the design architectures. Since CPUs are designed for parallel execution of sequential codes, their control units and cache memories cover larger spaces on the chip than the arithmetic logical units (ALU) where the floating point operations are computed. Because of this reason, GPUs were designed to satisfy the demand for massive floating point operations in a video game frame, while space of the control units and cache memories was decreased; the number of ALUs was increased [4].

Besides the multi-core or many-core architecture, there is also another parallel architecture called distributed memory architecture. In this architecture two or more processors having their own memories are connected to each other by a network. The processors may be CPU, GPU or both. The fastest computers in the world are distributed memory architectures [9]; they are not easily manufactured and used by every programmer who wants performance for the applications used in their daily life. Today, this demand is satisfied by the dual core or quad core CPUs even by GPUs in the desktop computers and laptops.

In order to benefit from the computational powers of the parallel architectures, applications must be developed with suitable parallel computing technique. For shared memory architectures there are mainly two programming approaches. In the first one, threads were created as execution units, and executed by cores or processors. In this approach since whole system uses single RAM and can be accessed by only one processor at a time, writing to and reading from the memory has to be sequential. Consequently concepts like race conditions, deadlocks, synchronization, etc. compromises from use of multiple sources at a time [10]. OpenMP (Open Multi Processing [11]) is an example for shared memory programming methods for controlling and executing threads for C++ and FORTRAN languages. In a different approach instead of creating multiple threads at computers, multiple processes created and data is transferred from one process to another by message passing libraries. MPI (Message Passing Interface [12]) is a library specification for message passing. MPICH2 (Message Passing Interface Chameleon 2 Library [13]) is a widely portable implementation of MPI standard to support different computation and communication platforms such as commodity clusters, high-speed networks and proprietary high-end computing systems. Besides MPI, another standard is also available for message passing in distributed memory architectures, which is PVM (Parallel Virtual Machine [14]). The differences between two standards were discussed by Gropp et al. [15].

Since graphical cards have their own memory and processors, programming methods for GPUs differ from the methods of other architectures. The most common methods for programming GPUs are CUDA (Compute Unified Device Architecture [17, 18]) and OpenCL (Open Computing Language [19]). Karimi et al. [20] compared these two methods; and concluded that CUDA had better performance than OpenCL. Besides CUDA and OpenCL, another method for GPU programming is DirectCompute (Microsoft Direct Compute [21]). CUDA can be used for only NVidia graphical cards, whereas DirectCompute and OpenCL can be used for other graphical cards too. Although DirectCompute and OpenCL are better choices from portability point of view, CUDA has better performance; it is easier to developing a program with it, used by the majority and is updated frequently.

1.2.2 Solution of Linear Equations

The system of linear equations obtained from FEA can be expressed as:

$$[K]\{d\} = \{F\} \tag{1.1}$$

In Equation 1.1, for a system with *n* degrees of freedom (DOF), K is the *n* by *n* stiffness matrix; d and F are the *n* by 1 sized displacement vector and *n* by 1 force vector respectively. If there are more than one loading conditions, the displacement vector d and force vector F become *n* by *nrhs* matrices where *nrhs* is the number of loading conditions.

Iterative and the direct methods are the two main methods for the solution of linear system of equations. Iterative methods solve the system of equations with trial and error approach. They are based on converging correct solution by iterations after starting from an initial guess. Since they are scalable and needs less storage, they are appropriate for solution of large problems. On the other side, iterative methods may be inefficient for multiple loading cases, since they have to start the solution over for

each right hand side vector. In addition, problem dependent preconditioning techniques must be used to reduce the number of iteration and these techniques affect the convergence and solution time of iterative methods. Moreover, since the solution is obtained in an iterative manner, the solution time cannot be estimated [22, 23].

In direct methods the coefficient matrix, which is K in FEA, is first factorized with LDL^T, LU or Cholesky Factorization and then the solution is obtained by forward and backward substitutions. Although, such methods need larger memory than iterative methods, they are efficient for solution of multiple loading conditions. Second, execution time for exact solution is predictable. Moreover, direct solvers can manage numerical challenges such as nearly-singular matrices while iterative solvers inefficient. Because of these reasons, direct solvers are used by most of the FEA software [22, 23].

1.2.3 Sparsity and Parallel Sparse Solutions

The systems of linear equations obtained from FEA are generally sparse which have large number of zero terms and this fact can be used for decreasing the required memory and number of operations. Because of this reason different storage formats and solution algorithms are used for sparse systems. The common storage formats are Coordinate List (COO), Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC). In COO format, non-zero values, their row and column indices stored and this format is efficient for modifications. CSC and CSR formats are efficient for matrix vector operations and they are the most commonly used sparse formats. In these formats non-zero values, cumulative ID of the first non-zero value in column or row and the row or column addresses are stored. An example for CSC format is presented in Figure 1.1.

$$A = \begin{bmatrix} 1.25 & 0 & 0.87 & 0 & 0 \\ & 2.46 & 0 & 0.94 & 0.36 \\ & & 1.64 & 0.05 & 0 \\ & \textit{Sym.} & & 2.48 & 1.90 \\ & & & & 3.88 \end{bmatrix}$$

$$\text{val\_A} = \{1.25, 2.46, 0.87, 0.94, 0.05, 2.48, 0.36, 1.90, 3.88\}$$
$$\text{row\_ind} = \{0, 1, 0, 2, 1, 2, 3, 0, 1, 3, 4\}$$
$$\text{col\_ptr} = \{0, 1, 2, 4, 7, 10\}$$

Figure 1.1: CSC Format for a 5×5 matrix

In Figure 1.1, an example for CSC format is given for a 5×5 matrix; three arrays are stored which are val_A, row_ind and col_ptr. val_A stores the value of the non-zero terms, row_ind stores the row addresses of the non-zero term and col_ptr stores ID of first non-zero term in that column.

There are two types of parallel solution algorithms, these are iterative and direct methods. As it was mentioned before, although iterative methods require less memory, they are not as robust as direct methods and they need preconditioning techniques. Global-Subdomain Implementation (GSI), Primal Subdomain Implementation (PSI) and the FETI method [24] can be given as examples of iterative methods. Bitzarakis et al. [25] discussed about these three methods and concluded that the FETI method was better for the solution of large systems because of its numerical stability and less sensitivity to the quality of preconditioning.

Direct methods for sparse matrix factorization were classified into three groups, left looking (fan-in), right looking (fan-out) and multifrontal methods by Duff and van der Vorst [26,27]. In left looking algorithm (fan-in), first, update of column is

completed by using previous columns then factorization of that column is completed and the data on the left is accessed. SPOOLES (SParse Object Oriented Linear Equations Solver [28, 29]) and SuperLU [30, 31] are two examples for sparse solvers using left-looking algorithm. On the other hand, in right looking algorithm (fan-out), first, factorization of the column is completed and then update of the following columns is completed and the data on the right is accessed. Oblio [32] is an example for sparse solvers using right-looking algorithm.

The third method for direct sparse factorization is multifrontal method. The frontal method [33], is based on assemble and solution of a dense matrix called frontal matrix. In the parallel version of the frontal method, namely multifrontal method [34] several frontal matrices are assembled and solved simultaneously. The main advantage of multifrontal methods is utilizing highly optimized dense linear algebra routines during solution. This way the requirement of indirect addressing for sparse matrices is eliminated. WSMP (Watson Sparse Matrix Package [35, 36]) and MUMPS (a Multifrontal Massively Parallel sparse direct Solver [37, 38]), are two examples for commonly used parallel multifrontal sparse solvers.

As sparse solvers use different factorization algorithms, they are also developed as different platforms. While CHOLMOD [39, 40], Oblio, UMFPACK [41, 42] and SuperLU were developed as serial platforms, MUMPS, SPOOLES and WSMP were designed as parallel platforms. There are several studies where the performances of the solvers are compared in the literature. The serial performances of the solvers were compared by Gould et al. [43]. According to this study CHOLMOD had best performance among the other solvers. But when the parallel performances of the solvers were tested by Gupta and Muliadi [44], it was concluded that MUMPS and WSMP had better performances due to use of optimized dense solver routines.

1.2.4 Sparse Solvers running on GPU

As a massively parallel architecture GPU, offers high performance for applications requiring large number of floating point operations such as solution of linear sparse systems. While there are numerous CPU implementations of parallel sparse solvers, the situation is not pretty same for GPU implementations. Moreover, most of the implementations are not direct solvers.

Bolz et al. [45] implemented two iterative sparse solvers running on GPU, a sparse matrix conjugate gradient solver and a regular-grid multigrid solver. They reached better performances with GPU than CPU implementation in conjugate gradient solver. Krüger and Westermann [46] implemented linear algebra operators for solution of equations. They focused on developing matrix and vector layouts for efficient matrix-vector and vector-vector operations in the implementation of iterative methods such as conjugate gradient and Gauss-Seidel. Buatois et al. [47] implemented an optimized linear sparse solver running on GPU. An iterative method, preconditioned conjugate gradient algorithm with an optimizer was preferred in this study due to ease of parallelization of iterative methods than direct methods. The results obtained GPU implementation had a better performance than the high performance CPU functions. Couturier and Domas implemented generalized minimal residual algorithm which is also an iterative method, obtaining speedups ranging from 8 up to 23 for solution of sparse systems [48]. Lucas et al. [49] implemented a multifrontal solver on GPUs. In this study the workload is distributed to multi core CPU and GPU. While the factorization of smaller matrices was completed on CPU, the larger dense matrices were factorized by GPUs. In GPU, the factorization process was completed 5.91 times faster than the CPU using single core and it was 1.34 times faster than the CPU using the 8 cores.

1.3 Objective & Scope

The main goal of this study is to develop a high performance direct sparse solver running on GPU for FEA. Moreover, it is aimed in supreme level to propose an alternative way of efficient parallel solution of linear equations with GPU which is a cheaper and more portable hardware.

Hence, the objective of this study can be summarized as:

- Developing a multiple front sparse solver and a multifrontal sparse solver both running on GPU for FEA. Moreover, investigating limits of the solver caused by the GPU hardware and the variables influencing the performance of the solver such as number and size of the substructures.

- Implementing parallel algorithms for sparse system condensation and frontal matrix assembly algorithms on GPU for multiple front and multifrontal methods. Furthermore, developing the subroutines of these algorithms in a way that allowing to be used separately as parts of different algorithms and heterogeneous architectures.

- Developing a parallel dense system condensation algorithm on GPU allowing condensation of multiple frontal matrices at the same time for multifrontal method.

In this study, GPU implementations of multiple front and multifrontal sparse solvers were developed. A sparse solver requires additional algorithms such as work balancing and ordering algorithms, besides the matrix factorization and solution algorithms, however, these additional algorithms were not considered. Thus, this

study mainly focused on the matrix factorization and solution algorithms which are condensation of sparse matrices, assembling of frontal matrices, condensation of frontal matrices, solution of frontal matrices and finally solution of the system. Consequently, the algorithm takes preordered substructure stiffness matrices, load vectors and assembly tree of the system as input data and gives the displacement vector as output.

Since the GPU memory is limited and very small compared to main storage units (hard disk), it was preferred using memory for testing larger system of equations, rather storing multiple loading conditions. Because of this reason, the algorithms were designed for single loading case. However, they can be adopted for multiple loading conditions with slight modifications. All functions used in the solver were decided to be developed for GPUs rather than using CPU implementations. There are two reasons for this decision; first, it is preferred that functions to be used separately as parts of heterogeneous solution procedures like completion of one task in GPU, another task in CPU, second, to minimize overhead caused by data transfer between GPU and host machine.

For GPU hardware, NVidia graphical cards were used and GPU kernels were developed in CUDA. Although both NVidia and AMD support OpenCL, CUDA is more common and more frequently updated improving the performance. The test problems were selected from the structures composed of 2D elements. Structures with different sizes were tested for different number of substructures. The performances of kernels were investigated.

1.4 Thesis Organization

The rest of the thesis is organized as follows: In Chapter 2, general information about the GPU hardware was given and the graphical cards used in this study as GPU hardware were introduced. General concepts about multiple frontal method and its GPU implementation, also test problems and the performance of the method can be found in Chapter 3, In the following chapter, multifrontal method and its GPU implementation were explained. The performance of the method was included in this chapter too. In the final chapter the conclusion of this study and the future work was presented.

CHAPTER 2

GPU HARDWARE & GPGPU

2.1 Introduction

GPU hardware was selected as target architecture for their portability and lower price than other parallel architectures. While they have much more processing cores than multi core CPUs, the architecture of processing cores of GPU is different from the CPUs. As CPUs are much more efficient for parallel execution of sequential programs, with many processing units GPU becomes an attractive source of high performance for massive floating number operations. The information about the processors, their components and the properties of the storage devices on the graphical cards were presented in the first part of this chapter. Besides, the information about the general GPU architecture, the hardware properties and limitations of the graphic cards, used in this study, are also introduced at this section of the chapter.

As developing parallel programs, different aspects of the problem should be considered. While, since only one processor was used in sequential coding, completion of operations and access to memory are both sequential, but in parallel programming sequence of memory access and operations of each thread should be considered carefully. In addition to the circumstances caused by parallel programming, the difference in the architecture requires use of CUDA specific functions for developing codes running on GPU. Because of these reasons it is convenient to give information about general concepts of GPGPU. As a result, in the

second section of the chapter general information about some important built-in CUDA functions, concurrent execution of threads, communication between them, data transfer between GPU and CPU, and type of kernels such as host, device and global kernels was presented.

2.2 Architecture of a GPU

It is more appropriate to explain parallel GPU architectures in several levels for the sake of clarity. The first level is formed by streaming processors (SP) [50]; these processors are the smallest processing particles on the GPU. In Figure 2.1 architecture of a SP is illustrated.

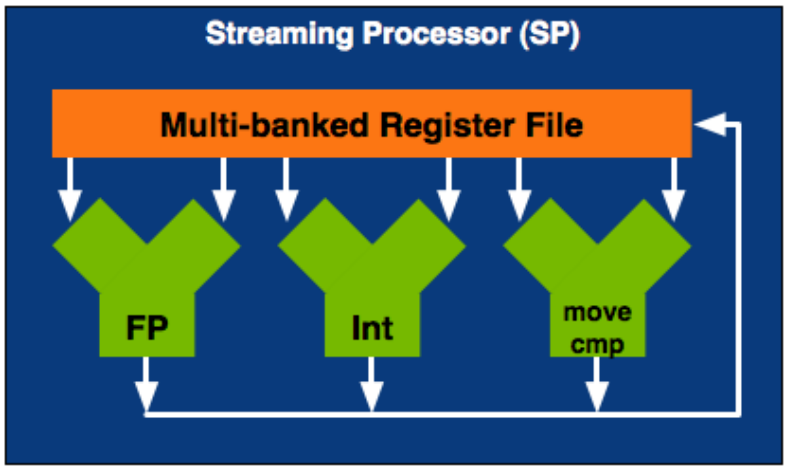


Figure 2.1: Architecture of a streaming processor (SP) [50]

SPs include two arithmetic logic units (ALU) and a floating-point unit (FPU). The ALUs are responsible for integer operations and logical comparisons and they are denoted as “Int” and “move cmp” in the figure respectively. On the other hand, a FPU is responsible for floating point operations and denoted as “FP” in the figure. As shown in the Figure 2.1, streaming processors do not have any cache memory, because of this reason a single streaming processor can only be used for arithmetic

operations. That is why several streaming processors were gathered and form streaming multiprocessors for performing numerical computations in parallel.

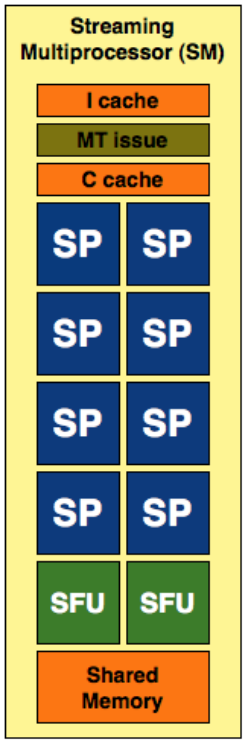


Figure 2.2: Architecture of a streaming multiprocessor (SM) [50]

In Figure 2.2, a streaming multiprocessor (SM) was illustrated. A SM can be defined as array of SPs [50], Most of the GPUs have eight SPs in a SM. A SM also includes two special processors called Special Function Units (SFUs). SFUs are responsible for special functions such as sin and cosine. The MT issue unit distributes the instructions to all SPs and SFUs in the block. Moreover, there are a small instruction cache (I cache), a read only data cache (C cache) and a 16 KB low latency read/write shared memory in a SM. The cache memories are kept very small to increase the number of SMs on the chip. Since the datasets dealt by the GPU are very small compared to CPU, by decreasing the cache memory and increasing the number of SMs, additional performance was obtained for a small sacrifice.

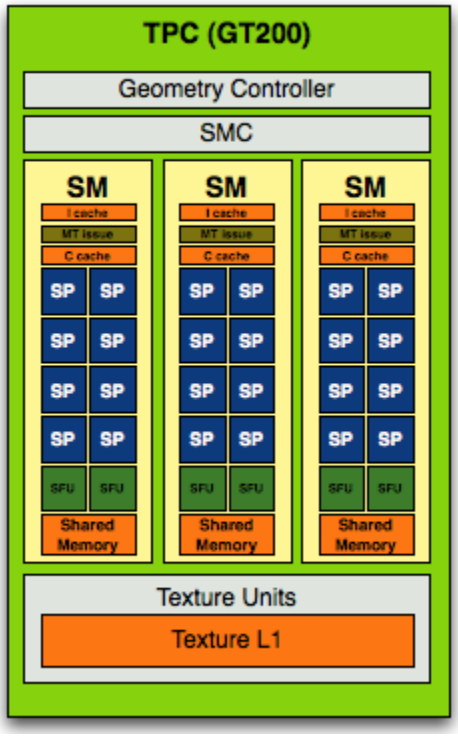


Figure 2.3: Architecture of a texture/processor cluster (TPC) [50]

At a higher level, multiple SMs form texture/processor clusters (TPC) [50]. In Figure 2.3 a TPC was shown. Different architectures may have different number of SMs on TPCs, for example, while GT200 family includes three SMs on a single TPC; G80 family includes two SMs on a single TPC. In addition to SMs, there are one control logic unit and one texture unit, which include a L1 texture cache for graphical operations, on a single TPC. While number of the SMs on a single TPC may vary, the components of the TPCs are same for all of the production families.

Streaming processor arrays (SPAs) are formed of TPCs. SPAs do not include any other components than TPCs. The number of TPCs can vary from one production family to another. This situation provides the modularity of the NVidia graphical cards, changing just the number of the components yields performance differences between the products.

At the end, with the composition of SPAs and some other components, the end product, a GPU from GT200 series was obtained and illustrated in Figure 2.4. Instructions from the CPU and the data in the system memory are transferred to GPU via PCIe bus. Below the PCIe Interface, there are schedulers and control logic to distribute workloads to the TPCs. Processing cores are in the middle. At the lower part there are L2 texture caches and raster operation processors (ROPs) for final filtering and output of the data. Last, there is dynamically random access memory (DRAM), referred as global memory of the GPU with a higher latency than shared memory or cache memories.

In this study three different GPUs were used. These are:

- GeForce GTX 275: It has 30 SMs with totally 240 SPs. Maximum amount of shared memory for a single SM is 16 KB and 1024 threads can be created for a single SM [18]. Its memory bandwidth is 127 GB/sec and capable of 1010.88 GFLOPS [51].

- GeForce GTX 580 Amp: It has 16 SMs with totally 512 SPs. Maximum amount of shared memory for a single SM is 48 KB and 1536 threads can be created for a single SM [18]. Its memory bandwidth is 192 GB/sec and capable of 1581.06 GFLOPS [52].

- Tesla C2050: It has 14 SMs with totally 448 SPs. Maximum amount of shared memory for a single SM is 48 KB and 1536 threads can be created for a single SM [18]. Its memory bandwidth is 144 GB/sec and capable of 1030 GFLOPS [53].

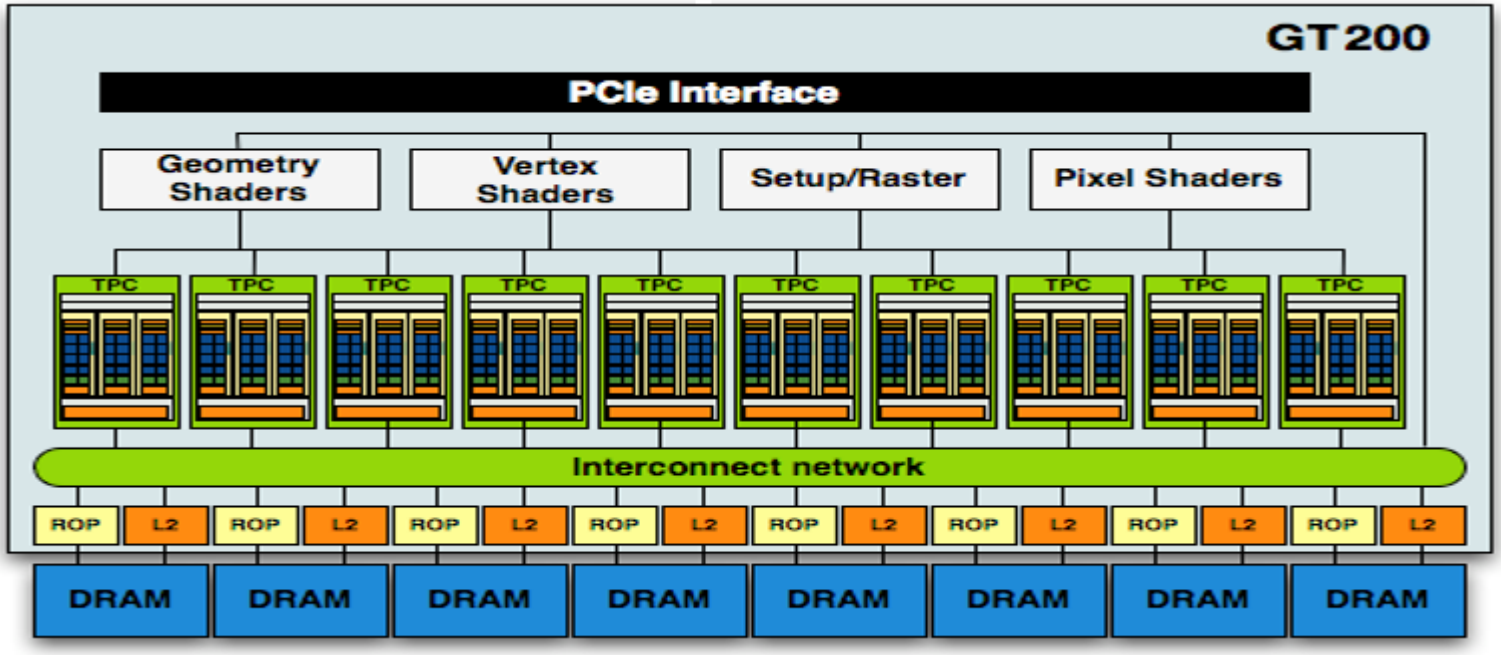


Figure 2.4 Architecture of a GPU from GT200 series [50]

2.3 GPGPU

As GPUs are becoming very attractive sources of high performance in general purpose programming with many cores on it, developing a code running on GPU requires additional information according to the sequential coding in CPU. Parallelism in GPGPU is based on concurrent execution of a kernel, which is parallel code portion executed on GPU, by threads [54]. Generally a GPU program have some portions to be run on “host”, referred as CPU, and some kernels to be run on “device”, referred as GPU. Although, a thread can execute only one kernel at a time, the execution of the same kernel by many threads provides high performance. Since every thread has an ID, each thread can compute different memory addresses and can make different control decisions. This situation transforms the kernel, executed by all threads, unique for each thread.

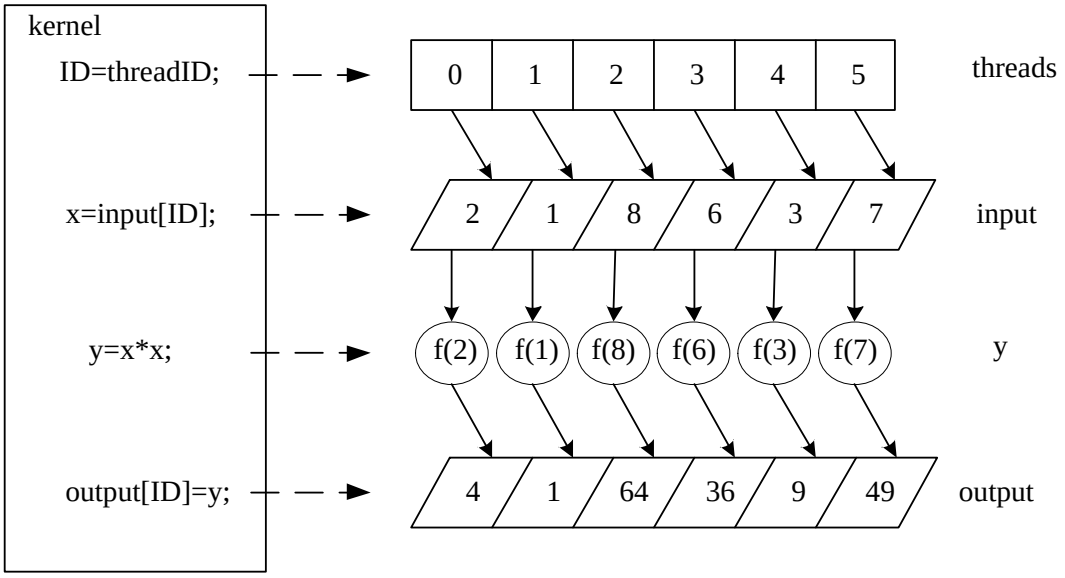


Figure 2.5: Execution of an example kernel by multiple threads

In Figure 2.5 execution of a simple kernel by multiple threads was illustrated. In this figure, with the execution of the kernel, a thread takes the corresponding value on the

input array according to its ID and assigns it to variable x, consequently, x value for each thread is different. Then, y value is calculated by multiplication of x by itself for every thread. Finally, y value is written to corresponding place on the output array. In this simple example six arithmetic operations are completed during the calculation of y at a time. For the sequential implementation of this code, one operation cycle would be needed for each arithmetic operation.

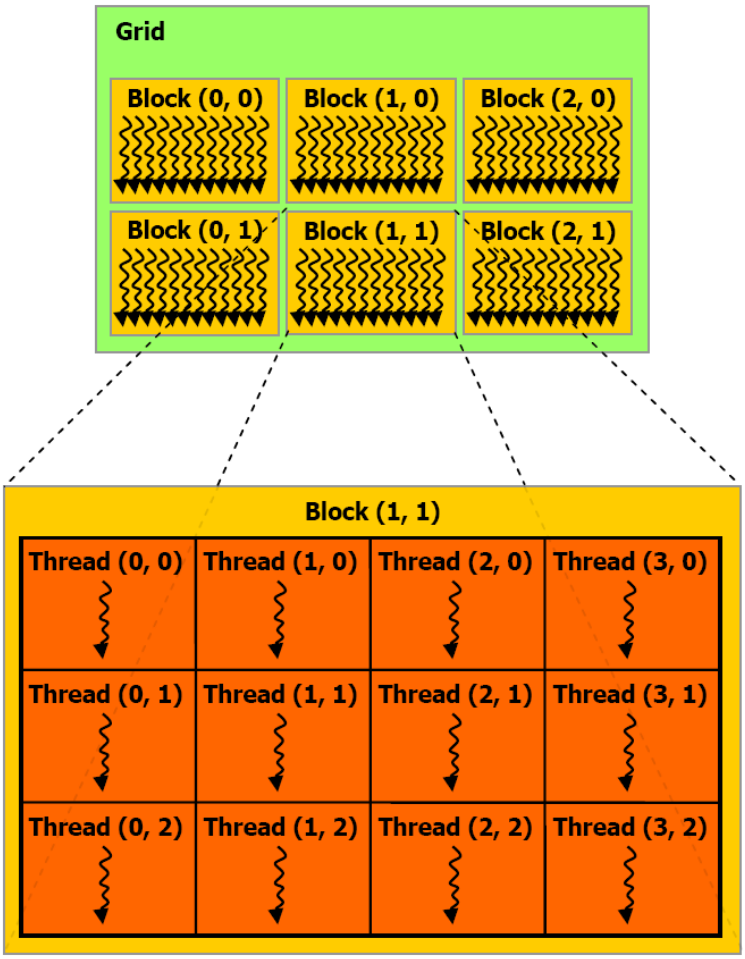


Figure 2.6: Organization of Grid and Thread Blocks [18]

As increasing number of threads, the efficient use of GPU becomes challenging. The use of thread blocks and grids are very useful to organize the threads. In Figure 2.6,

an example of thread blocks and grid of thread blocks was illustrated. As threads, blocks have also ID. In addition to ID values, dimension information is also kept for blocks. For example in Figure 2.6, “Block (1, 1)” has a dimension of (4, 3) and “Grid” has a dimension of (3, 2) for x and y. CUDA allows create three dimensional blocks, however there is a limiting value for the maximum number of the threads in a dimension of the blocks. These limiting values vary from one GPU model to another.

In addition to ease of organization, the use of thread blocks has other important benefits. One of them is accessibility of the threads in the same block to shared memory. Unfortunately, all problems cannot be divided into independent parts so easily. Sometimes the operation of a thread may depend on another thread’s operation; in other words, output of a thread may be input of another thread. In those cases, to avoid unnecessary calculations, thread cooperation is needed [54]. Roughly, after a thread writes its output to memory, this output can be accessed by other threads, this concept is thread cooperation. For efficiency the latency of the memory access should be low, this low latency memory demand is satisfied with shared memory within the thread blocks. While every thread in the same thread block can write and read from the shared memory of that block, they cannot access to shared memory of other blocks. In Figure 2.7 memory hierarchy of the threads, thread blocks and grids are presented. Each thread have its own local memory, each thread block has its own shared memory accessible by threads within the block. Moreover, each thread can access to global memory, constant memory and texture memory of the device. Since the data transferred between the host and device via global memory, global memory is the one of the most commonly used memory types. Furthermore a thread can access to constant memory and texture memory directly. All of these memory types have advantages and disadvantages. Shared memory allows very low latency memory access, however its capacity is very small. The global memory has very large storage capacity, but it is much slower than shared memory. Constant and texture memories are faster than the global memory but they are read only.

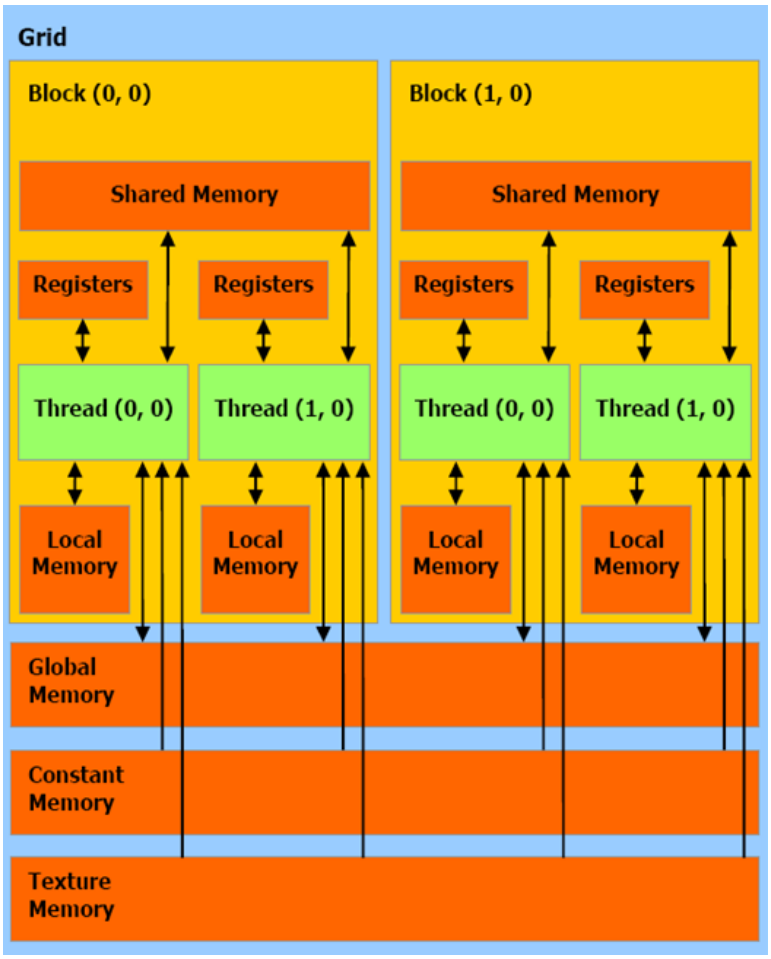


Figure 2.7: Memory Hierarchy of GPU [55]

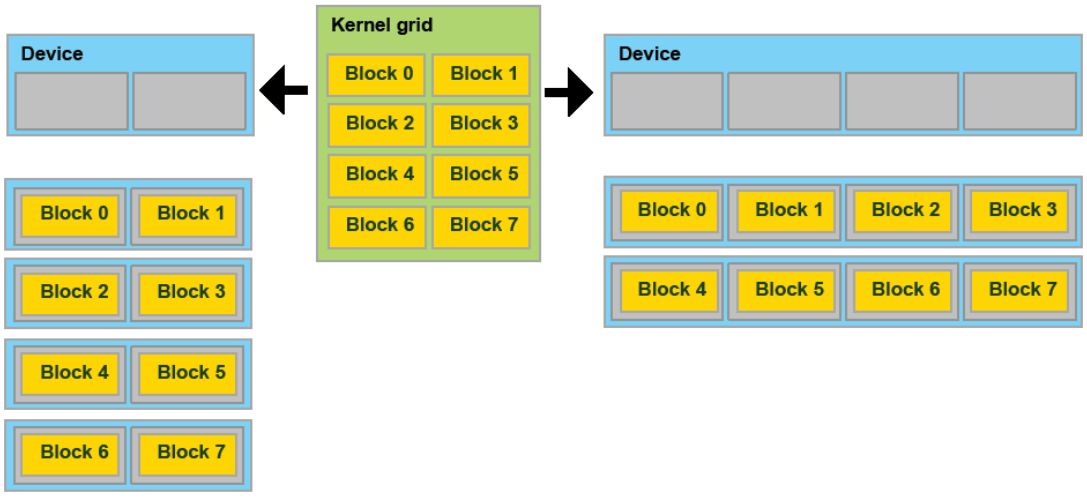


Figure 2.8: Transparent Scalability [54]

In Figure 2.8 Transparent Scalability [54] was illustrated. According to this concept thread blocks are allocated automatically according to the number of streaming multiprocessors of the hardware. While two thread blocks are executed concurrently for the device with two multiprocessors (on the left), four thread blocks are executed concurrently for the device with four SMs (on the right).

As it was mentioned before generally a GPU program has some serial portions to be executed on the host and some portions to be executed on device. In Figure 2.9 the execution scheme of a GPU program was illustrated. While the serial portions are executed on the host, parallel portions are executed on GPU.

Since the GPU hardware is different, there are some CUDA specific functions to develop codes running on GPU. The most common ones of these functions are as follows:

- **cudaMalloc** (**void**** pointer, **size_t** nbytes): Allocates device memory with the size of "nbytes".
- **cudaMemset** (**void**** pointer, **int** val, **size_t** nbytes): Initializes the nbytes of memory with the integer val.
- **cudaFree** (**void*** pointer): Frees the allocated memory.
- **cudaMemcpy** (**void** *dst, **void** *src, **size_t** nbytes, **enum cudaMemcpyKind** direction): Copies "nbytes" of memory from "src" to "dst". **cudaMemcpyKind** can be one of **cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost** and **cudaMemcpyDeviceToDevice.**

Kernels are C functions which cannot access host memory, they are not recursive, they must return void, and they cannot take static variables as inputs [54].

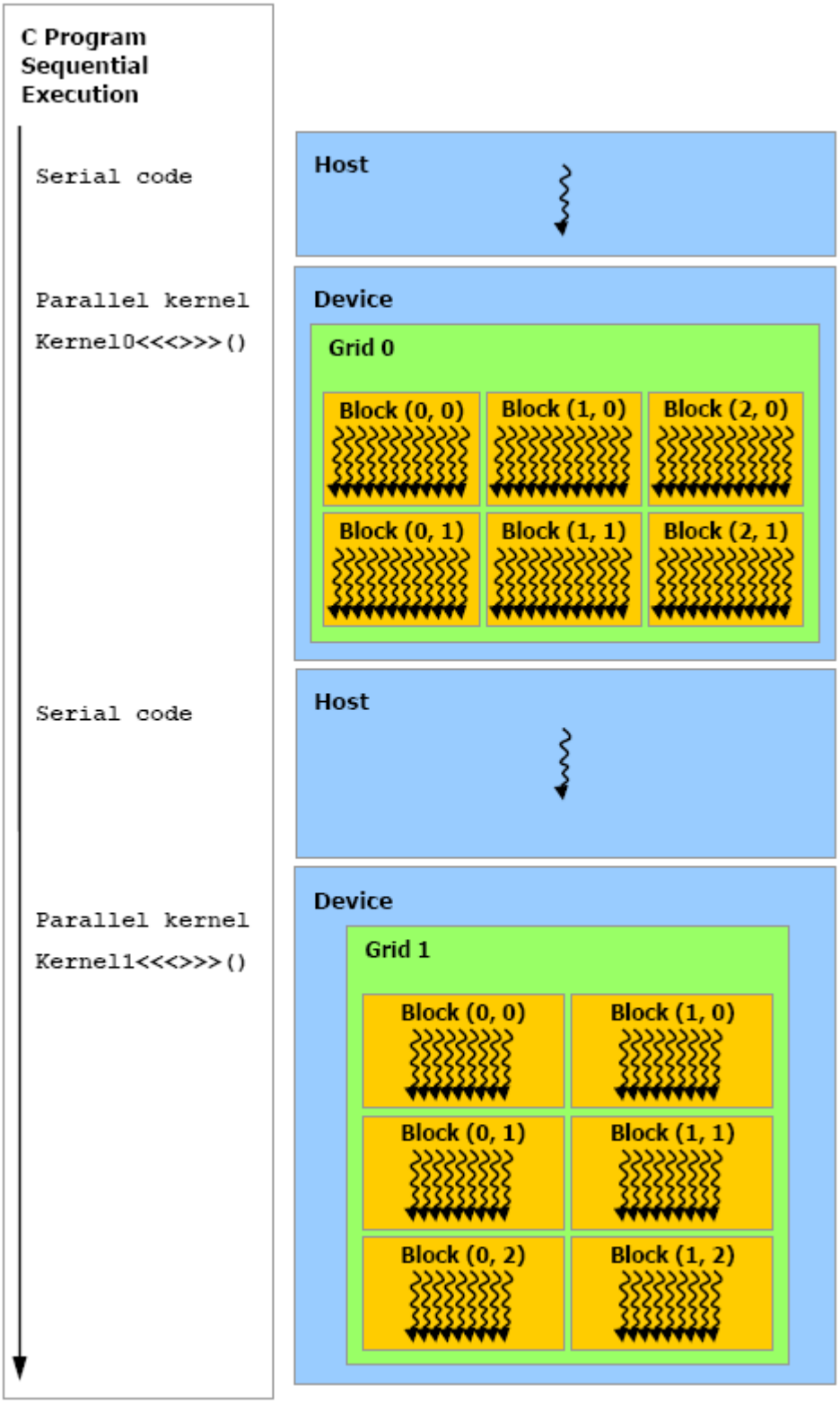


Figure 2.9: Use of both CPU and GPU [18]

There are three types of kernel. These are:

__global__ : They must return void. They are called from host and executed on device.

__device__ : They are called by device and executed on device.

__host__ : They are called by host and executed on host.

The syntax of a kernel as follows:

kernel0<<<**dim3** dimGrid, **dim3** dimBlock>>> (**args**)

In this syntax dimGrid is the dimension of grid, in other words the number of the blocks in x and y directions. dimBlock is the dimension of the block, unlike grids, blocks can be three dimensional.

In Figure 2.10 a sample code was given for the example problem in the Figure 2.5. In the first part a kernel, called KernelSample, was defined. Memory allocations, memory copying and invoke of the kernel are completed in the main function. The kernel is executed for one block with 6 threads.

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

__global__ void KernelSample(int *input,int *output)
{
	int x=input[threadIdx.x];
	int y=x*x;
	output[threadIdx.x]=y;
}

int main()
{
	//input array on the host machine
	int h_input[6]={2,1,8,6,3,7};

	//output array on the host machine
	int h_output[6];

	//input array on the device machine
	int *d_input=0;

	//output array on the device machine
	int *d_output=0;

	//allocation of input array on the device memory
	cudaMalloc((void**)&d_input,6*sizeof(int));

	//allocation of output array on the device memory
	cudaMalloc((void**)&d_output,6*sizeof(int));

	//copy input array from host to device
	cudaMemcpy(d_input,h_input,6*sizeof(int),cudaMemcpyHostToDevice);

	// invoke kernel for 1 block of 6 threads
	KernelSample<<<1,6>>>(d_input,d_output);

	//copy output array from device to host
	cudaMemcpy(h_output,d_output,6*sizeof(int),cudaMemcpyDeviceToHost);

	// free input memory
	cudaFree(d_input);

	// free output memory
	cudaFree(d_output);

	return 0;
}
```

Figure 2.10: Sample code for the example in Figure 2.5

CHAPTER 3

IMPLEMENTATION OF MULTIPLE FRONT SOLVER ON GPU ARCHITECTURE

3.1 Introduction

This chapter includes the detailed information about the multiple front solution method and its implementation on GPU architecture. In the first section of this chapter main steps of the multiple front algorithm, which are partitioning, local assembly, condensation, assembly and solution of the interface equations and the back substitution, are presented. In the following section, the implementation of the multiple front solution method on GPU architecture is given. Finally, the test problems are introduced and the results obtained from these tests are discussed in the last section of this chapter.

3.2 The Multiple Front Algorithm

Multiple front solution method is actually the classical substructure based solution method. The main steps of the multiple front algorithm was illustrated in Figure 3.1. These parts are partitioning, condensation, assembly and solution of interface equations. The first step of the multiple front algorithm is partitioning. In this step, a structure is divided into multiple substructures. Partitioning is usually handled by automatic graph partitioning algorithms [22] that basically attempt to equate the

number of elements in a substructure while keeping the interface sizes as small as possible.

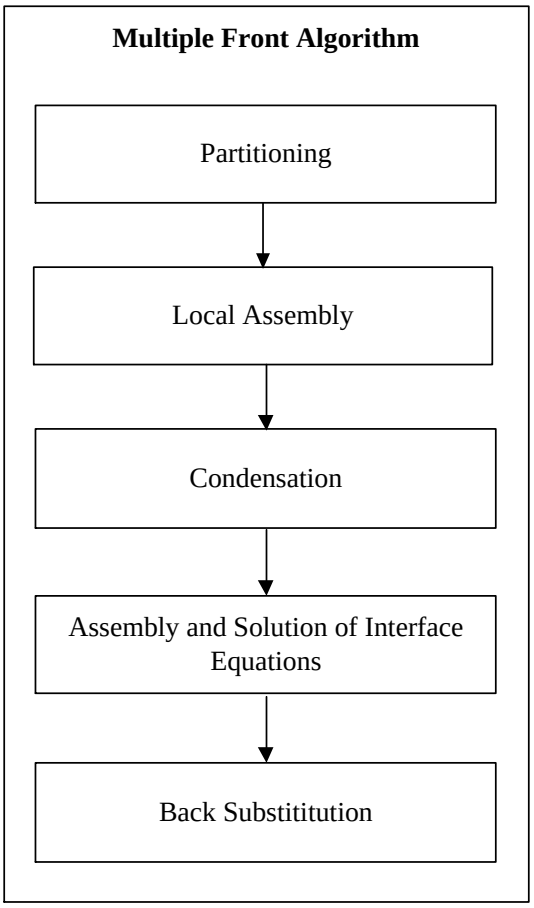


Figure 3.1: Multiple Front Algorithm

When a structure is partitioned into smaller substructures, some of the nodes become common for multiple substructures and some of the nodes belong to only one substructure. Such common nodes are called interface nodes and the others are called internal nodes. After partitioning, local assembly step initiates. Stiffness matrices of finite elements in each substructure are computed and assembled forming the equation system of the substructure in such a way that the equations belonging to the interface nodes were stored at the end of the stiffness matrix. The stiffness matrix of each substructure is highly sparse. Since the internal DOFs of a substructure do not affect other substructures, the internal equations of each substructure are reduced to

the interface equations by a procedure very similar to Gauss elimination. This procedure is called condensation. After the condensation only interface equations of the substructures remain. These equations are assembled together forming the system of interface equations which is a dense matrix. With the solution of this system, the displacement values of the interface DOFs are obtained. Then, the solution of the internal equations is obtained by back substitution, also called recovery, procedure.

3.2.1 Partitioning

Beside the solution of the linear system of equations, formation of the stiffness matrices and force vectors may also be very time consuming procedures in FEA. Because of this reason substructuring methods become an attractive way of solution for FEA. In multiple front method internal DOFs of each substructure are condensed to the common DOFs forming the interface equations. Since the partitioning affects the time required for condensation of internal equations and solution of the interface equations, it has a significant role in the substructure based methods. Static partitioning is used for the problems where the computational workload can be calculated before the solution and remains constant during the solution. In multiple front algorithms, static partitioning algorithms are suitable since all substructures are assembled and factorized once. In this study initial partitioning was performed by the use of METIS [56], a software package using multilevel partitioning method. In Figure 3.2, a 160×160 meshed structure is partitioned into 16, 64 and 128 substructures, by METIS [56], respectively and every substructure is presented with a different color.

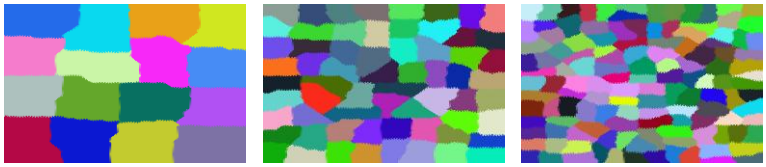


Figure 3.2: 160×160 meshed structure with 16, 64 and 128 substructures

In addition to partitioning of a structure into substructures, the assembly sequence of substructures is also determined by partitioning. The aim of determining the assembly sequence is to form the interface equations in a way that allocating the resources most efficiently and avoid a processor to become idle while other processors are working.

After partitioning, the assembly sequence for the structure is determined. This sequence can be expressed as an assembly tree [34]. In Figure 3.3 an assembly trees for a 4×4 meshed structure is illustrated. In this figure, while individual finite elements are shown as rectangles with dashed line borders, substructures are shown as rectangles with solid line borders. In Figure 3.3, the structure is partitioned into four substructures. The substructures in the first level (substructures 1-4) are assembled forming their parent structure in the second level of the assembly tree.

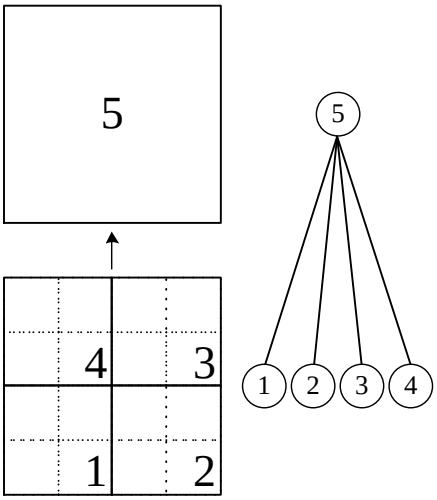


Figure 3.3: Assembly tree with four substructures

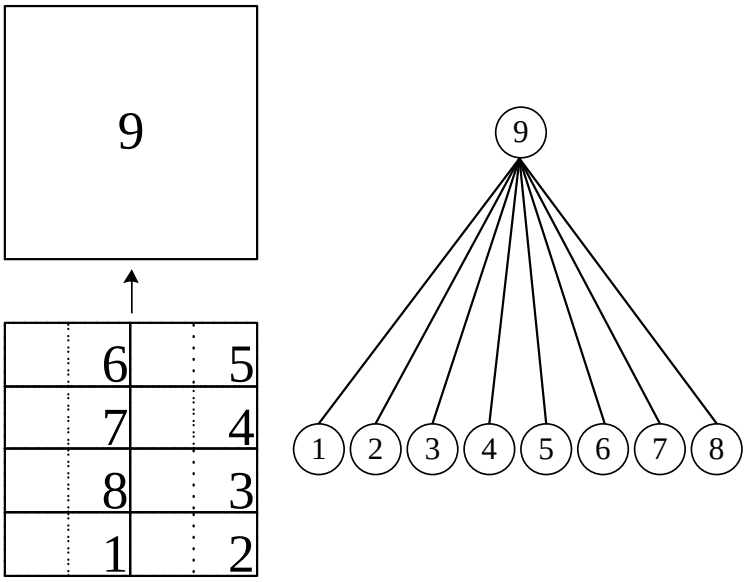


Figure 3.4: Assembly tree with eight substructures

Besides the information stored by assembly trees, the number of initial substructures have a significant role in the performance of the multiple front algorithm. In multiple front algorithm assembly trees have two levels only. In the first level, there are finite element substructures whose number and the numbers of the finite elements in each substructure are defined in the partitioning stage. By assembly of these substructures the uppermost level in the assembly tree is obtained. In Figure 3.4 the structure shown in Figure 3.3, is partitioned to 8 substructures. Although both assembly trees belong to the same structure and they give the same results, the performance of the solutions differs. This difference caused by the numbers of the initial number of substructures of the assembly trees, which is four in the first assembly tree and eight in the latter one. Since the number of the substructures is less in Figure 3.3, there are more individual finite elements in a single substructure. Therefore, the time required for condensation of a substructure in the first assembly tree more than the one in Figure 3.4. Moreover since the second assembly tree have more substructures, it allows to use more threads during condensation in parallel. But dividing the structure into larger number of substructure causes an increase in the size of interface matrices.

3.2.2 Condensation

As the partitioning part is completed by use of METIS [56] the substructures and the assembly sequence are obtained and the solution is initiated with the condensation step. For a better understanding of the remaining steps of the solution algorithm, an illustrative example problem is utilized. For this purpose the 4×4 meshed structure in the Figure 3.5 is used for the sake of simplicity. It is divided into four substructures and the assembly tree presented in the Figure 3.3 is used. In Figure 3.6 the finite elements in the first substructure and the node numbering are illustrated, note that these properties are the same for the remaining of the structure. As it can be observed from Figure 3.5, each substructure is composed of 2×2, 4-node bilinear quadrilateral elements and each element has 8 DOFs. Node numbering starts from the internal nodes first, then it continues with the nodes contributing to interface equations, these nodes are shown in red color in Figure 3.6.

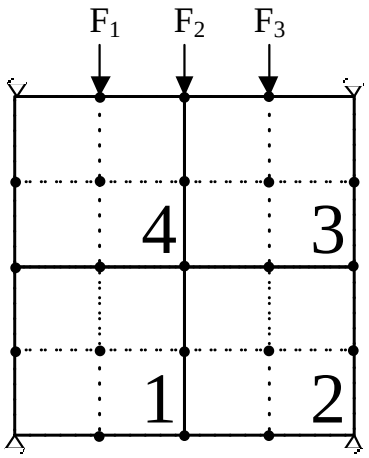


Figure 3.5: Example structure with 4×4 mesh

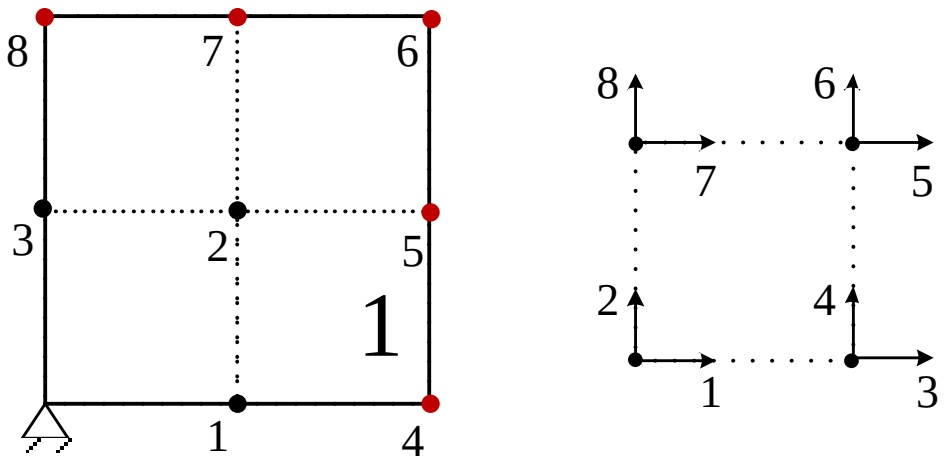


Figure 3.6: Substructure 1 and a bilinear 4-node membrane element

In condensation process the internal DOFs are reduced to the interface DOFs. This process is illustrated in Figure 3.7 for the first substructure. The internal DOFs (DOFs 1-6) were condensed to interface DOFs (DOFs 7-16) of the substructure forming the interface equations of the substructure.

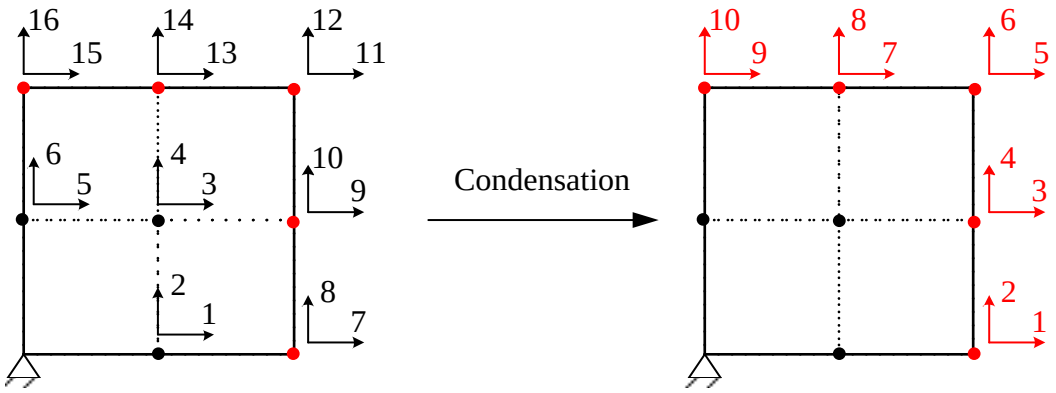


Figure 3.7: Condensation of the first substructure

In direct solution approaches, condensation is completed by the partial factorization of the stiffness matrix till the first interface equation. In this study LDL^T method is

used for decomposing symmetric stiffness matrices. For the decomposition the following equations are utilized.

$$[K] = [L][D][L]^T \tag{3.1}$$

$$D_{jj} = K_{jj} - \sum_{k=1}^{jl} L_{jk}^2 D_{kk} \qquad j=1 \text{ to } NEQ \tag{3.2}$$

$$L_{ij} = K_{ij} - \sum_{k=1}^{jl} L_{ik} L_{jk} D_{kk} \qquad \text{for } i>j,\ j=1 \text{ to } NEQ \tag{3.3}$$

$$L_{ij} = \frac{L_{ij}}{D_{jj}} \qquad \text{for } i>j,\ j=1 \text{ to } LEQ \tag{3.4}$$

$$F_i = F_i - \sum_{k=1}^{jl} L_{ki} F_k \qquad i=1 \text{ to } NEQ \tag{3.5}$$

$$F_i = \frac{F_i}{D_{ii}} \qquad i=1 \text{ to } LEQ \tag{3.6}$$

where,

NEQ = number of equations

LEQ = number of internal DOFs

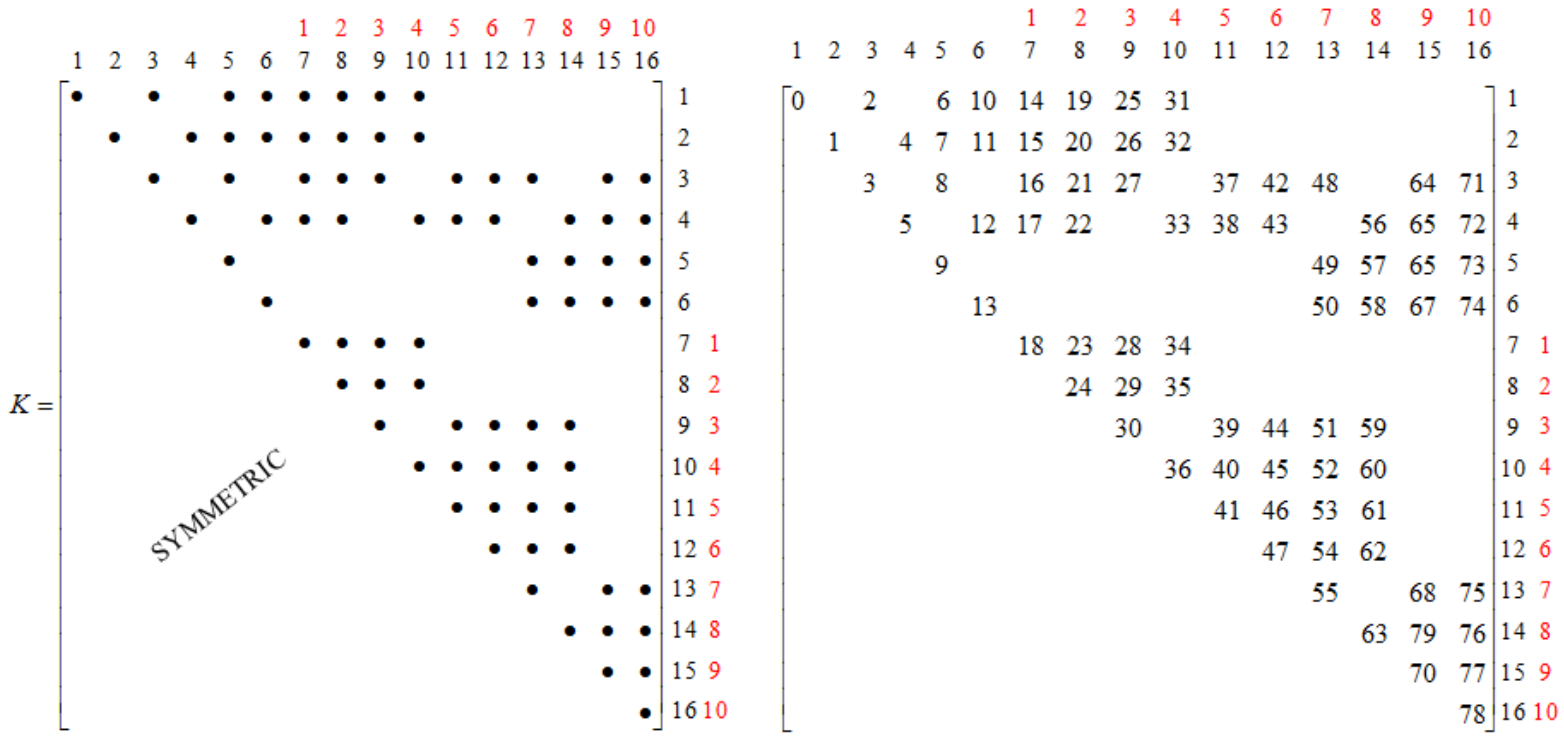
jl = minimum of j-1 or *LEQ*

The formulae used in condensation procedure given in the Equations 3.1-3.6 are for the full matrices. In Equations 3.1-3.6, *K* denotes the symmetric stiffness matrix of the substructure, *L* denotes the unit lower triangular matrix, *D* denotes the diagonal matrix and *F* denotes the force vector of the substructure. The condensation of the stiffness matrix is completed by using the Equations 3.2-3.4 and the condensation of

the force vector is completed by using the Equations 3.5-3.6. In these equations *NEQ* refers to number of equations which equals to 16 for the first substructure. On the other hand *LEQ* refers to number of equations to be reduced, in other words, it is the number of internal DOFs which equals to 6 for the first substructure.

The condensation process is completed by using Equations 3.1-3.6 recursively. The decomposition of a sparse matrix is, however, not as straight forward as it is in a full matrix. Since, only the non-zero elements stored in sparse systems, a symbolic factorization process has to be performed for the determination of the volume and the calculation sequence of the non-zero elements in *L*. Because of this reason, the formulae can be applied numerically only after the symbolic factorization process.

In Figure 3.8 non-zeros in the upper triangular part of the symmetric stiffness matrix and their storage in CSC format is illustrated. While the numbers in black above the matrices represent row and column IDs of the DOFs within the substructure stiffness matrix, the numbers in red represent the row and column IDs of the DOFs within the interface stiffness matrix. In the matrix, at the right side of Figure 3.8 the IDs of non-zero elements in the storage algorithm are shown. The numbering starts from element in the first row, first column, then continues with the uppermost non-zero element in the next column. The two arrays used for the storage algorithm was shown in Figure 3.8, col_ptr and row_ind. The col_ptr stores the locations of the first non-zero elements in the column, in other words the i^{th} value in col_ptr array gives the ID of the first non-zero element in the ith column. The row_ind stores the row indices of the non-zero terms. According to the size and the sparsity of the matrix, CSC format may be very advantageous, since it requires significantly less storage size than storing the same matrix in dense format. Since the non-zero terms are accessed by using row_ind and col_ptr arrays, the performance of the programs, however decreases due to the indirect addressing.



col_ptr = [0, 1, 2, 4, 6, 10, 14, 19, 25, 31, 37, 42, 48, 56, 64, 71, 79]: Pointer array of the locations of first non-zero elements in the column

row_ind = [1, 2, 1, 2, 3, 1, 2, 4,..]: Row indices of the non-zero elements

Figure 3.8: Non-zero elements in the stiffness matrix of the first substructure in CSC format

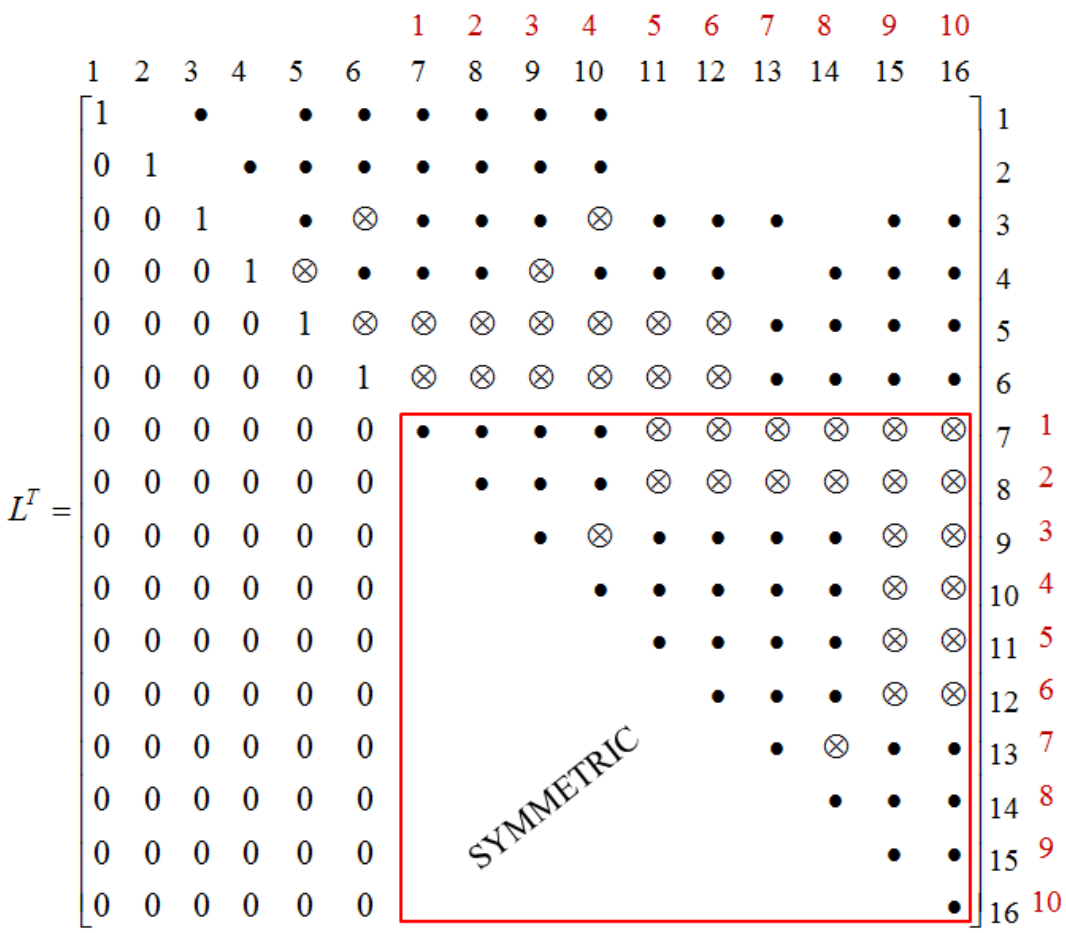


Figure 3.9: Transpose of the unit lower triangular matrix

In sparse systems, besides the original non-zero elements, some of the zero terms become non-zero during the elimination process. These terms called fill-in elements. The transpose of *L* is shown in Figure 3.9. In this figure the fill-in elements were shown with a cross in a circle. Moreover, since a fill-in element will affect the subsequent calculations, a factorization sequence is also needed. This necessary information is obtained with the symbolic factorization in two arrays. First array stores the locations of the first non-zero elements in the unit lower triangular matrix. This array is similar to col_ptr array but it stores the non-zero values in the lower triangular matrix. The second array stores the information of the next column to be

factorized, for each column. For example, since the element in the 3rd column and the elements 5th-10th columns, in the first row, are not zero, the first column is used in the elimination of these columns. With this information a sequence for the elimination of the matrix is obtained.

The information obtained from symbolic factorization is used in the numerical factorization. The numerical factorization calculates the values and places of non-zero terms of *L* matrix and the values in the main diagonal of *D* matrix by using Eq. 3.2-3.4. It continues until the first interface DOF of the substructure, which is 7th DOF in the example case. The square matrix and the vector under that DOF in the substructure stiffness matrix and the force vector, give the condensed equations of the substructure those will contribute to the interface stiffness matrix and interface force vector. The condensed part of the stiffness matrix was shown in the red box, in Figure 3.9. This part is also called Schur complement.

3.2.3 Assembly and Solution of Interface System

After the condensation of the all substructures, the condensed parts should be assembled according to the assembly tree of the structure to form the interface equations. In Figure 3.10 the assembly of the condensed substructures was illustrated. As it was shown in the figure, every substructure is condensed to its interface DOFs, which are shown in red. Then these substructures are assembled according to the assembly tree of the structure. The fifth structure is the parent node of the substructures and it is the final structure in the assembly tree. While the original structure has 23 nodes with 46 equations, the final assembled structure has 9 nodes with 18 equations. The remaining 28 equations are the internal equations to be solved within each substructure after the interface equations are solved and sent back to the substructures.

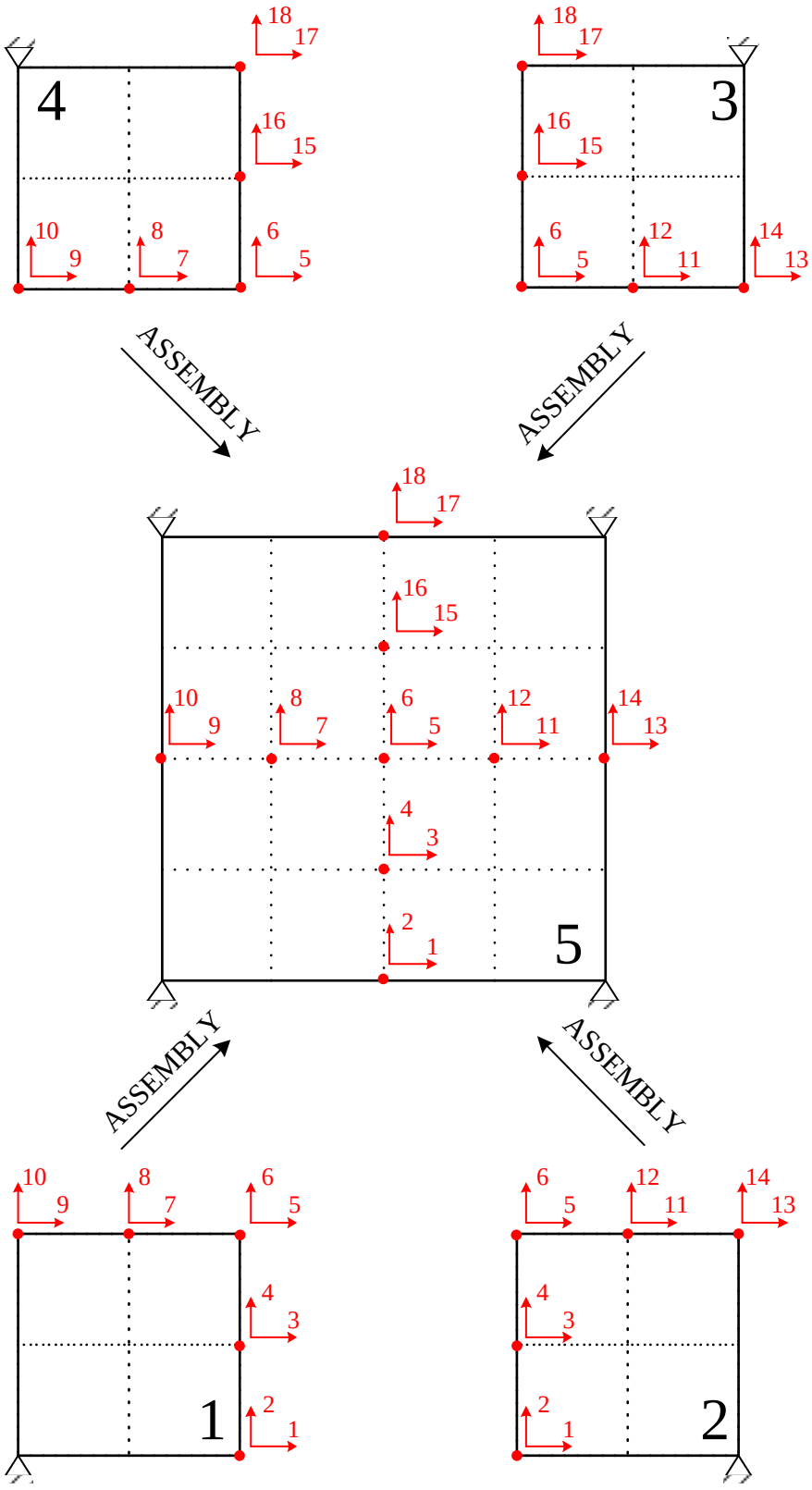


Figure 3.10: Assembly of Substructures

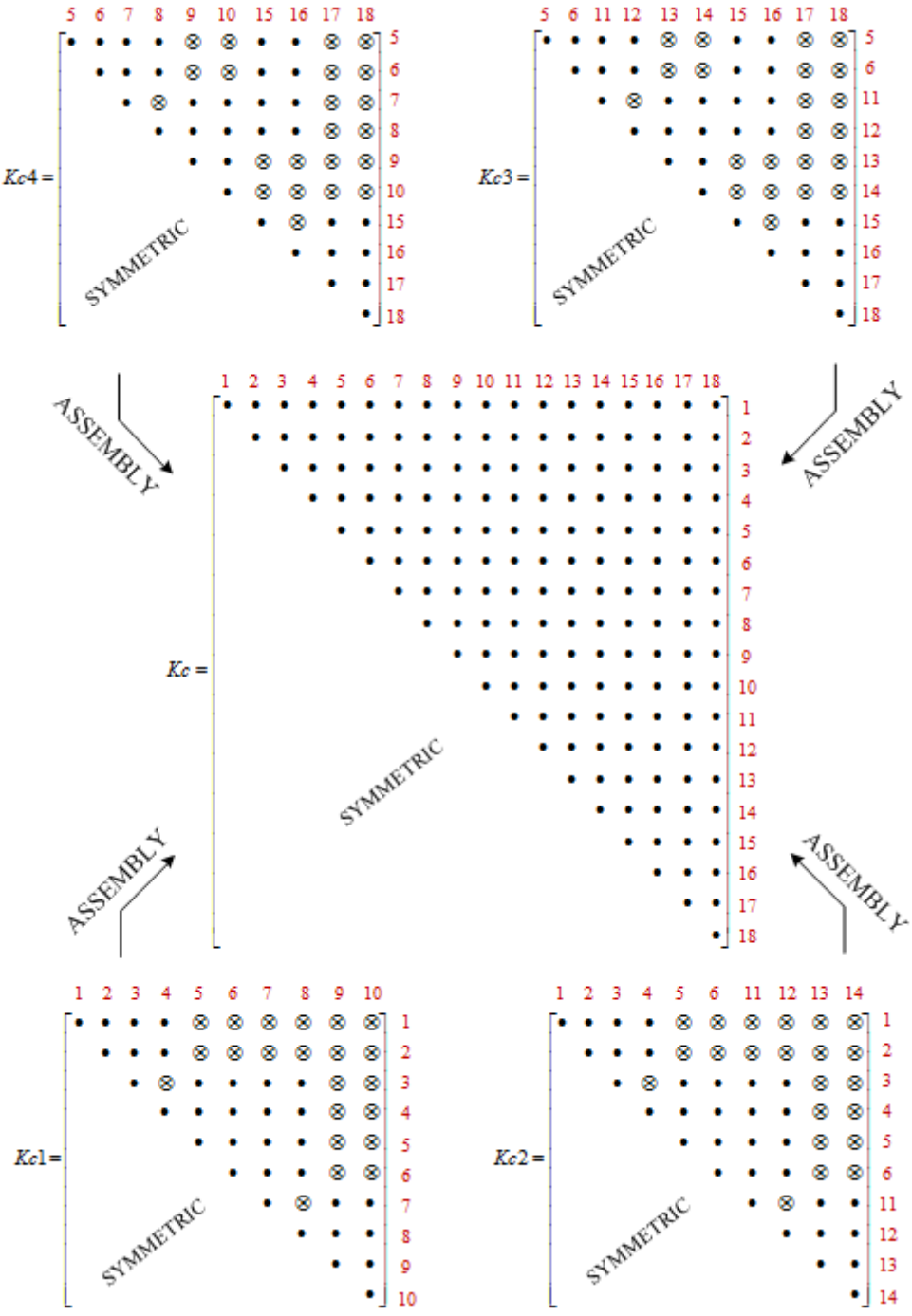


Figure 3.11: Assembly of Interface Matrix

As the assembly of the structures operations was shown in Figure 3.10, the assembly of the stiffness matrices of the substructures was shown in the Figure 3.11. In this figure the condensed matrices were denoted by *Kc1*, *Kc2*, *Kc3* and *Kc4* for the substructures 1, 2, 3 and 4 respectively. The final interface stiffness matrix belonging to fifth substructure was denoted by *Kc*. For each row and column of the condensed stiffness matrices of the substructures, the corresponding row and column IDs of the final interface matrix were shown with the red numbers, on the above and the right side of the substructure condensed stiffness matrices. For example the element on the second row and third column in *Kc4* matrix will contribute to the element on the sixth row and seventh column in the final interface matrix *Kc*. Therefore substructure condensed stiffness matrices and condensed force vectors are assembled according to these numbers, which are given as input information with the assembly tree. After the assembly of the condensed stiffness matrices of the substructures, the interface equation system is obtained. Since the obtained system of equations is dense, its solution is straight forward.

3.2.4 The Solution of Internal Equations

After the solution of the interface equations, the displacement vector belonging to the interface DOFs are obtained. Interface DOFs are sent to each substructure. With the known interface displacements, the only unknowns are the displacement values of the internal DOFs within each substructure. To obtain the solution of the whole structure the internal equations of every substructure have to be solved as a final step. In Figure 3.12 the system of equations for the first substructure after the solution of the interface equations, was illustrated. In the left hand side, the factorized stiffness matrix and displacement vector of the substructure were shown. In the displacement vector, the displacement values of the internal DOFs, those have to be computed

were shown as "?" and the known displacement values belong to interface DOFs of the system were denoted as dc_i. By using the portion of the factorized stiffness matrix in the blue box and the known displacement values of the interface DOFs, the internal equations were computed recursively by starting from the last internal DOF in the substructure. While dense solution algorithms are used for the solution of the interface equations, recovery process is handled by sparse algorithms as it is in condensation operation. With the solution of internal equations within the each substructure, the solution is finalized

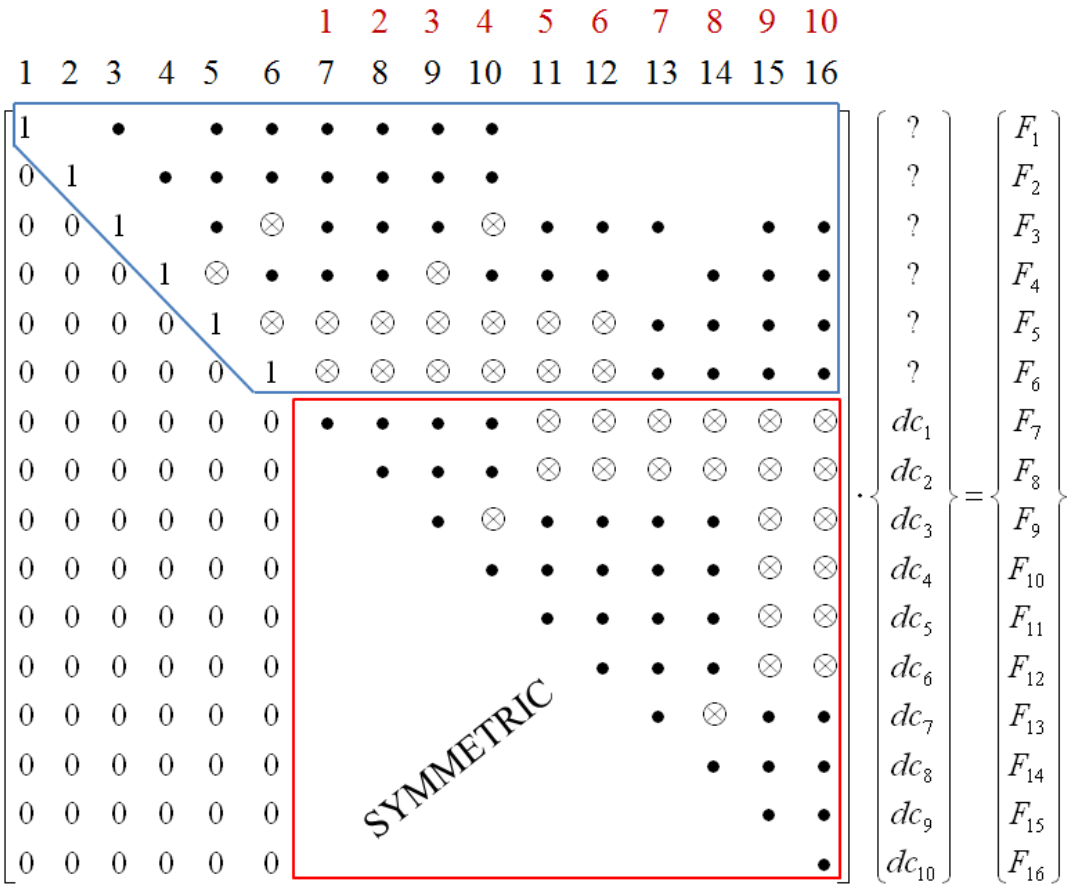


Figure 3.12: System of Equations for Substructure 1

3.3 GPU Implementation

The kernels used for the GPU implementation of the algorithm were shown in the Figure 3.13. As it is mentioned before the algorithm is composed of five main steps which are: partitioning, local assembly, sparse matrix condensation, assembly and solution of the interface equations and the back substitution. Only the last three steps of the solution algorithm, i.e. condensation, assembly, and solution of interface equations are handled by GPU.

3.3.1 Sparse Condensation

In this step *GPU_SparseSymbolic* and *GPU_SparseCondense* kernels, which are responsible for the completion of the symbolic sparse factorization and the numerical factorization respectively, are used. Since the algorithms for the symbolic and the numerical factorization of a sparse matrix are sequential, no parallelism can be obtained from the operations within the condensation procedure of a single substructure. Because of this reason the only parallelism can be obtained by creating multiple threads, each of whom is responsible for the condensation of one substructure. In other words the number of the threads created for symbolic factorization and numerical factorization is equal to the number of substructures. Thus, each thread is responsible from the factorization operations of a single substructure and can only reach to the data related with that substructure. Since the amount of the data is too large for shared memory, data is stored in global memory, causing a decrease in the performance of the algorithm.

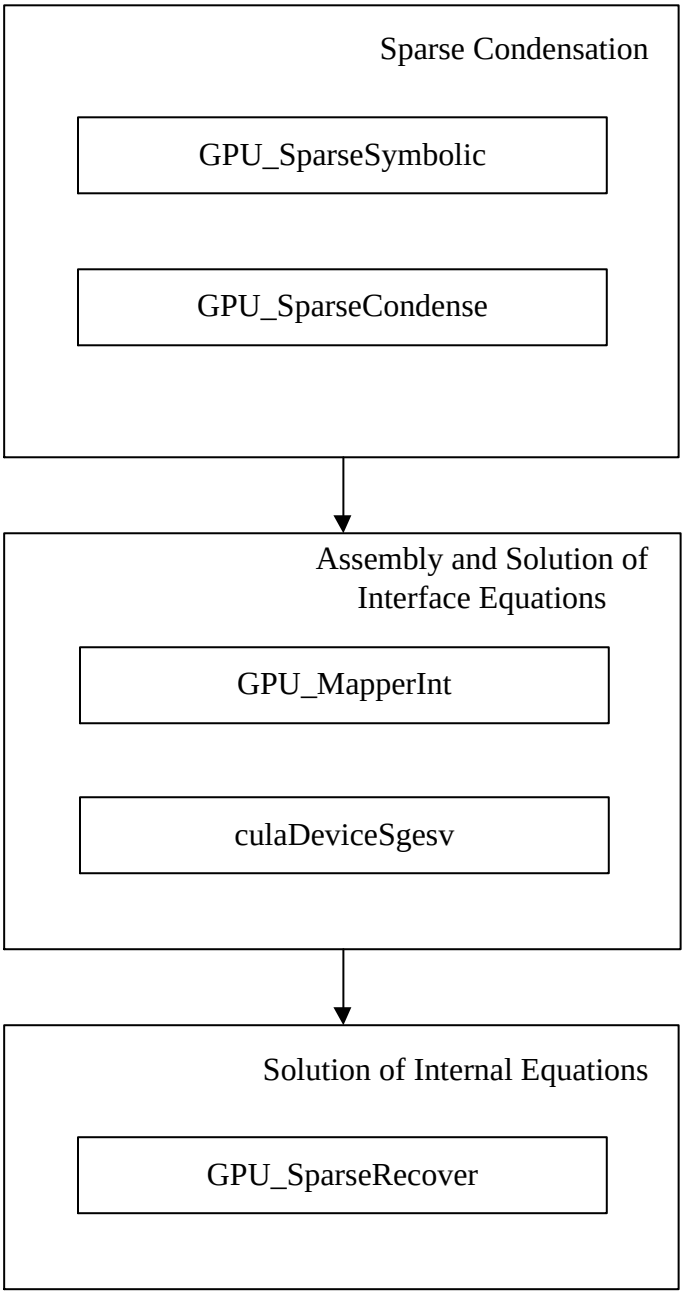


Figure 3.13: Subroutines used for GPU implementation of multiple front algorithm

3.3.2 Assembly and Solution of Interface Equations

Assembly of the interface equations is completed by *GPU_MapperInt* kernel. The number of the operations completed in the *GPU_MapperInt* and its effect to the performance of the algorithm is negligible according to other functions such as *GPU_SparseCondense* kernel. Because of this reason, an algorithm, which can be easily implemented, is chosen.

Since elements from different Schur complements of the substructures may be addressed to the same location in the interface equations, the mapping operations of the substructures are completed sequentially to avoid race condition. On the other hand, because of the reason that each element within the same substructure has different locations in the interface equations, the mapping operations can be completed concurrently within a substructure. In other words, during the *GPU_MapperInt* kernel a thread block is created, this thread block starts mapping operations of the first substructure. Each thread assembles an element from the Schur complement of the first substructure to the interface equations in parallel. After the completion of the mapping operations of that substructure, thread block starts mapping of another substructure.

After the assembly of the interface equations, a dense matrix solver for the solution of the interface system is required. For this purpose a commercial GPU accelerated linear algebra library, CULA [57] was used. CULA includes various linear algebra functions for single and double precision. Besides the host function, CULA also includes device functions where the data in the GPU memory can be used directly. *culaDeviceSgesv* function is a CULA subroutine, which is a device function using the data directly from the GPU memory, and solves the linear system of equation by using Gauss elimination.

3.3.3 Back Substitution

The final step of the algorithm is the solution of the internal equations of the substructures. After the solution of interface equations, the remaining internal DOFs of each substructure are calculated by the *GPU_SparseRecover* kernel. Like the sparse matrix factorization, the solution of the internal equations, in other words recovery operations, should be completed in order within the stiffness matrix of a substructure. On the other hand, the recovery operations of different substructures can be managed concurrently. Thus, the algorithm for the recovery operations is parallel in substructure level but sequential within the substructures. As for the factorization operations, one thread is created for each substructure, and these threads complete the recovery operations of those substructures.

3.4 Test Problems and Results

The performance of the multiple front algorithm was tested for different structures divided into various numbers of substructures with different GPUs. Two test structures with different sizes were used for the testing the performance of the algorithm. The first structure is composed of 50×50 shell elements. Each shell element has 4 nodes, with 6 DOFs for each node. The equation system has a size of 15000 equations. This system was partitioned into 8, 16, 32 and 64 substructures. The time values obtained from the solution of the first structure with the graphic cards GTX 275, GTX 580 Amp and Tesla C2050 were presented in the Figure 3.14.

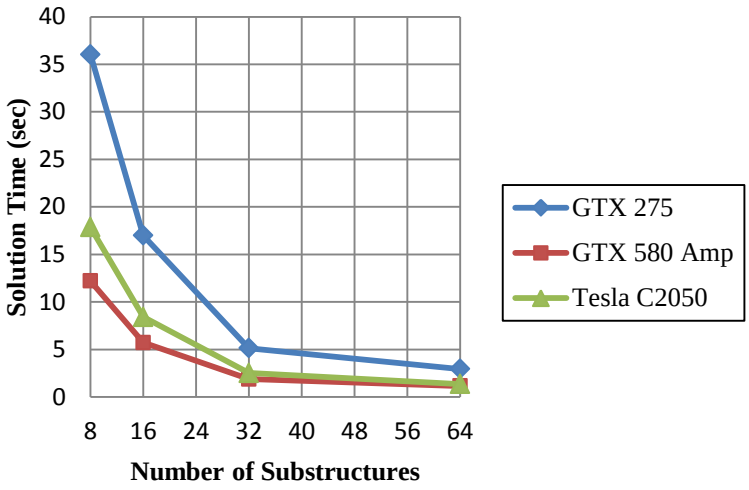


Figure 3.14: Solution time values obtained from the solution of the structure with 50×50 elements

A larger structure with 160×160 shell elements, having 153600 equations, was used as the second testing structure. This structure was partitioned into 16, 32, 64 and 128 substructures. But, since the larger number of substructures yields interface equations with larger sizes, the global memory sizes of GTX 275 and GTX 580 Amp were insufficient for the solution of the structure with large number of substructures. Because of this reason the solutions of the structure with 16 and 32 substructures were obtained from GTX 275 graphical card and the solutions of the structure with 16, 32 and 64 were obtained from GTX 580 Amp graphical card. On the other hand, the size of the global memory of Tesla C2050 was sufficient for the solution of the structure with 16, 32, 64 and 128 substructures. The solution time values of the second structure with different numbers of substructures were presented in Figure 3.15.

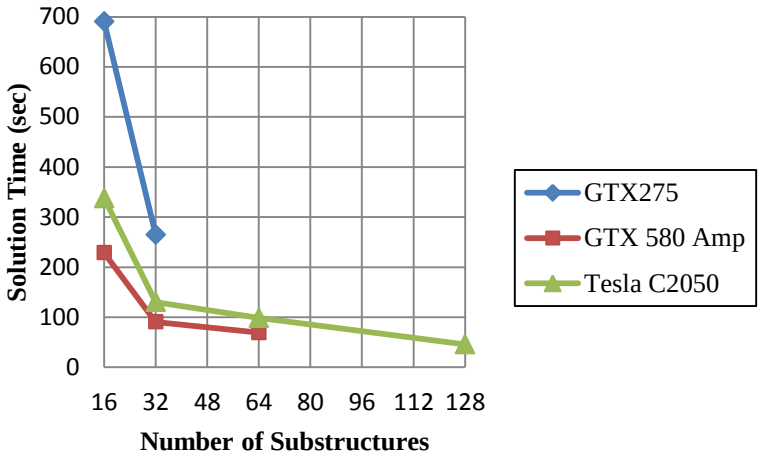


Figure 3.15: Solution time values obtained from the solution of the structure with 160×160 elements

As it can be observed from the Figure 3.14 and Figure 3.15, larger numbers of substructures yield shorter solution time for the solution of the both structures. In Figure 3.15 since the solution of the structure with 128 substructures could be obtained only with Tesla C2050, the shortest time period for the solution of the second structure was obtained from this GPU. On the other hand, GTX 580 Amp has the best performance for the same number of substructures among the performances of the solver on GTX 275 and Tesla C2050. The main reason of this situation is caused by the maximum number of floating point operations per second that the device is capable of. This value is 1581.1 GFLOPs, 1030.4 GFLOPs and 1010.9 GFLOPs for GTX 580 Amp, Tesla C2050 and GTX 275 respectively. As a result it is expected that GTX 580 Amp solves the same system of equations in a shorter period of time than GTX 275 and Tesla C2050.

The ratios of elapsed time periods in each solution step to total solution time of the 50×50 element structure divided into 8 and 64 substructures were presented in Figure 3.16. The pie charts given in the upper side of the figure belong to the solution of the structure with 8 substructures, whereas the pie charts given in the lower side belong

to the solution of the structure divided into 64 substructures. According to the figure it can be observed that, while the time passed during the sparse condensation shortens, the elapsed time during the solution of interface equations increases with the increasing number of substructures. Moreover, this situation is valid for all of the GPUs used in the tests. Although the GTX 580 Amp and Tesla C2050 have similar architectures, the portions of solution time of the steps in these GPUs differ. The reason of this situation is caused by the fact that since the sparse condensation part cannot fully utilize the computational power of the GPUs, this step is completed faster in GTX 580 Amp than Tesla C2050, because the clock rate of GTX 580 Amp is greater. However, an optimized package used for the solution of interface equations, the time passed in this step is very close to each other for both GPU. This causes the difference between the amounts of portions of time passed in each step for these two GPUs.

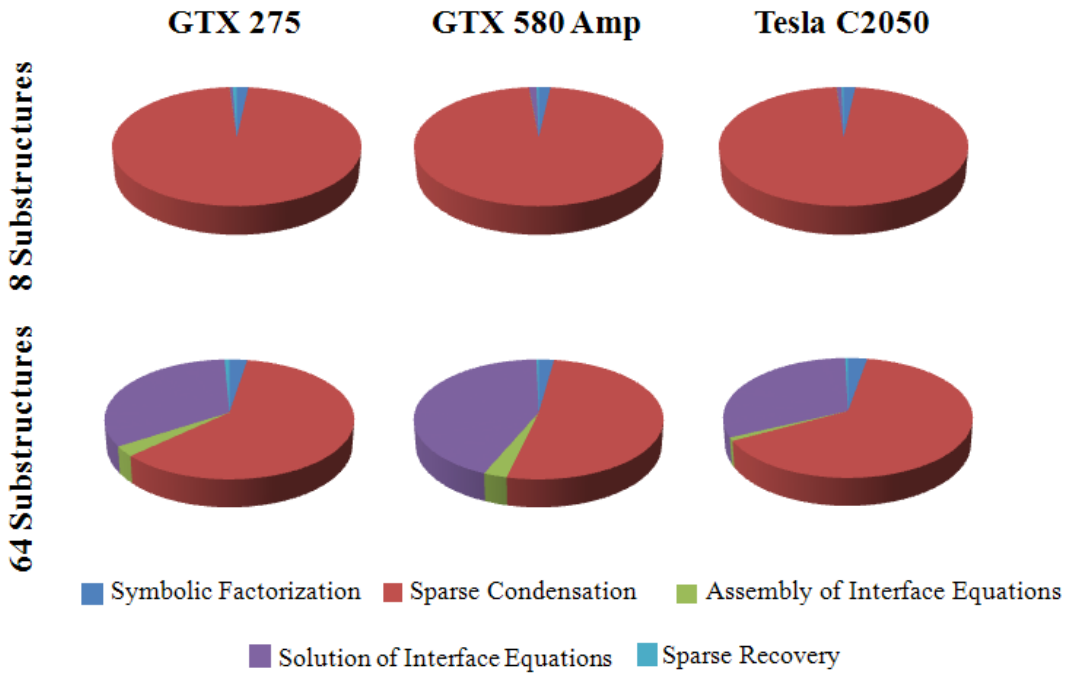


Figure 3.16: The effect of solution steps to the solution time of structure with 50×50 elements with 8 and 64 substructures

When the problem size becomes larger, elapsed time values of the solution steps change drastically with the increasing number of substructures. As in the first structure, the increasing number of substructures yields a decrease in the elapsed time for the sparse condensation and an increase in the elapsed time of the solution of the interface equations. The values of time passed during these steps in the solution of the structure with 160×160 elements by using Tesla C2050 were shown in Figure 3.17. For increasing the performance of the solver, increasing the number of substructures becomes helpful for shortening the time passed during the sparse condensation. However, as it can be observed from the Figure 3.16 and Figure 3.17 increasing the number of substructures causes an increase in the solution time of the interface equations. This increase caused by the increase in the size of the interface equations. In Figure 3.18 the sizes of interface equations of both systems for the various numbers of substructures were presented. Consequently, this situation causes an increase in the solution time of the interface equations. Thus, increasing the number of substructures yields an increase in the performance of the solver due to the performance gain in the sparse condensation part, but after a point increasing the number of substructure causes a performance loss in overall due to the increase in the solution time of the interface equations.

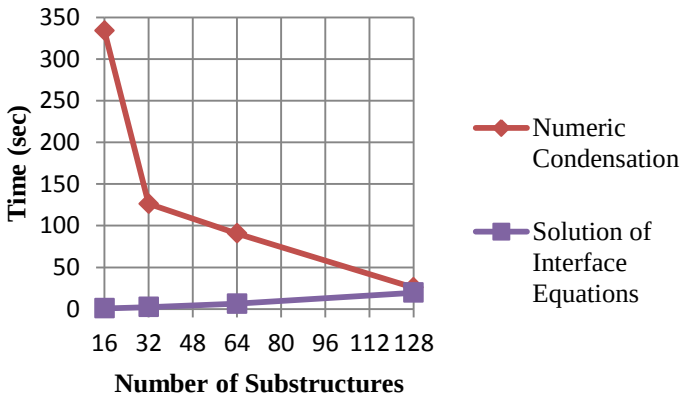


Figure 3.17: The sparse condensation time and solution time of the interface equations for 160×160 structure

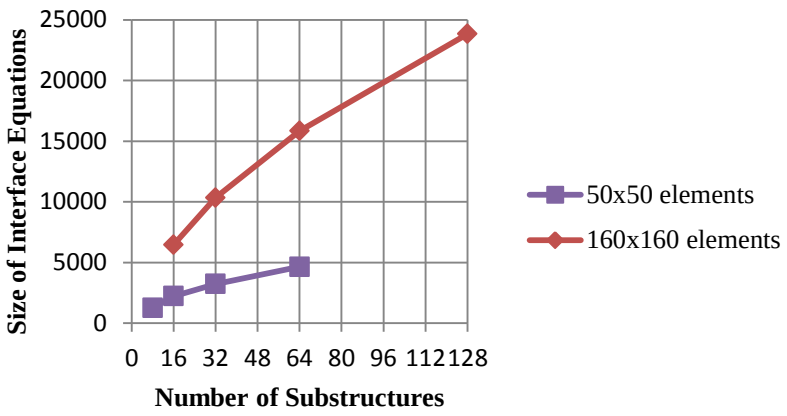


Figure 3.18: Sizes of interface equations of both structures for various numbers of substructures

Sparse condensation is sequential within each substructure stiffness matrix. Because of this reason only way to parallelize this part is increasing the number of the substructures. On the other hand increasing number of substructures causes an increase in the size of the interface equations. There are two disadvantages of this situation which are:

- The increasing size of the interface equations causes an increase in the solution time of the interface equations
- The global memory of the graphical cards becomes insufficient for storing the large size of interface equations

CHAPTER 4

GPU MULTIFRONTAL SOLVER

4.1 Introduction

This chapter includes the detailed information about the multifrontal solution method and its implementation on GPU architecture. In the first section of this chapter main steps of the multifrontal algorithm, which are partitioning, local assembly, condensation, assembly and solution of the interface equations and the back substitution are presented. In the following section, the implementation of the multifrontal solution method on GPU architecture is given. Finally, the test problems are introduced and the results obtained from these tests are discussed.

4.2 The Multifrontal Algorithm

The flowchart of the multifrontal solution algorithm is illustrated in Figure 4.1. Similar to the multiple front algorithm the multifrontal solution algorithm involves the partitioning, local assembly sparse condensation, assembly and solution of the interface equations and sparse recovery steps. In addition to these steps dense condensation and dense back substitution steps are also part of the multifrontal algorithm. The algorithm initiates with partitioning of the structure into several substructures. Besides the division of the structure, the formation of an assembly tree is also completed in the partitioning step.

Figure 4.1: Multifrontal algorithm

Figure 4.1: Multifrontal Algorithm

In multiple front algorithm after the sparse condensation step all the substructures are assembled together forming the interface equations, however in multifrontal algorithm substructures are assembled in multiple steps. After the assembly and solution of the interface equations, the back substitution is also completed in multiple steps and the algorithm is finalized.

4.2.1 Partitioning

Besides the division of the structure into substructures, in multifrontal algorithm it is required to determine the assembly tree with multiple levels. In Figure 4.2 an assembly tree for a 4×4 meshed structure is shown.

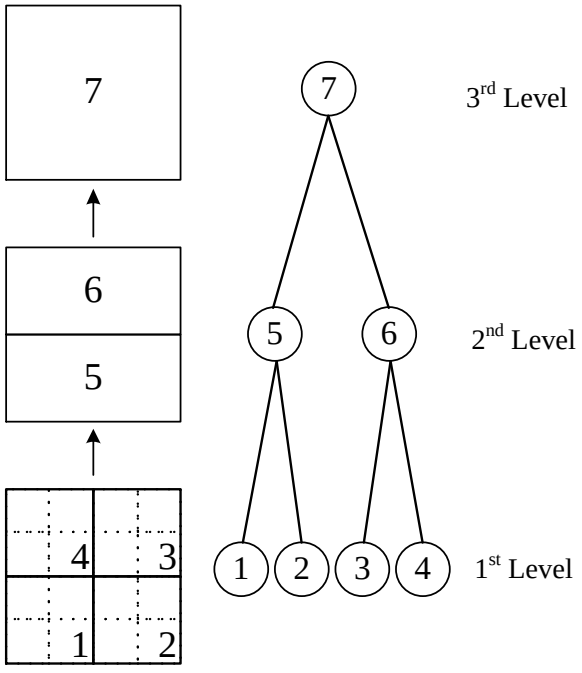


Figure 4.2: Assembly tree for a 4×4 square mesh with three levels

Finite elements are illustrated as the squares bordered with dash lines, on the other hand the substructures are illustrated as the shapes with bordered with solid lines. As

it can be observed from the figure, there are three levels in the assembly tree. In the first level there are four substructures each of which has four finite elements. These substructures are determined by the initial partitioning procedure. In the second level there are two intermediate substructures. These substructures are formed by the assembly of the Schur complements of the substructures in the first level. And in the third level, the Schur complements of the substructures in the previous level are assembled. As it was mentioned before, the substructures in the first level are determined by METIS [56] with multilevel partitioning algorithm in this study. On the other hand for the formation of the substructures in the intermediate level (2^{nd} Level), the information of which substructures should be matched to be assembled with each other forming the intermediate substructures is required.

In order to determine of the intermediate substructures of the assembly tree, an algorithm is implemented. The main goal of this algorithm is forming the smallest sized intermediate substructures, to decrease the number of dense condensation operations and the required amount of memory. Because of this reason the logic of the algorithm is based on matching the substructures which have the largest number of common DOFs. According to this algorithm the intermediate substructures are determined according to the three rules, which are as follows:

1. Only two child substructures are assembled to form their parent node.
2. The two child substructures which have largest number of DOFs in common are chosen to be assembled together from the unmatched substructures.
3. If a substructure has same number of common DOFs with multiple substructures, it is assembled with the substructure that has the smallest ID.

For a better understanding of the algorithm, formation of an assembly tree for a 10×10 meshed structure with eight substructures is illustrated in Figure 4.3 as an example. On the right side of Figure 4.3, the assembly tree of the structure is illustrated. The assembly tree has four levels. On the left side of each level of the assembly tree, a table showing the number of common equations between the

substructures and the number of equations of each substructure at that level is presented. In these tables, while the off diagonal values give the number of common equations between two substructures, the diagonal values give the number of equations of that structure. When the table for the first level is examined, it can be observed that the first structure has the largest number of common equations, which is 30, with the sixth substructure. Thus, they are matched together to be assembled and the cell showing the number of common equations between the first and the sixth substructure is colored to blue. Also in the assembly tree the lines, those are connecting the first and the sixth substructure to the ninth substructure, are colored to blue too. After matching the first pair, the unmatched substructure with the smallest ID, which is second substructure, is chosen. Second substructure has the largest number of common equations with the fifth substructure. Consequently, the second and the fifth substructures are matched to be assembled together. The same procedure is repeated and all the substructures in the first level are matched and assembled. After the substructures in the second level are determined, the same table is formed for the substructures in the second level. According to the number of the common equations between the substructures, they are matched together to be assembled. This procedure continues until the last structure is formed. As a result, the formation of the assembly tree is completed.

The 160×160 meshed structure with 16 substructures is assembled according to this algorithm in Figure 4.4. The substructures in the each level of the assembly tree are presented with different colors.

	SS
	15
SS **15**	84

	SSTR.	
	13	**14**
SSTR. **13**	174	84
14	84	168

	SUBSTR.			
	9	**10**	**11**	**12**
SUBSTR. **9**	108	48	42	6
10	48	114	6	48
11	42	6	102	42
12	6	48	42	108

	SUBSTR.							
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
SUBSTR. **1**	138	0	18	0	18	30	6	18
2	0	138	0	0	30	0	24	0
3	18	0	162	18	0	0	24	30
4	0	0	18	120	0	0	30	0
5	18	30	0	0	120	24	24	0
6	30	0	0	0	24	138	0	0
7	6	24	24	30	24	0	114	0
8	18	0	30	0	0	0	0	90

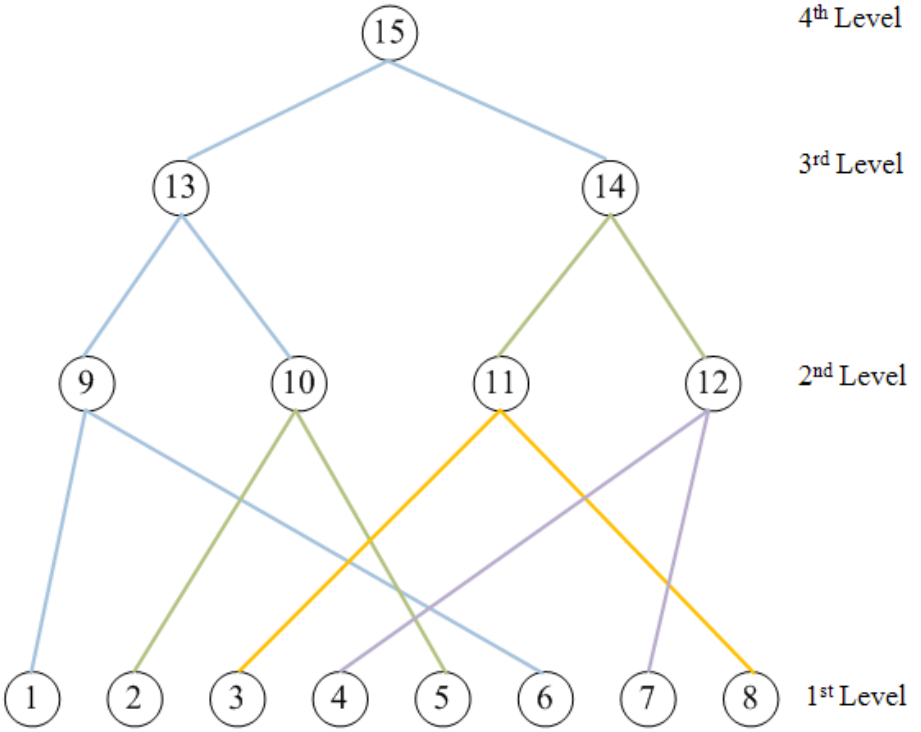


Figure 4.3: Assembly tree for a 10×10 square meshed structure with 8 substructures

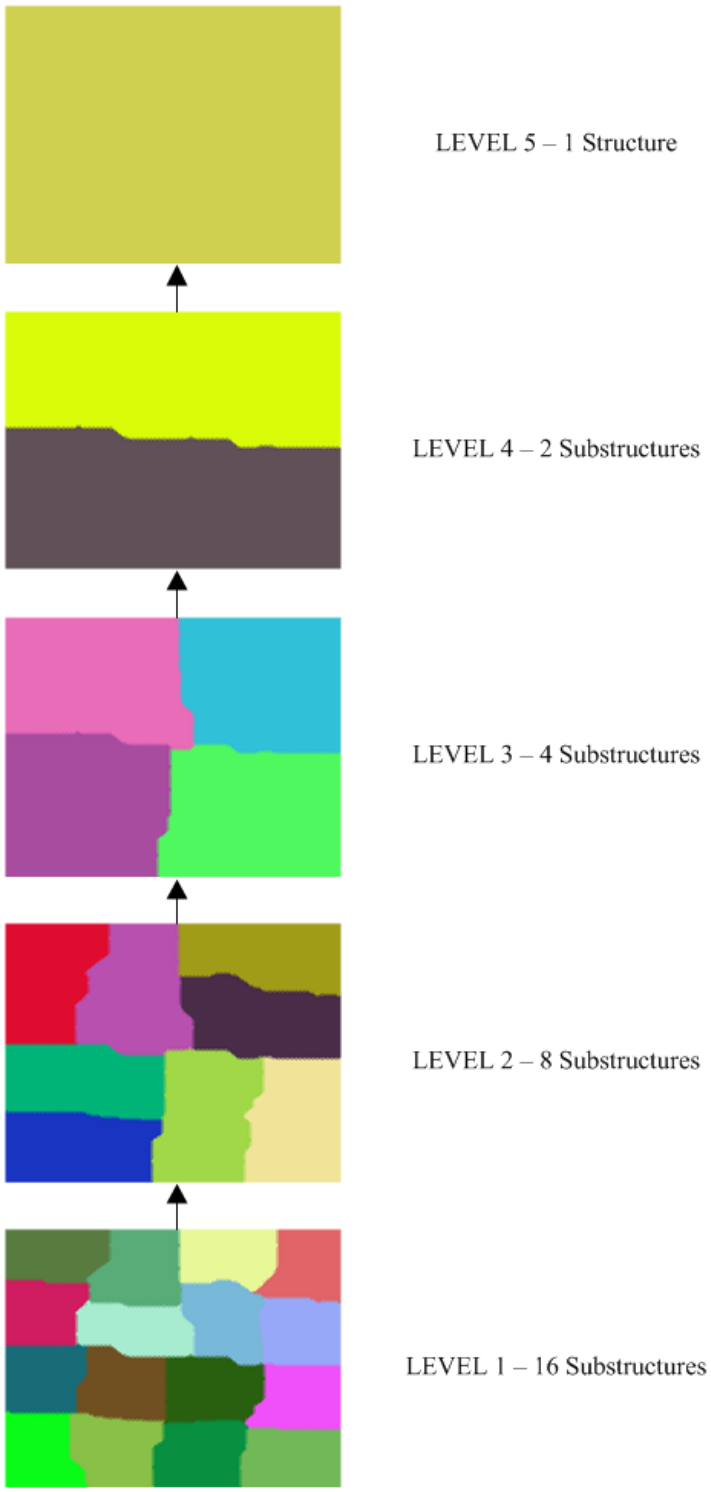


Figure 4.4: Assembly of 160×160 structure with 16 substructures

4.2.2 Sparse Condensation

The multifrontal algorithm is very similar to the multiple front algorithm. The sparse condensation, solution of interface equations and the sparse recovery parts are common for both of the algorithms. In the multifrontal algorithm, however, after the sparse condensation part, the intermediate substructures are still condensed to form higher level intermediate substructures by dense matrix condensation techniques. At this point, it is more convenient to give an example for better comprehension of the algorithm. The illustration of the example structure was given in Figure 4.5.

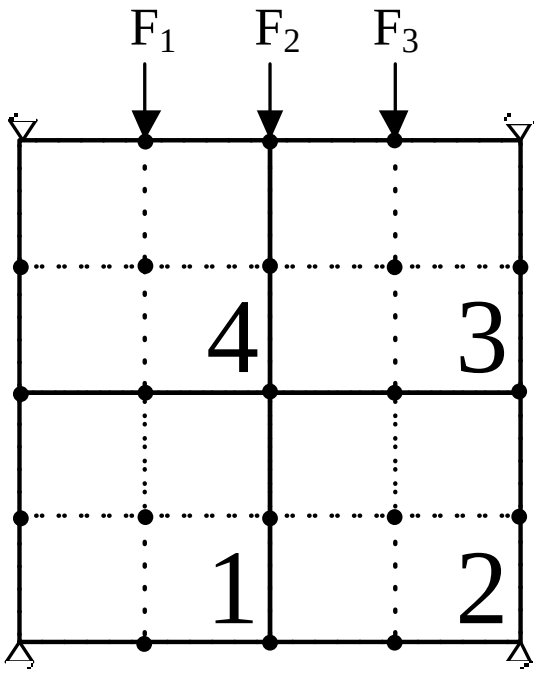


Figure 4.5: Example structure with 4×4 mesh

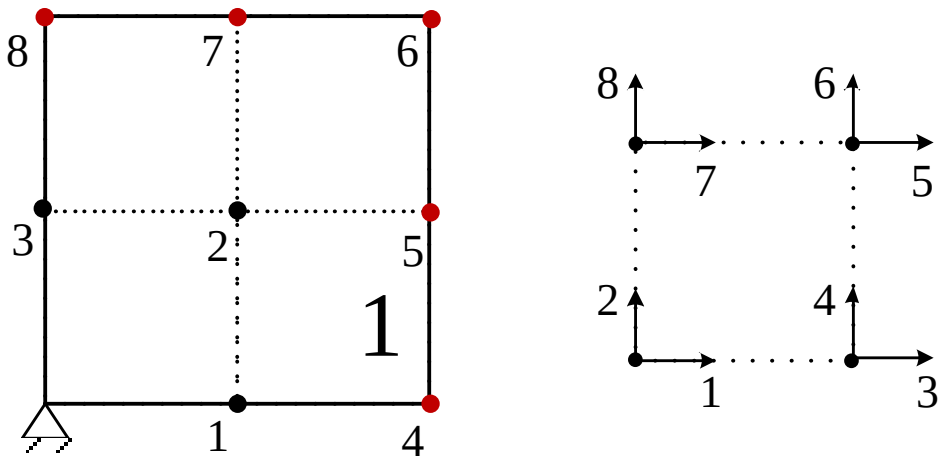


Figure 4.6: Substructure 1 and a bilinear 4-node membrane element

The assembly tree in Figure 4.2 is used for the multifrontal algorithm. In Figure 4.5 and Figure 4.6 the example structure and the first substructure are illustrated. The structure is divided into four substructures. Each substructure has 2×2 bilinear quadrilateral finite elements. Also the DOFs of the substructure and the node numbering can be seen in the Figure 4.4.

Multifrontal algorithm initiates with the sparse matrix condensation part as the multiple front algorithm. In this procedure, condensation of the substructures in the first level is completed. The internal equations of the substructures are condensed to the interface equations of the substructure. Condensation of the first substructure was illustrated in Figure 4.7. As it can be observed from the figure, the internal DOFs (DOFs 1-6) were condensed to interface DOFs (DOFs 7-16) of the substructure forming the Schur Complements of the substructures.

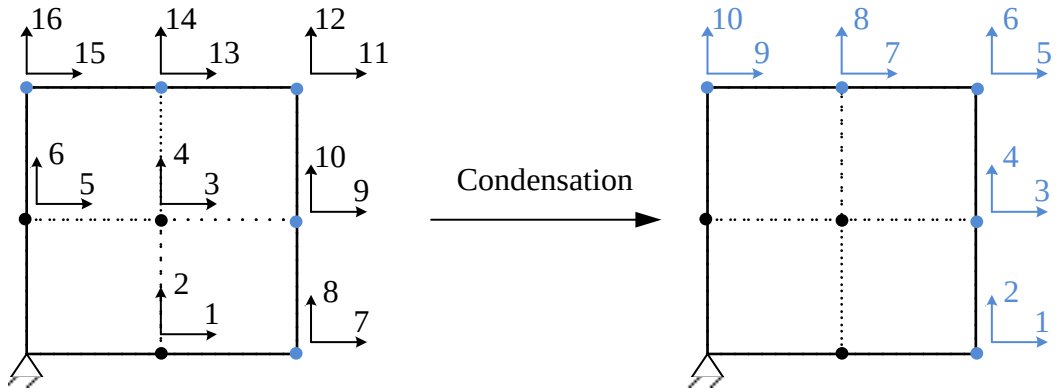


Figure 4.7: Condensation of the first substructure

4.2.3 Assembly of Intermediate Substructures

In multifrontal algorithm assembly of the Schur Complements of the first level substructures gives the intermediate substructures. Assembly of Schur Complements of the first two substructures and the last two substructures are illustrated in Figure 4.8 and Figure 4.9 respectively. As it is shown in Figure 4.8, the first and second substructures are assembled together forming the fifth substructure in the second level of the assembly tree. And in Figure 4.9, the third and the fourth substructures are assembled to form the sixth substructure. As it can be observed from the figures, DOFs 5-14 of both substructures are common, these DOFs contribute to the interface system and the DOFs 1-4 are internal DOFs, these DOFs are condensed to the interface DOFs.

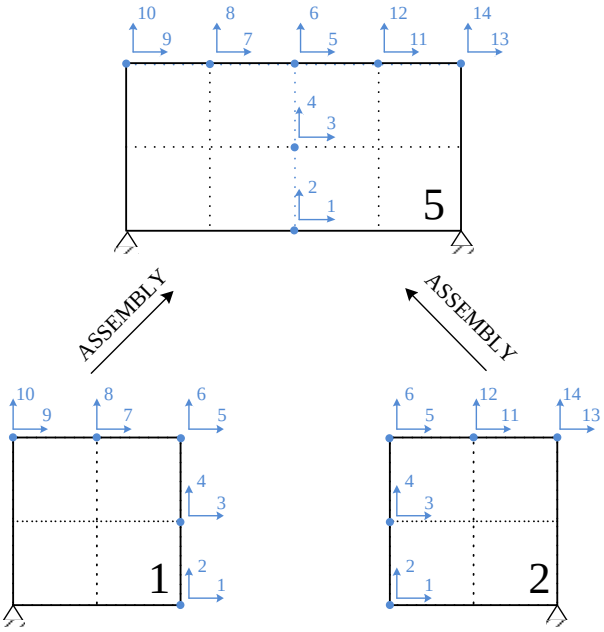


Figure 4.8: Assembly of the first two substructures

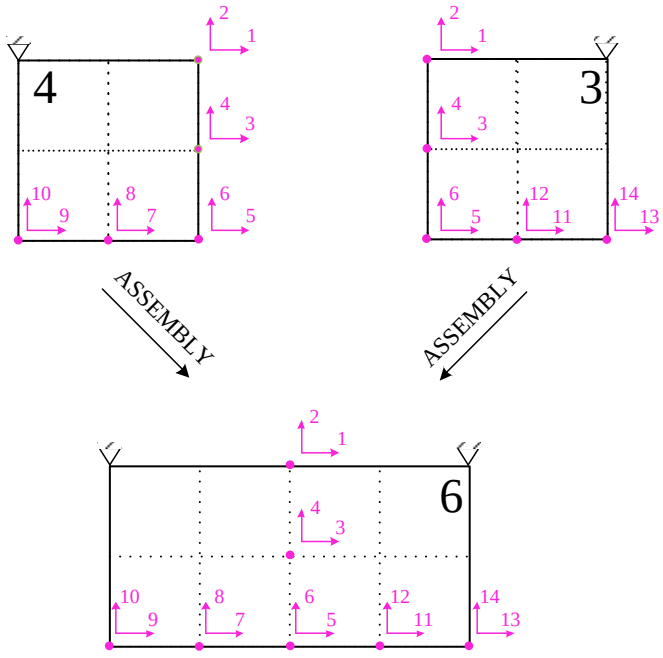


Figure 4.9: Assembly of the last two substructures

4.2.4 Dense Condensation

After assembly of the Schur Complements of the first level of substructures (Substructures 1-4) forming the intermediate substructures (Substructures 5-6), the internal equations of the intermediate substructures were condensed to the interface equations of the higher level substructures (Substructure 7). But this time the condensation procedure is handled by the dense matrix condensation algorithms, since the system of equations of the intermediate substructures is dense. The condensation procedures of the intermediate substructures are illustrated in Figure 4.10 and Figure 4.11. As it can be observed from the figures, instead of assembling the all of the substructures forming a single structure at once, the size of the system of equations is continued to be reduced at each intermediate levels of the assembly tree by condensation procedure.

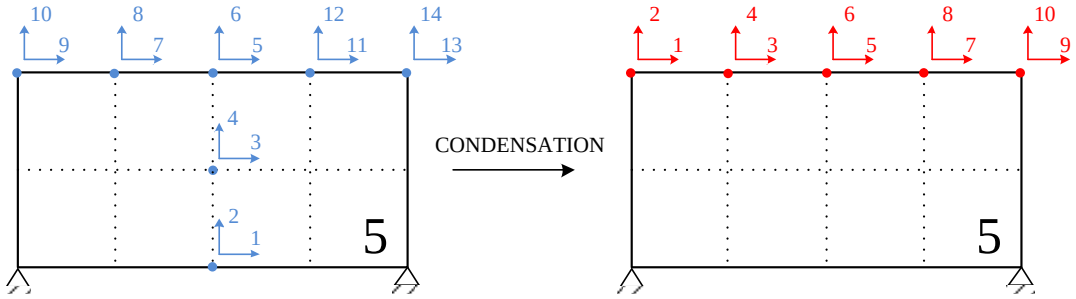


Figure 4.10: Condensation of the fifth substructure

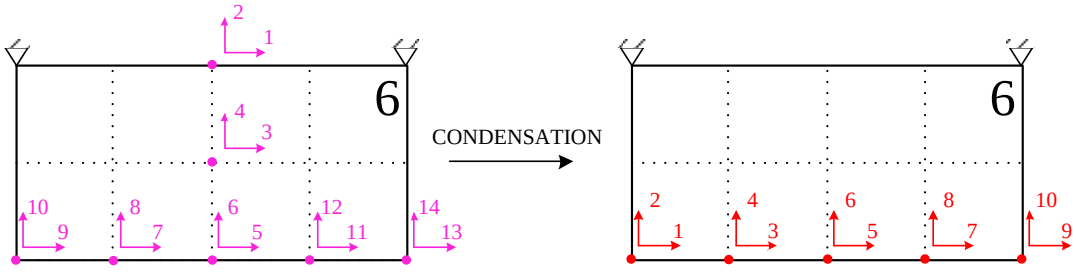


Figure 4.11: Condensation of the sixth substructure

As it is mentioned before, dense matrix algorithms are used for the condensation procedure of the intermediate substructures. Although the main operations are same with the sparse matrix condensation operations, the dense matrix condensation operations can be completed in parallel within the matrix. The reason of this situation is caused by storage and addressing of the data required for the operations. Since only non-zero terms are stored for storage of a sparse matrix, condensation procedure requires to be completed in sequential by use of indirect addressing in large amounts. However, in dense matrix condensation algorithms, since the data required can be accessed directly, the operations can be completed concurrently within a matrix. As a result, dense condensation operations can be completed more efficiently than sparse condensation operations. Since main theory is the same for both sparse and dense condensation algorithms, the detailed information about the mathematical formulae used in the condensation procedure was not repeated in this section. This information can be found in the previous chapter. However, the implementation of dense condensation algorithm is different from its implementation for the sparse systems, the detailed information about the GPU implementation of the dense condensation algorithm can be found in the following sections of this chapter.

4.2.4 Assembly and Solution of Interface Equations

After the condensation of the internal equations of all intermediate substructures, a final assembly operation has to be completed before the solution of the interface equations. With this assembly, the equation system of the uppermost structure (7^{th} structure) in the assembly tree is formed. This assembly operation is illustrated in Figure 4.12.

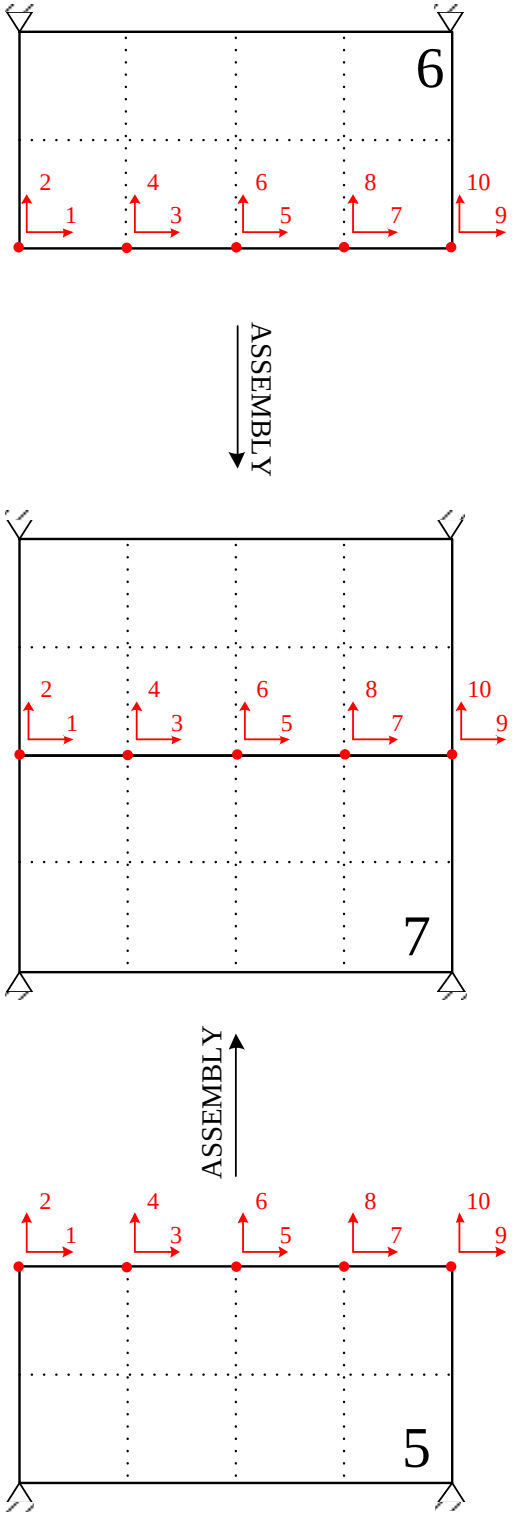


Figure 4.12: Assembly of the fifth and sixth substructures

As it can be observed from the figure, the Schur Complements of the fifth and sixth substructures are assembled forming the final interface equations. This system of equations is called “final”, because these are the equations which will be solved before the back substitution procedure. Note that, while the size of the final interface equation system is a 10×10 matrix in multifrontal algorithm, the size of the final interface equations system is 14×14 matrix in multiple front algorithm. Since the size of the example structure is not so large, the difference between the sizes of the interface equations systems formed in the two algorithms is not so significant. However this difference grows dramatically for the problems with large sizes. After the assembly of the final interface equations, this system of equations is solved with the same way in the multiple front algorithm.

4.2.5 Dense Back Substitution

The results obtained from the solution of the interface equations are sent back to the lower level substructures in the assembly tree by back substitution procedure. As it is mentioned before the assembly tree gives the condensation and assembly sequence of the substructures from the down to up, and the back substitution sequence from up to down. For each substructure with the back substitution operation the solution of the internal equations of these substructures are obtained. According to the assembly tree, the displacement values of the fifth substructure are transferred to the first and second substructures. And the displacement values of the sixth substructure are transferred to the third and fourth substructures. This step is repeated until the substructures in the bottom level of the assembly tree are reached. Same operations for sparse recovery presented in the previous chapter are used for dense back substitution step also. The subroutines differ from each other with the type of data access for the arithmetic operations. While indirect addressing has to be used for sparse systems, the required data can be accessed directly in the dense systems.

4.2.6 Sparse Back Substitution (Recovery)

With a last back substitution operation the displacements of the internal DOFs of the substructures in the first level (Substructures 1-4) are obtained, giving the solution of the whole structure. This step is exactly same with the one used in the multiple front algorithm. The detailed information about this step can be found in previous chapter.

4.3 GPU Implementation

In addition to multiple front algorithm, the multifrontal algorithm includes some additional subroutines for the dense condensation and dense back substitution operations for the intermediate substructures. Beside these additional subroutines the GPU implementation of the algorithm is same with the GPU implementation of the multiple front algorithm.

In Figure 4.13 the subroutines used in the GPU implementation of the algorithm were presented. As it can be observed the figure the most of the parts of the multifrontal algorithm are common with the multiple front algorithm. These parts were illustrated in the figure as black boxes. In addition to the multiple front algorithm, only the dense condensation and the dense back substitution parts are included in the multifrontal algorithm and these parts were illustrated in the figure as red boxes. In this section only GPU implementation of these additional subroutines will be presented, detailed information about the GPU implementation of the remaining parts can be found in the previous chapter.

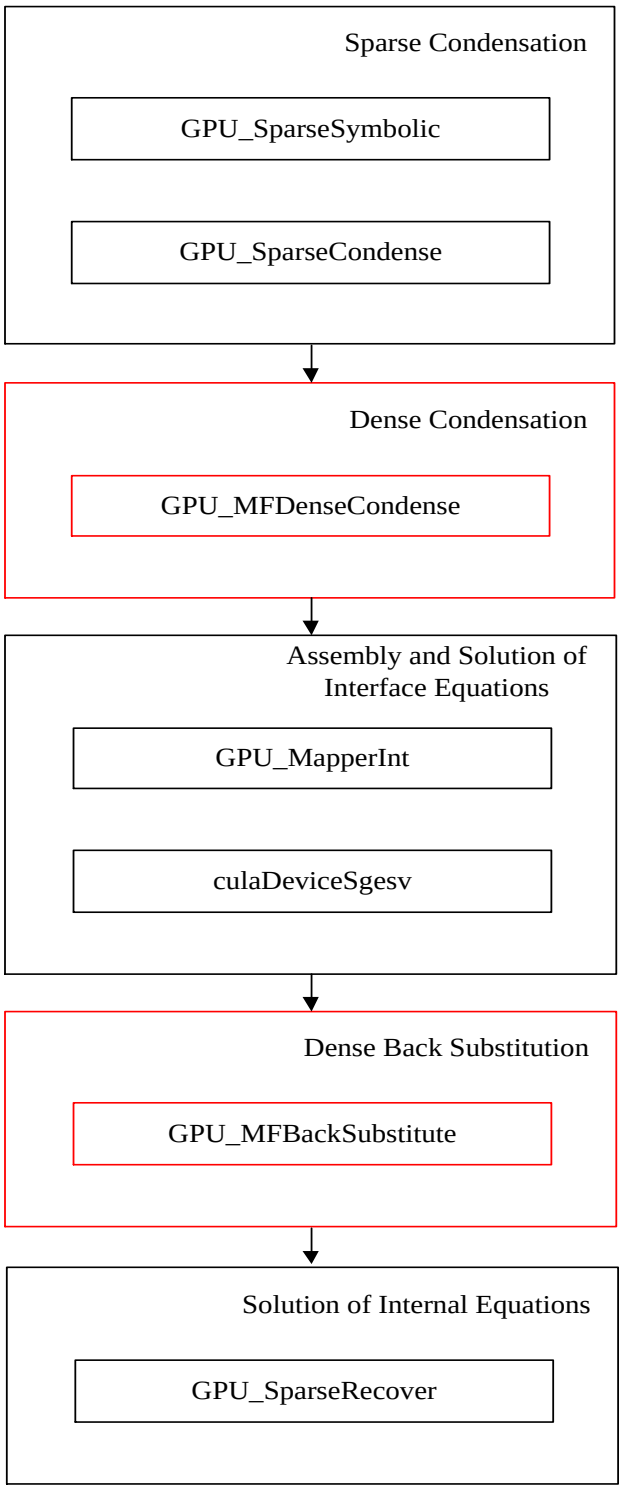


Figure 4.13: Subroutines used for GPU implementation of multifrontal algorithm

4.3.1 Dense Condensation

The dense condensation algorithm has two levels of parallelism. The first one is substructure level parallelism, meaning that each dense matrix belonging to a substructure is condensed in parallel by thread blocks assigned to them. In the second level of parallelism, each thread in a thread block is responsible for the calculation of the elements of a column in the dense matrix. Since the system is already partitioned to the substructures, the first level of the parallelism can be obtained easily by just assigning a thread block for each substructure in the same level of the assembly tree. However, to parallelize the condensation operations within a single matrix, the condensation operations should be divided into portions those can be executed by threads concurrently. In Figure 4.14 the sequential algorithm of condensation procedure is given. In the algorithm, A is the dense matrix and *nrows*, *ncols* and *SSchur* denote the number of rows, number of the columns and the size of the Schur Complement of the matrix A respectively. In the algorithm "loop i" defines the row that will be used for the elimination operation; the "loop j" defines the row that will be changed and the "loop k" changes the elements in each column in the j^{th} row of the matrix A. In the GPU implementation of the algorithm, the "loop k" is removed and the each thread assigned to a column. So the elements in different columns are calculated by different threads concurrently.

```
for i:=0 to (nrows-1)-SSchur     /* First Loop*/
       for j:=i+1 to nrows-1            /* Second Loop*/
              coeff:=-A[j,i]/A[i,i];
              for k:=i to ncols-1 /* Third Loop*/
                     A[j,k]:= A[j,k]+coeff*A[i,k];
              endfor
       endfor
endfor
```

Figure 4.14: Sequential dense condensation algorithm

An illustration of the GPU implementation of dense condensation of a single dense matrix is shown in Figure 4.15, Figure 4.16 and Figure 4.17 for each step of the first loop. In these figures the condensation operations of a 5×5 matrix with a 2×2 Schur Complement are presented. The rows and columns of the matrix were bordered with black solid lines whereas the Schur complement of the system is bordered with dashed lines. The threads used in the algorithm are shown above the matrix and denoted with "T" letter. In this example 4 threads are used in the condensation operations. According to the size of the matrix and the Schur complement *nrows*=5, *ncols*=5 and *SSchur*=5, so the first loop continues from zero to two for this example. In Figure 4.15 the variable *i* in the first loop equals to 0, so the elements in row 0 and column 0 are used for the calculation of the *coeff* variable. This row and the column are colored in blue. The elements on the right side of the column 0 and below the row 0 was calculated by the threads. Since the stiffness matrices are symmetric only upper triangular parts of the matrices are stored in this algorithm. Because of this reason in Figure 4.15, first thread (T0) updates the element in row 1 and column 1, the second thread (T1) updates the element in rows 1-2 in column 1, the third thread (T2) updates the elements in rows 1-3 in column 3 and finally the fourth thread (T3) updates the elements in rows 1-4 in column 4.

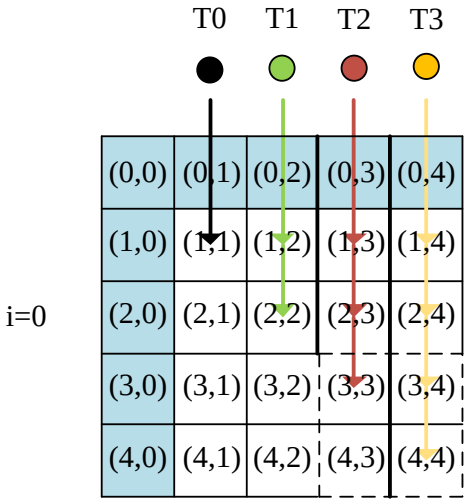


Figure 4.15: The condensation operations when *i=0*

Before initiating the condensation operations in the second step of the first loop, it is required that all of the threads completed their calculations because of this reason a barrier function is needed. Since all threads belong to the same thread block, __synchtreads() function can be used for synchronization of the threads. In Figure 4.16 the condensation operations completed when i=1 are illustrated. As it can be observed from the figure the first thread becomes idle, since calculation of all elements in the column 1 is completed in the previous step.

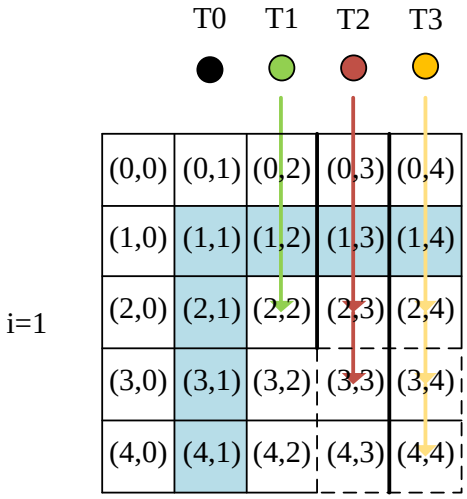


Figure 4.16: The condensation operations when *i=1*

The condensation operations completed when i=2, are illustrated in Figure 4.17. At this step the first and second threads become idle and the third and fourth threads calculate the elements in the Schur complement of the matrix. Note that the first loop stops at the row and column where the Schur Complement begins. As a result, the condensation of the matrix is completed with the end of this step.

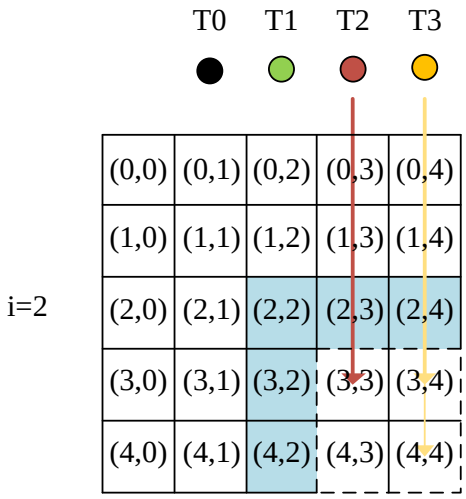


Figure 4.17: The condensation operations when *i=2*

There are some disadvantages of the algorithm. First, since each thread block is responsible for the calculation of a single matrix, larger number of matrices allows execution of larger number of threads. Because of this reason algorithm performance is expected to be higher in the lower levels of assembly tree where the number of matrices to be condensed is larger than the higher levels. Another disadvantage of the algorithm is caused by the column-wise algorithm implemented for condensation procedure of a single matrix. As it is mentioned before, during the condensation procedure a significant portion of the sources becomes idle. Besides, these disadvantages, the algorithm's ease of implementation and requirement for less memory due to the storage of only upper triangular part of the matrix, are important advantages and they are reasons for the implementation of this algorithm.

Since the size and the number of the matrices for large structures become too large, the shared memory, which has a high data transfer rate, cannot be used for the storage of the data. Because of this reason global memory is used for the storage of the data required for the dense condensation operations.

4.3.2 Dense Back Substitution

The back substitution of a system is sequential by nature for single right hand side vector. Furthermore, it is observed from the test results obtained in the previous chapter that the effect of back substitution part is not so significant on the overall performance of the solver. Because of these reasons back substitution operations calculated in parallel in substructure level but sequential within a system of equations of a substructure.

4.4 Test Problems and Results

The performance of the multifrontal algorithm is tested by 160×160 elements. The solution time values for various numbers of substructures obtained from GTX 275, GTX 580 Amp and Tesla C2050 are presented in Figure 4.18, Figure 4.19 and Figure 4.20 respectively. Note that, the acronym MPF denotes multiple front algorithm and MF denotes the multifrontal algorithm in these figures.

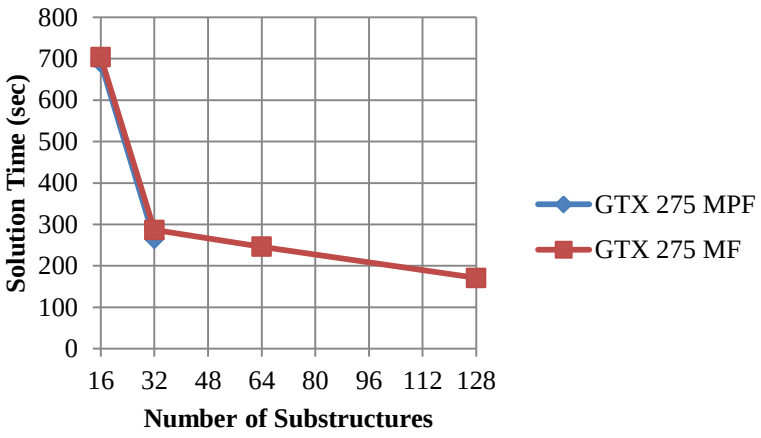


Figure 4.18: Solution time of the structure with 160×160 elements with GTX 275

As it can be observed from the Figure 4.18, the structure can be solved by multiple front algorithm only when the number of substructures is small enough for the storage of the interface equations in the GPU memory. On the other hand since the size of the interface equations becomes smaller in the multifrontal algorithm, the structure can be solved with 16, 32, 64 and 128 substructures. Since the sparse condensation time decreases with the increasing number of the substructures, the smallest solution time value is obtained from the solution of the structure with 128 substructures by using the multifrontal algorithm.

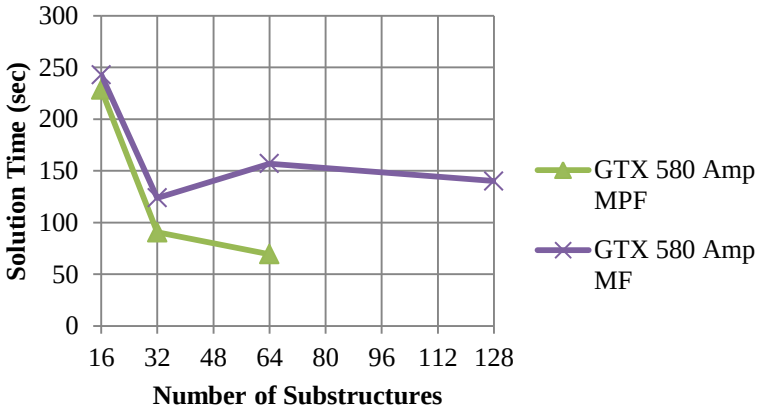


Figure 4.19: Solution time of the structure with 160×160 elements with GTX 580

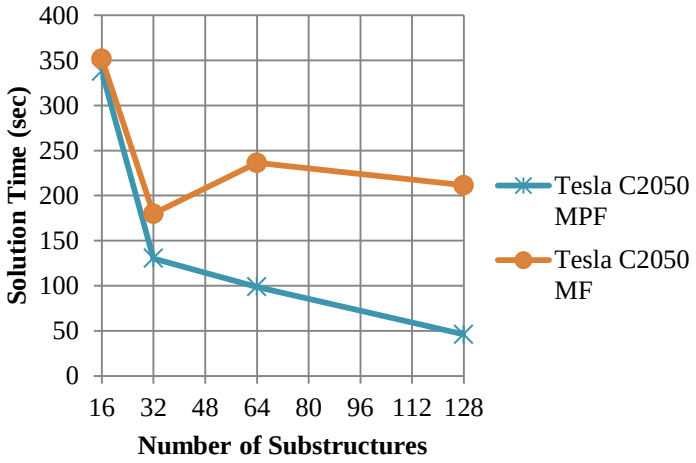


Figure 4.20: Solution time of the structure with 160×160 elements with Tesla C2050

The test results of solution of the same structure by using GTX 580 Amp and Tesla C2050 were presented in Figure 4.19 and Figure 4.20 respectively. When the two algorithms were tested for the same number of substructures, the multiple front algorithm solves the system in a shorter time period than the multifrontal algorithm for both GPUs. To find the reason of this situation the effect of each solution step to the total solution time was investigated. The ratios of time values passed in the solution steps to the total solution time of the 160×160 with 128 substructures were presented in **Error! Reference source not found.**

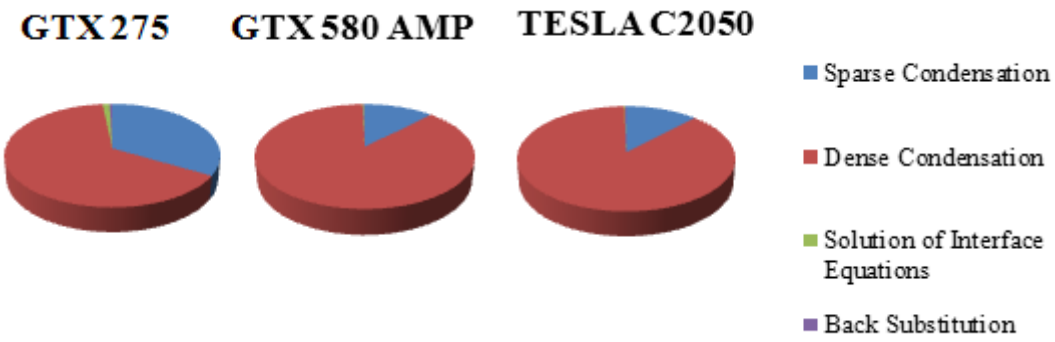


Figure 4.21: The effect of solution steps to the solution time of structure with 160×160 elements with 128 substructures

As it can be observed from Figure 4.21, dense condensation and sparse condensation parts are the most time consuming parts for all GPUs. For a better understanding of the bottlenecks, factors affecting the performance of these parts were examined. For this purpose, NVidia Compute Visual Profiler [58] was used as profiling tool to analyze the performance of the algorithm for the solution of 160×160 meshed structure with 128 substructures. The results for the analysis of the performance of the algorithm in GTX 580 AMP can be summarized as follows:

- The execution of GPU_SparseCondense kernel takes 13.32% and GPU_MFDenseCondense kernel takes 85.33% of execution time of whole algorithm, which is 98.65% in total. The time required for data transfer between host and device is 0.02%.

- In sparse condensation step, since the parallelism is limited with the substructure level, number of substructures is too small for the use of full performance of the GPU. In addition, due to the size of the matrices and the storage algorithm of the sparse systems, instead of shared memory, global memory with lower throughput and higher latency (400-800 cycles) [59] had to be used. This latency could not be overlapped due to the small number of threads. Moreover, threads are executed in groups called warps. If the threads in the same warp do not execute the same thread instructions, it causes significant performance loss. This situation is called divergent branching [58]. The divergent branching can be caused by if-else statements or loops with different start and end conditions. The ratio of different thread instructions those are executed by the threads of the same warp to the all thread instructions in that kernel is defined as control flow divergence [58]. This ratio is an indicator for performance loss due to divergent branching and it should be as low as possible. Since each thread completes the different number of operations of different substructures in the sparse condensation step, this ratio is 96.88 % showing that this step was suffered from this kind of performance loss. Additionally a performance loss occurred due to the indirect addressing in this step. As a result of these performances losses, the achieved global memory throughput was 10.47 GB/s where it equals to 5.32% of the peak performance global memory throughput of the graphic card showing that the performance of sparse condensation step is low.

- The performance of the dense condensation step varies with the level of the assembly tree. The ratio of dense condensation time of each level to the total dense condensation time for the 160×160 meshed structure with 128 substructures is presented in Figure 4.20. For the structure partitioned to the 128 substructures, the assembly tree has $\log_2(128)+1=8$ levels, where the first level has 128 substructures and the last level has one substructure. As it was mentioned before, the dense condensation procedure is used at intermediate levels of the assembly tree (Levels 2-7). Because of this reason, there are six

levels starting from Level 2 and ending with Level 7 in the y axis of Figure 4.22.

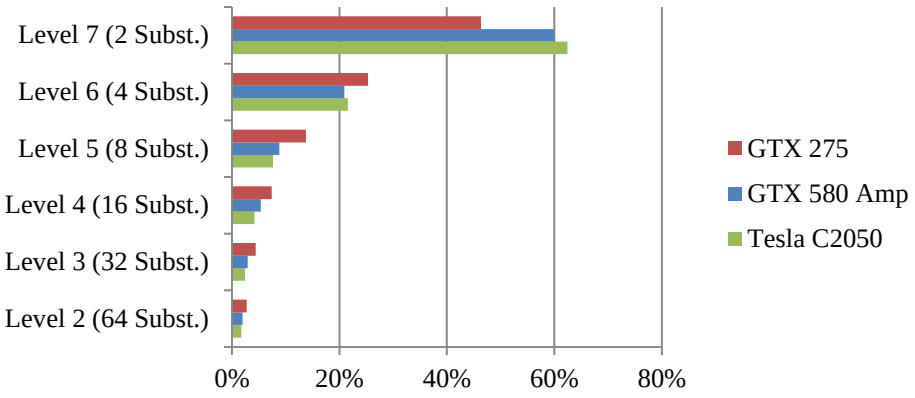


Figure 4.22: The ratio of dense condensation time values to total dense condensation time

According to the Figure 4.22 the dense condensation time values increase with the increasing level in the assembly tree. Since in upper levels of the assembly tree, the number of substructures decreases; so smaller number of thread blocks can be allocated. As a result at the upper levels, smaller number of threads can be executed concurrently than the lower levels. Another reason for the lower performance at the upper levels of the assembly tree is caused by the divergent branching. Due to the increase in the size of the matrices, the percentage of control flow divergence reaches to 92-97% for the dense condensation step in upper levels of the assembly tree. When the ratio of achieved global memory throughput to the peak global memory throughput is taken as a performance indicator, it was observed that this ratio becomes 1-5% for the top two levels in the assembly tree, whereas it is 50% at the lowest level.

Although the sparse condensation algorithm is not so flexible for changes to improve the GPU utilization, the dense condensation step may be improved by redeveloping

the algorithm by taking the memory access and thread instructions of the threads in the same warp into account. With a new algorithm using shared memory and allowing execution of larger number of threads also in the upper levels of the assembly tree may yield a significant performance gain. However, besides these changes, to improve the performance of the solver, an important performance gain can be obtained with the modification of the algorithm used for forming the assembly trees.

Two different assembly trees for the same structure, which have eight substructures initially, are presented in Figure 4.23. The assembly tree on the left side of the figure is formed according to the algorithm mentioned in the section 4.2.1. When the algorithm forming this assembly tree is used in the tests, it is observed that the dense condensation of the upper levels takes too much time. Thus, instead of assembling the substructures two by two, all substructures at a level, where interface system is small enough to be stored in GPU memory, can be assembled to final substructure as it is shown on the right side of the figure. In this assembly tree, the eight substructures in the first level are assembled forming the four substructures in the second level, but the substructures in the second level are assembled forming the final substructure. Thus the new assembly tree does not include upper levels of the previous assembly tree where the performance of the dense condensation is the least.

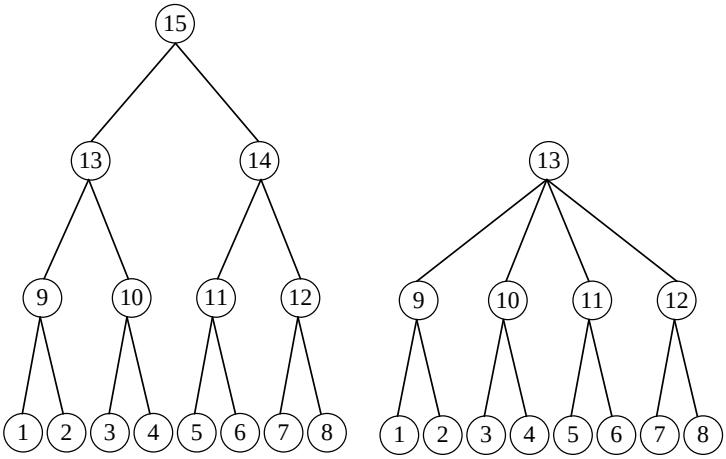


Figure 4.23: Two different assembly trees for the structure with eight initial substructures

The shortest solution time values of the structures with 160×160 elements for all algorithms are presented in Figure 4.24. In the figure acronym "MLMF" denotes Multi-Level Multifrontal method referring to new algorithm for formation of the assembly trees. As it can be observed from Figure 4.24, the smallest solution time values were obtained from the MLMF method. While the performance difference between this algorithm and the multiple front algorithm is significant for GTX 275 and GTX 580 Amp, this difference is very small for Tesla C2050. Because, the system with 128 substructures cannot be solved with multiple front algorithm with GTX 275 and GTX 580 Amp due to insufficient GPU memory. But with the multi frontal algorithm and MLMF algorithm the system partitioned into 128 substructures can be solved, consequently the sparse condensation time is much smaller than the multiple front algorithm. However, the performance of the multifrontal algorithm is negatively influenced by the low performance dense condensation of substructures at upper levels in the assembly tree. But in MLMF algorithm sparse condensation time is much shorter than the multiple front algorithm, in addition to this, dense condensation time is significantly less than the multifrontal algorithm. On the other hand, since the structure partitioned into 128 substructures can be solved with multiple front algorithm, the sparse condensation time values for all of the algorithms are same with each other. So small difference between the solution time of the multiple front algorithm and the MLMF algorithm is caused by the decrease in the

solution time of the interface equations according to the decrease in the interface equations size due to the dense condensation in the MLMF algorithm.

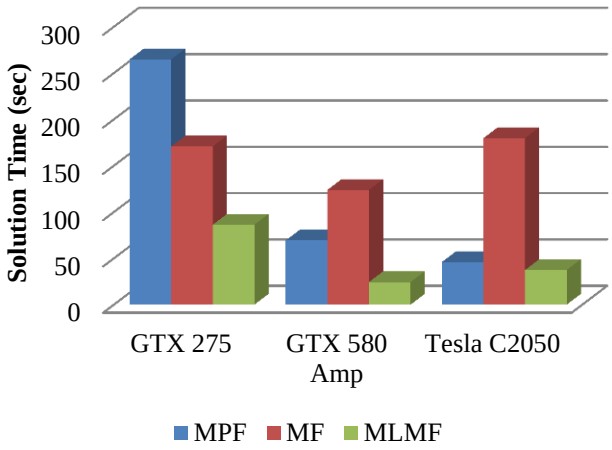


Figure 4.24: The shortest solution times obtained from the solution of 160×160 meshed structure with the three methods

As a final test a larger system with 200×200 elements is solved with GTX 580 Amp and Tesla C2050. Since the size of the global memory of GTX 275 is insufficient for the problem. It is not tested with GTX 275. Moreover the results are compared with the solutions obtained from Intel Core2 Quad 2.5 GHz clock time computer by using MUMPS [38], a software package for solution of sparse systems by using multifrontal algorithm. The results are presented in Figure 4.25.

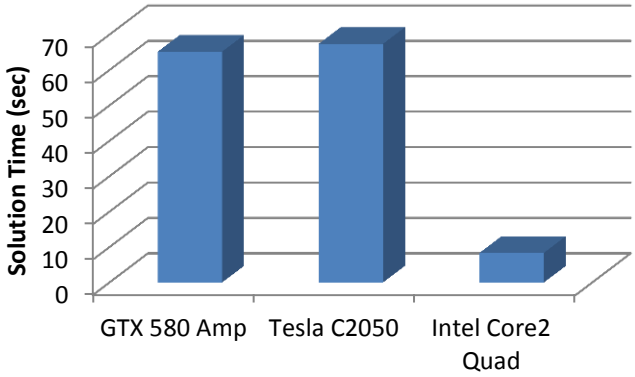


Figure 4.25: The solution time of the 200×200 structure with different architectures.

In Figure 4.25 the results are obtained from the structure with 128, 256 and 4 substructures in GTX 580 Amp, Tesla C2050 and Intel Core2 Quad respectively. Although the performance of the solver is increased significantly with the improvement in the assembly tree, the performance of the solver is lower than the performance of solver runs on CPU.

CHAPTER 5

CONCLUSION

5.1 Summary

The solution of the linear system of equations has an important role for most of the engineering problems. And it is one of the core aspects of FEA software. Since the large amount of arithmetic operations are required for the solution of these systems, the influence of the solution of linear equations on the performance of the software is very significant. As a result, an increase in the performance of the solution of the linear equations is an important source of performance gain for the FEA software.

In recent years, the increasing demand for performance in the game industry caused significant improvements on the performances of GPUs. With their massive floating point operations capability, they became attractive sources of performance for the general purpose programmers. Because of this reason, in this study GPUs are chosen as the target hardware to develop an efficient parallel direct solver for the solution of the linear equations obtained from FEA. To achieve this goal two substructure based algorithms, multiple front and multifrontal algorithms were implemented.

Besides the solution of the equations, also assembly of the stiffness matrices and force vectors of the structures may be very time consuming for large systems. At this condition, instead of assembling the whole system, dividing the system into substructures, and making the necessary calculations for each substructure

concurrently becomes much more efficient. Because of this reason the frontal methods are appropriate for FEA. The multiple front and the multifrontal algorithms are two common examples of the frontal methods.

In this study first multiple front algorithm was implemented. The multiple front algorithm can be summarized in tree steps. The first step is the sparse condensation of the substructure equations to obtain the Schur Complement of each substructure. The second step is the assembly of the Schur Complements of all substructures forming a larger system of equations called interface equations and the solution of this system. The final step of the algorithm is the back substitution procedure. With this procedure the internal equations of the substructures are obtained. As a result the solution of the system is completed.

The multiple front algorithm was tested for different size of structures with different number of substructures. According to the results obtained from the tests, sparse condensation part is the most time consuming part of the solution for the smaller number of substructures. As the number of substructures increases, the time passed during the sparse condensation part decreases, whereas the time elapsed during the solution of the interface equations increases. As a result, when the structure is divided into larger number of substructures to increase the performance of the solver by speeding the sparse condensation up, the performance of the solver is limited by the performance of the interface solution. Furthermore, increasing size of the interface equations requires much more memory space than the GPUs have. So the solution of the system becomes physically impossible due to the insufficient memory of the GPU.

The multifrontal algorithm avoids the disadvantages of the multiple front algorithm by completing the condensation procedure in multiple steps. After the sparse condensation procedure in multifrontal algorithm instead of assembling all of the

Schur Complements of the substructures forming a single large interface system, multiple smaller interface systems are formed. The numbers of the equations of these systems are reduced by efficient dense condensation algorithms. At the end, a smaller size interface equation system is obtained. So the time required for the solution of the interface system becomes shorter than the multiple front algorithm.

The multifrontal algorithm was tested for different size of structures with different number of substructures. According to the results obtained from the tests, sparse condensation part is the most time consuming part of the solution for the smaller number of substructures. As the number of substructures increases, the time passed during the sparse condensation part decreases, whereas the time elapsed during the dense condensation part increases. When the cause of this increase was investigated, it was observed that the dense condensation procedure became inefficient at the higher levels of the assembly tree. For this reason an improvement in the algorithm for the formation of the assembly tree was implemented and the higher levels of the assembly tree were reduced. With this improvement the performance of the multifrontal algorithm increased significantly and the fastest solutions obtained from the multifrontal algorithm. However when the results were compared with one of the optimized parallel sparse solvers running on CPU, it was observed that the solution obtained from the CPU is much faster than the sparse solvers implemented in this study.

5.2 Conclusion

In the literature, the studies about the solution of linear sparse system of equations on GPU architecture are mainly based on iterative methods or hybrid algorithms using both GPU and CPU algorithms. On the other hand, in this study algorithms were developed for only GPU architecture by use of direct methods for solution of sparse systems. For this purpose multiple front and multifrontal algorithms were

implemented on GPU architecture. Based on the results of the performance of implementation of these algorithms on GPU architecture, the following observations are obtained:

- The multiple front algorithm is limited with the sparse condensation step. For better performance, number of substructures or performance of the sparse condensation step should be increased.

- Increasing number of substructures yields a significant performance gain in the multiple front algorithm. However, GPU memory size becomes insufficient for storing the interface equations system of large number of substructures such as 256, 512 and 1024.

- Performance of the sparse condensation part is low. Because:
    - Only one thread can be used for the arithmetic operations of condensation procedure of a substructure. So the number of threads executed concurrently is equal to the number of substructures. This number is not sufficient enough to fully utilize computational source of GPU and hide the latency caused by data transfer.
    - Since shared memory cannot be used due to the size of the matrices and the sequential nature of the algorithm, global memory with lower throughput and higher latency should be used. Moreover, indirect addressing causes additional performance loss.
    - In GPU architecture, threads are executed in groups called warps. The execution of different branches in the code sample by the threads in the same warp causes a significant performance loss. Since each thread completes the condensation of different substructure, the number of operations, the start and end conditions of the loops vary from one thread to another causing a significant performance loss.

- Since sparse condensation algorithm is not so flexible, it is very hard to change it to optimize the GPU utilization. As a result, because of the reason

that, performance of the multiple front algorithm is limited with the number of substructures due to the size of the GPU memory, multifrontal algorithm is much more promising for efficient solution of sparse systems.

- The performance of the multifrontal algorithm, is limited with the sparse condensation and dense condensation steps. Multifrontal algorithm allows partitioning the system to large number of substructures, since size of interface equations reduced by dense condensation procedure. So the performance of sparse condensation step can be increased in multifrontal algorithm by increasing the number of substructures.

- The influence of algorithm for formation of the assembly tree is significant for the multifrontal algorithm. Small changes in this algorithm may yield important performance gain.

- The performance of the dense condensation is the core aspect of the multifrontal algorithm. The overall performance of the dense condensation algorithm is low. Because:
    - The performance of the dense condensation algorithm decreases significantly with decreasing number of matrices to be condensed concurrently. Since for each matrix, a single thread block is used, the total number of the threads executed concurrently equals to (number of threads in a thread block)×(number of substructures). Thus, for small number of substructures, the computational source of GPU cannot be fully utilized.
    - Divergent branching increases with the large size of matrices, causing a significant performance loss.
    - Use of global memory instead of shared memory decreases the performance of the dense condensation step.

5.3 Future Work

- **Improvement on Sparse Condensation:** The sparse condensation step limits the performances of the both algorithms. To increase the performance of this step, in the current implementation only way is to increase the number of the substructures, which causes new problems such as an increase in the amount of data to be stored in the GPU memory. For this reason it is expected that an efficient sparse condensation implementation, which can use threads concurrently also within a substructure, may yield a significant performance gain. However the nature of the algorithm is not so flexible for GPU optimization. For this reason, completion of the sparse condensation part on CPU may be an important alternative.

- **Improvement on the algorithm for formation of the assembly tree:** Since the assembly tree directly affects the size and the number of the operations completed in the interface equations, an improvement in this algorithm yields important changes in the performance of the solver. The effect of the shape of assembly trees should be investigated for better performance. However, overall performance gain due to the improvement on this algorithm will be limited, unless either one of the performance of the sparse condensation or dense condensation steps is improved.

- **Improvement on Dense Condensation:** A significant performance gain is expected if the following modifications can be completed.

    - Modify the algorithm allowing the use of flexible number of thread blocks for condensation process of any number of matrices. Thus, performance loss due to inefficient utilization will be avoided.
    - Modify the algorithm allowing use of shared memory by partitioning the matrices into smaller parts those fit into shared memory.
    - Modify the input matrices so the data accessed by the threads in the same warp is stored adjacently.

REFERENCES

1 Wilson, G. V. The History of the Development of Parallel Computing. *url:http://ei.cs.vt.edu/~history/Parallel.html.* (visited on 03/20/2011).

2 Anthes, G. The Power of Parallelism. *url: http://www.computerworld.com/s/ article/65878/The_Power_of_Parallelism.* (visited on 03/20/2011).

3 Sanders, J. and Kandrot, E. CUDA by example.

4 Kirk, D. and Hwu, W. Programming Massively Parallel Processors: A Hands-on Approach. *Morgan Kaufmann Publishers.* (2010).

5 Sutter, H. and Larus, J. Software and the concurrency revolution. *ACM Queue.* 3(7), 54-62 (2005).

6 AMD. Multicore processing: Next evolution in computing. *AMD White Papers.*(2005).

7 Held, J., Batista, J. and Koehl, S. From a Few Cores to Many: A Tera-scale Computing Research Overview (Intel White Paper). *Intel Corporation*. (2006).

8 Hwu, W. W., Keutzer, K. and Mattson T. The concurrency challenge. *IEEE Design and Test of computers.* July/August, 312-320 (2008).

9 TOP500. TOP500 Super Computer Sites. *url: http://www.top500.org/* . (visited on 03/20/2010).

10 Sottile, M. J., Mattson, T. G. and Rasmussen, C. E. Introduction to Concurrency in Programming Languages. *Chapman and Hall.* (2010).

11 OpenMP. Open Multi Processing. *url: http://openmp.org* . (visited on 03/20/2011).

12 Message Passing Forum. The Message Passing Interface Standard. *url: http://www.mcs.anl.gov/research/projects/mpi.* (visited on 03/20/2011).

13 Message Passing Interface Chameleon 2 Library. *url: http://www.mcs.anl.gov/research/projects/mpich2.* (visited on 03/20/2011).

14 PVM. Parallel Virtual Machine. *url: http://www.csm.ornl.gov/pvm/* . (visited on 03/20/2011).

15 Gropp, W. and Lusk, E. PVM and MPI are completely different. (1998).

16 Nvidia. CUDA Zone. *url: http://www.nvidia.com/object/cuda_home_new. html.* (visited on 03/20/2011).

17 Nvidia. CUDA Zone. *url: http://www.nvidia.com/object/cuda_home_new. html.* (visited on 03/20/2011).

18 Nvidia CUDA. Nvidia CUDA C Programming Guide Version 3.2 (2010).

19 Khronos. OpenCL, The Open Standard for Parallel Programming of Heterogeneous Systems. *url: http://www.khronos.org/opencl/* . (visited on 03/20/2011).

20 Karimi, K. , Dickson, N. G. and Hamze., F. A Performance Comparison of CUDA and OpenCL. *In: arXiv.org* (2010).

21 Microsoft. Microsoft DirectX Developer Center. *url: http://msdn.microsoft. com/en-us/directx.* (visited on 11/14/2010).

22 Kurc, O. A Substructure Based Parallel Solution Framework for Solving Linear Systems with Multiple Loading Conditions. PhD thesis. Georgia Institute of Technology 2005.

23 Guney, M. E. High-Performance Direct Solution of Finite Element Problems on Multi-Core Processors. PhD thesis. Georgia Institute of Technology 2010.

24 Farhat, C., Crivelli, L. and Rous, F.X. Extending substructure based iterative solvers to multiple load and repeated analyses. *Comput. Methods Appl. Mech. Engrg.* 117, 195-209 (1994).

25 Bitzarakis, S., Papadrakis, M., and Kotsopulos, A. Parallel solution technique in computational structural mechanics. *Comput. Methods Appl. Mech. Engrg.* 148, 75-104 (1997).

26 Duff, I. S. and van der Vorst, H. A. Developments and trends in the parallel solution of linear systems. *Parallel Computing*. 25, 1931-1970 (1999).

27 Dongarra, J. J., Duff, I. S., Sorensen, D. C., and van der Vorst, H.A. Numerical linear algebra for high performance computers. *Society for Industrial and Applied Mathematics.* (1998).

28 Ashcraft, C. and Grimes, R. SPOOLES: An object-oriented sparse matrix library. *Proceedings of the 9th SIAM Conference on Parallel Processing for Scientific Computing.* pp. 10 (1999).

29 Ashcraft, C. SPOOLES 2.2 : SParse Object Oriented Linear Equations Solver. *http://www.netlib.org/linalg/spooles/spooles.2.2.html*. (visited on 03/26/2011)

30 Li, X. S. An overview of SuperLU: Algorithms, implementation, and user interface. *ACM Trans. Math. Softw.* vol. 31, no. 3, pp. 302-325 (2005).

31 SuperLU. *http://crd.lbl.gov/~xiaoye/SuperLU/*. (visited on 03/26/2011).

32 Dobrian, F., Kumfert, G. and Pothen, A. The design of sparse direct solvers using object-oriented techniques. *Institute for Computer Applications in Science and Engineering (ICASE).* (1999).

33 Irons, B. M. A frontal solution scheme for finite element analysis. *Int. J. Numer. Methods Eng.* Vol. 2, pp. 5-32 (1970).

34 Duff, I. S. and Reid, J. K. The multifrontal solution of indefinite sparse symmetric linear. *ACM Trans. Math. Softw.* vol. 9, no. 3, pp. 302-325 (1983).

35 Gupta, A. and Joshi, M. WSMP: A high-performance shared- and distributed memory parallel sparse linear equation solver. *RC 22038, IBM Research Division* (2001).

36 Gupta, A. WSMP: Watson Sparse Matrix Package (Version 11.1.19). *http://www-users.cs.umn.edu/~agupta/wsmp.html*. (visited on 03/26/2011).

37 Amestoy, P. R., Duff, I. S., L'Excellent J.-Y. and Koster, J. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM J. Matrix Anal. Appl.* Vol. 23, no. 1, pp. 15-41 (2001).

38 MUMPS : A parallel sparse direct solver. *http://graal.ens-lyon.fr/MUMPS/* . (visited on 03/26/2011).

39 Chen, Y., Davis, T. A., Hager W. W. and Rajamanickam, S. Algorithm 8xx: CHOLMOD, Supernodal sparse Cholesky factorization and update/downdate. *TR-2006-005, University of Florida* (2006).

40 Davis, T. A. CHOLMOD, supernodal sparse Cholesky factorization and update/downdate. *http://www.cise.ufl.edu/research/sparse/cholmod/*. (visited on 03/26/2011).

41 Davis, T. A. Algorithm 832: UMFPACK V4.3---an unsymmetric pattern multifrontal method. *ACM Trans. Math. Softw.* Vol. 30, no. 2, pp. 196-199 (2004).

42 Davis, T. A. UMFPACK: Unsymmetric multifrontal sparse LU factorization Package. *http://www.cise.ufl.edu/research/sparse/umfpack/* (visited on 03/26/2011).

43 Gould, N. I. M., Scott, J. A. and Hu, Y. A numerical evaluation of sparse direct solvers for the solution of large sparse symmetric linear systems of equations. *ACM Trans. Math. Softw.* Vol. 33, no. 2, pp. 10 (2007).

44 Gupta, A. and Muliadi, Y. An experimental comparison of some direct sparse solver packages. *Proceedings of the 15th International Parallel & Distributed Processing Symposium.* (2001).

45 Bolz, J., Farmer, I., Grinspun, E. and Schröder, P. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. *ACM Trans.* Graph. 22(3) 917–924 (2003).

46 Krüger, J. and Westermann, R. Linear algebra operators for gpu implementation of numerical algorithms. *ACM Transactions on Graphics.* (TOG) 22(3) 908–916 (2003).

47 Buatis, L., Caumon, G. and Levy, B. Concurrent number cruncher an efficient sparse linear solver on the GPU. In: High Performance computation conference (HPCC). Springer lecture notes in computer sciences, vol 4782.

48 Couturier, R., Domas, S. Sparse systems solving on GPUs with GMRES. Springer Science + Business Media, LLC 2011.

49 Lucas, R. F., Wagenbreth, G., Davis, D. M. and Grimes, R. Multifrontal computations on GPUs and their multi-core hosts. (in press).

50 ANANDTECH, NVIDIA's 1.4 Billion Transistor GPU: GT200 Arrives as the GeForce GTX 280 & 260. *url:* http://www.anandtech.com/show/2549/2 . (last visited on 03/30/2011)

51 GPUReview, NVIDIA GeForce GTX 275, *url:* http://www.gpureview.com/GeForce-GTX-275-card-609.html . (last visited on 03/30/2011).

52 GPUReview, NVIDIA GeForce GTX 580, *url:* http://www.gpureview.com/GeForce-GTX-580-card-637.html. (last visited on 03/30/2011).

53 Nvidia. TESLA C2050/C2070 GPU Computing Processor Supercomputing at $1/10^{th}$ the Cost, *url:* http://www.nvidia.com/docs/IO/43395/NV_DS_Tesla_C2050_C2070_jul10_lores.pdf. (last visited on 07/04/2011).

54 Ruetsch G. and Oster B. Getting Started with CUDA, *url: http://www.nvidia.com/content/cudazone/download/Getting_Started_w_CUDA_Training_NVISION08.pdf* . (last visited on 03/30/2011).

55 The University of Arizona, Nvidia Graphics Processing Unit (GPU), *url: http://www2.engr.arizona.edu/~yangsong/gpu.html* , (last visited on 03/31/2011)

56 Karypis, G. and Kumar, V. METIS – A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices – Version 4.0, *University of Minnesota.* (1998).

57 CULA: GPU accelerated linear algebra library. url: *http://www.culatools.com/* . (last visited on 05/15/2011).

58 Nvidia. Compute Visual Profiler User Guide. DU-05 162-001_v04 (2011).

59 Nvidia. Cuda Optimization. url: *http://www.prace-project.eu/hpc-training/training_pdfs/2653.pdf* . (last visited on 07/04/2011).