LARGE SCALE FINITE ELEMENT ANALYSIS USING GPU PARALLEL COMPUTING

by

ASHKAN AKBARIYEH

Presented to the Faculty of the Graduate School of
The University of Texas at Arlington in Partial Fulfillment
of the Requirements
for the Degree of

MASTER OF SCIENCE IN MECHANICAL ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON

May 2012

Copyright © by ASHKAN AKBARIYEH 2012

All Rights Reserved

To my parents who provided me guidance and support through my life.

ACKNOWLEDGEMENTS

I would like to thank my supervising professor Dr. Brian Dennis for constantly motivating and encouraging me, and also for his invaluable advice during the course of my masters studies. I wish to thank my academic advisors Dr. Bo Wang and Dr. Kent Lawrence for their interest in my research and for taking time to serve in my dissertation committee.

I am grateful to all the teachers who taught me during the years I spent in school, in Iran and in the Unites States.

April 20, 2012

ABSTRACT

LARGE SCALE FINITE ELEMENT ANALYSIS
USING GPU PARALLEL COMPUTING

ASHKAN AKBARIYEH, M.S.

The University of Texas at Arlington, 2012

Supervising Professor: Brian H. Dennis

In the past years, graphic processing units have become a new abundant parallel computing resource on personal computers. In this work parallel computation of a typical case in finite element analysis for solids has been practiced. The solution of 3-D linear elastic static problems with 3 degree of freedom is fully implemented utilizing the current GPU technology. Discretization of the problem has been done for two cases of: Tetrahedral and Hexahedral elements. Acceleration of the solution has been realized using SIMT parallel algorithms. Sparse matrix storage formats as well as matrix-vector operations are investigated for optimum hardware utilization. preconditioned conjugate gradient method has been fully implemented on the GPU device as the iterative solver. FEA GPU implementation is compared with the corresponding optimized serial version run on a conventional processor with the same technology for various mesh sizes, sparse matrix storage schemes, and choice of basis function.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iv

ABSTRACT v

LIST OF ILLUSTRATIONS viii

LIST OF TABLES ix

Chapter Page

1. INTRODUCTION 1

1.1 GPU hardware 3

1.1.1 CUDA architecture 3

1.1.2 CUDA language 4

1.1.3 Kernels and threads 5

1.1.4 Hardware execution 8

1.1.5 GPU programing guide lines 10

2. FINITE ELEMENT METHOD 11

2.1 Galerkin formulation 'Weak form' 12

2.2 Displacement field approximation 13

2.2.1 Tetrahedral elements 14

2.2.2 Hexahedral elements 19

2.3 Solving an FEM problem 22

3. SPARSE MATRIX 24

3.1 Sparse matrix storage 25

3.1.1 Coordinate 25

3.1.2 Compressed sparse row 25

3.1.3 Ellpack-Itpack 27
3.1.4 Compressed sparse row block 27
3.2 Matrix vector multiplication parallel schemes 29
3.2.1 Assembled 29
3.2.2 Assembly-free 29
3.3 Iterative solver 30
3.3.1 Jacobi preconditioning 31
4. GPU IMPLEMENTATION 33
4.1 Preconditioned conjugate gradient iterative solver 33
4.2 Sparse matrix storage 36
4.2.1 CSRB-Ellpack 36
4.3 Matrix vector multiplication parallel schemes 38
4.3.1 Assembled 40
4.3.2 Assembly-free 43
5. RESULTS 45
5.1 PCG on GPU 45
5.2 Matrix-vector operation 48
5.2.1 Assembled 48
5.2.2 Assembly-free 49
5.3 Conclusion 50
5.4 Future work 52
REFERENCES 53
BIOGRAPHICAL STATEMENT 56

LIST OF ILLUSTRATIONS

Figure Page

1.1 Scalable architecture [1] 4

1.2 Thread hierarchy [1] 6

1.3 Memory hierarchy [1] 7

2.1 Tetrahedral elements 15

2.2 Hexahedral element 20

3.1 Coordinate storage format 26

3.2 Compressed sparse row storage format 26

3.3 Ell storage format 27

3.4 Compressed sparse row block storage format 28

4.1 GPU preconditioned conjugate gradient flowchart 34

4.2 GPU compressed sparse row block data structure 37

4.3 GPU compressed sparse row block example matrix 39

4.4 Block MVM 40

4.5 Stride and coalesce memory access 41

5.1 Mesh for cantilever beam 46

5.2 Speedup for PCG on GPU using full-matrix CSRB on both GPU and CPU for different mesh sizes 47

5.3 Speedup for PCG on GPU using full-matrix CSRB on GPU and symmetric CSRB on CPU for different mesh sizes 48

5.4 Speedup of MVM for CSRB vs CSR on GPU for different mesh sizes . 49

5.5 Assembly-free method performance chart 51

LIST OF TABLES

Table | Page

5.1 Meshes used for cantilever beam case 46

5.2 Meshes used for assembly-free performance study 50

5.3 Details of computation time in millisecond 51

CHAPTER 1

INTRODUCTION

Graphic processing units have become a new resource for parallel computations. The development of such processors was originally driven by consumer market demand for better computer visualization in the video games. The different computation requirements of computer graphics led to different processor architecture that makes it a suitable device for scientific computing. The graphics card vendors have realized the interest of scientists in the matter and developed the GPU technology to address their need. The GPU technology has accelerated lots of applications with lower cost and it continues to maintain the capacity to outperform the CPU. It is worth mentioning that not all applications are suitable for GPU implementation. NVIDIA has been the leading company in the GPU technology for the past years. They have developed a GPU programming language specifically for scientific computation called *CUDA*. Although CUDA is not portable between other graphic cards and parallel platforms, yet it has been adopted by many scientists and engineers.

Work pertaining to GPUs has extended to a large variety of applications. The following are just a handful of applications of GPU in mechanical engineering field. Examples in the field of fluid mechanics can be found in [2][3][4]. Solids finite element applications have also been a topic of interest in GPU computing. Nonlinear finite element analysis of brain for surgical simulation has been done [5]. High-order Earthquake finite element code has been ported to GPUs in single-precision and mesh coloring technique has been utilized for conflict free global array updates [6]. Implementation of parallel assembling process of stiffness matrix in FEM solution

procedure is done in [7] and minimum residual method has been used as iterative solver. Taylor et al [8] claim to have the first GPU implementation of a nonlinear finite element solver using total Lagrangian explicit finite element formulation. They have reported real-time solution of models with up to 16000 4-node tetrahedral elements. Fast implementation of the conjugate gradient iterative method with field multilevel preconditioning applied to solving real symmetric and sparse systems has been done [9]. Hybrid CPU-GPU solution for FEM problems using domain decomposition methods is presented in [10]. Parallel explicit finite element for sheet metal forming has been developed [11]. Implementation of conjugate gradient method on GPU clusters has been studied [12]. Block compressed sparse row matrix format has been used in a general purpose linear solver [13].

In this thesis a finite element analysis of 3-D solid objects with linear elastic material model is studied. This type of problems end up with large sparse system of equations after finite element discretization. These systems can potentially exceed millions of degrees of freedoms. The solution of such problems using iterative methods are often time consuming, which makes them a good candidate for parallel processing. There is no argue that faster solution with lower cost is always an engineering desire. It gives engineers access to more case studies. Also use of optimization techniques are more justifiable for larger systems.

In this work, a whole finite element problem is implemented on the CPU using C programming language while the solution procedure is ported into GPU using parallel programming techniques. Sparse matrix formats and sparse matrix operations have also been investigated for optimized GPU implementations. An optimized GPU code will utilize the hardware more efficiently and results in more acceleration.

1.1 GPU hardware

Graphic processor unit or GPU in a desktop computer is a separate circuit board specially designed to handle computer graphics. Over time, driven by market demand, the GPU has evolved into a highly parallel, multithreaded, manycore processor with tremendous computational power and very high memory bandwidth [1]. Figures (1-1 from the book) compares same generations of GPU and CPU technology. The reason behind the dramatic difference between GPU and CPU is that the GPU is designed such that more transistors are devoted to data processing rather than data caching [1]. Understanding the parallel architecture of the GPU hardware is essential to write efficient and compatible software code. In this section the current NVIDIA GPU architecture compute capability 2.x is briefly introduced to those who are not exposed to the matter. All information are extracted from the company references [1] [14] [15].

1.1.1 CUDA architecture

The multicore CPUs and manycore GPUs have become the mainstream computing systems. These chips are parallel systems and their parallelism continues to scale with Moore?s law. Developing application software that transparently scale its parallelism and compatibility with the increasing number of processor cores has become a challenge. The CUDA parallel programming model is designed to overcome this challenge while maintaining a low learning curve for programmers familiar with standard programming languages such as C. CUDA at its core has three key abstractions - a hierarchy of thread groups, shared memories, and barrier synchronization that are simply exposed to the programmer as a minimal set of language extensions [1]. These abstractions guide the programmer to partition the problem into coarse sub-problems that can be solved independently in parallel by blocks of threads. Each

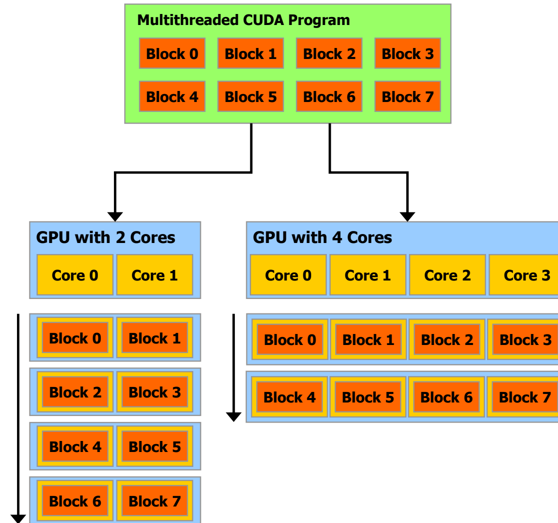

Figure 1.1. Scalable architecture [1].

sub-problem partitions into finer pieces that can be solved cooperatively in parallel by all threads within the block. This decomposition allows threads to cooperate when solving each sub-problem, and at the same time enables automatic scalability. Indeed, each block of threads can be scheduled on any of the available processor cores, in any order and only the runtime system needs to know the physical processor count. Figure 1.1 depicts the automatic scalability of this decomposition.

1.1.2 CUDA language

CUDA is the hardware and software architecture that enables NVIDIA GPUs to execute programs written with C, C++, Fortran, and other languages[Fermi paper]. CUDA allows software developers to use C as a high-level programming language. CUDA C enables users familiar with the C programming language to easily write programs executable by the device. The CUDA C introduces programming language extensions that allow users to define a *kernel* as a C function and use some new

syntax to specify the parallel execution properties each time the kernel function is called.

1.1.3 Kernels and threads

As mentioned earlier, kernels are the user defined parallel functions defined in CUDA C. A kernel executes in parallel across a set of parallel *threads*. The GPU instantiates a kernel program on a grid of parallel *threadblocks*. Each thread within a thread block executes an instance of the kernel, and has a thread ID within its thread block that is accessible within the kernel through the built-in *threadIdx* variable. A thread block is a set of concurrently executing threads that can cooperate among themselves through barrier synchronization and shared memory. A thread block has a *blockID* within its grid. A *grid* is an array of thread blocks that execute the same kernel, read inputs from global memory, write results to global memory, and synchronize between dependent kernel calls.

1.1.3.1 Thread hierarchy

Threads can be identified using a one-dimensional, two-dimensional, or three-dimensional thread index, forming a one-dimensional, two-dimensional, or three-dimensional thread block. There is a limit to the number of threads per block, since all threads of a block are expected to reside on the same processor core and must share the limited memory resources of that core. However, a kernel can be executed by multiple equally-shaped thread blocks, so that the total number of threads is equal to the number of threads per block times the number of blocks. Blocks are organized into a one-dimensional, two-dimensional, or three-dimensional grid of thread blocks as illustrated by Figure 1.2. Threads within a block can cooperate by sharing data

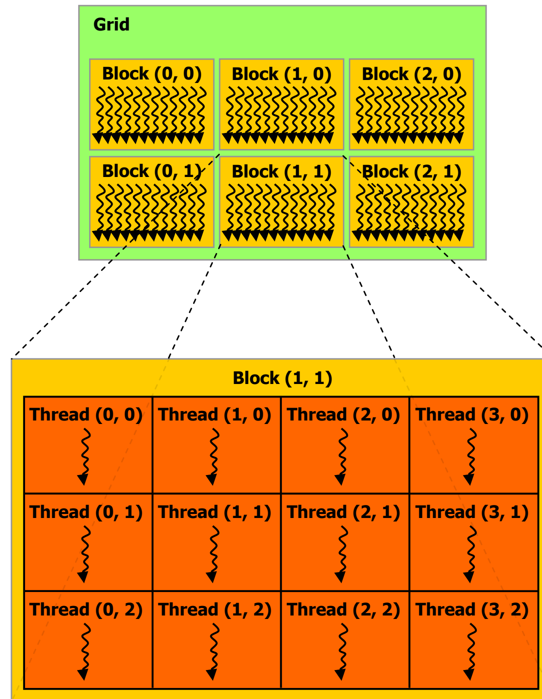

Figure 1.2. Thread hierarchy [1].

through some shared memory and by synchronizing their execution to coordinate memory accesses.

1.1.3.2 Memory hierarchy

CUDA threads may access data from multiple memory spaces during their execution as illustrated by Figure 1.3. In the CUDA parallel programming model, each thread has a per-thread private memory space used for register spills, function calls, etc. Each thread has private local memory. Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block. All threads have access to the same global memory. The CUDA programming model assumes that the CUDA threads execute on a physically separate device that operates as a coprocessor to the host running the C program. This is the case, for

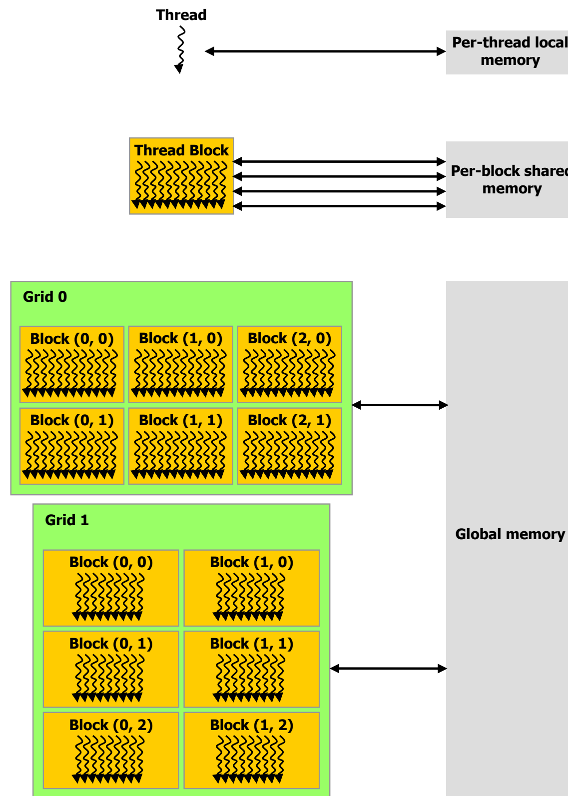

Figure 1.3. Memory hierarchy [1].

example, when the kernels execute on a GPU and the rest of the C program executes on a CPU. The CUDA programming model also assumes that both the host and the device maintain their own separate memory spaces in DRAM, referred to as host memory and device memory, respectively.

1.1.3.3 Atomics

Parallel programmers must be aware of race condition when they access the memory. A race condition refers to a state that two or more threads simultaneously issue a read or write instruction to the same memory location. Race conditions are programing mistakes and must be avoided by changing the parallel algorithm, using synchronizing functions or using *atomic* function. Atomic functions solve the race

condition with the high penalty of serializing memory access and therefore massive reduction in memory bandwidth.

1.1.4 Hardware execution

The CUDA architecture is built around a scalable array of multithreaded Streaming Multiprocessors (SMs). When a CUDA program on the host CPU invokes a kernel grid, the blocks of the grid are enumerated and distributed to multiprocessors with available execution capacity. The threads of a thread block execute concurrently on one multiprocessor, and multiple thread blocks can execute concurrently on one multiprocessor. As thread blocks terminate, new blocks are launched on the vacated multiprocessors [1]. CUDA's hierarchy of threads maps to a hierarchy of processors on the GPU; a GPU executes one or more kernel grids; a streaming multiprocessor (SM) executes one or more thread blocks; and CUDA cores and other execution units in the SM execute threads. [14]

1.1.4.1 Streaming multiprocessors

A multiprocessor is designed to execute hundreds of threads concurrently. To manage such a large amount of threads, it employs a unique architecture called SIMT (Single-Instruction, Multiple-Thread). The instructions are pipelined to leverage instruction-level parallelism within a single thread, as well as thread-level parallelism extensively with simultaneous hardware multithreading. Threads on a CPU are generally heavyweight entities. The operating system must swap threads on and off of CPU execution channels to provide multithreading capability. By comparison, threads on GPUs are extremely lightweight. In a typical system, thousands of threads are queued up for work (in warps of 32 threads each). If the GPU must wait on one warp of threads, it simply begins executing work on another. Because separate reg-

isters are allocated to all active threads, resources stay allocated to each thread until it completes its execution [15].

1.1.4.2 Warp

The multiprocessor creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps. Individual threads composing a warp start together at the same program address, but they have their own instruction set and register state and are therefore free to branch and execute independently. When a multiprocessor is given one or more thread blocks to execute, it partitions them into warps that get scheduled by a warp scheduler for execution. Each warp contains threads of consecutive, increasing thread IDs with the first warp containing thread 0. A warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agree on their execution path. If threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path. Different warps execute independently regardless of what they are executing. At every instruction issue time, a warp scheduler selects a warp that has threads ready to execute its next instruction (the active threads of the warp) and issues the instruction to those threads. In particular, each multiprocessor has a set of 32-bit registers that are partitioned among the warps, and a parallel data cache or shared memory that is partitioned among the thread blocks. The number of blocks and warps that can reside and be processed together on the multiprocessor for a given kernel depends on the amount of registers and shared memory used by the kernel and the amount of registers and shared memory available on the multiprocessor. There are also a maximum number of resident blocks and a maximum number of resident warps per multiprocessor.

1.1.5 GPU programing guide lines

Because of the hardware implementation, there are a series of performance guidelines that must be observed when programming a GPU using CUDA. The most important ones refer to: memory transfers between host and device, memory latency when accessing global and local memory, global memory access pattern , shared memory access patterns (to avoid memory bank conflicts), and execution configuration (number of threads per block and number of thread blocks specified for a kernel launch). When choosing the execution configuration, based on the advice given in the programming guide, the number of threads per block should be a multiple of 64. The fastest kernels minimize access to device memory, avoid non-coalesced accesses to global memory, avoid bank conflicts when reading from or writing to shared memory, and try to minimize register and/or shared memory usage to maximize occupancy. At the same time, one strives to work with many blocks running per multiprocessor to overlap the latencies of memory transfers. In the GPU memory the data can be re-organized in order to obtain maximum memory access performances. The data organization and alignment must ensure coalesced memory access as much as possible, especially for memory writes. The details on these guidelines are included in the CUDA Programming Guide and should be observed during the software implementation in order to obtain maximum performance.

CHAPTER 2

FINITE ELEMENT METHOD

Most problems in engineering mechanics can be stated either as continuous or discrete problems. Discrete problems involve finite number of degrees of freedom. All discrete and continuous problems can be classified as equilibrium, eigenvalue, and propagation problems. The finite element method is applicable for the solution of all three categories of problems. The finite element method is a numerical procedure that replaces a continuous problem by an equivalent discrete one. A finite element solution process for a typical structural problem may be established by:

1. Using the virtual work (or weak form), equations for equilibrium equations
2. Introducing an approximation for the displacement field u in terms of shape functions.
3. Computing strains from displacement field.
4. Computing stresses using material constitutive relation.
5. Performing the integrations over each element.
6. Assembling the element local matrices to form the global stiffness matrix and load vector.
7. Imposing the known traction and displacement boundary conditions.
8. Solving the resulting stiffness and load matrices.
9. Reporting desired parts of the solution.

Finite element method formulations presented in this section are extracted from reference books [16] [17].

2.1 Galerkin formulation 'Weak form'

The Galerkin formulation is a method of weighted residuals which the original shape (or basis) functions are used as weighting. The finite element approximation is derived from applying Galerkin formulation of the weighted residual process to the equilibrium equation. Galerkin method, often leads to symmetric matrices. The well-known finite difference method of approximation is a particular case of collocation with locally defined basis functions and is thus a case of a Galerkin scheme. The integral statement necessary for formulation in terms of the finite element approximation is supplied via the principle of virtual work. Virtual work statement is a 'weak form' of equilibrium equations. The following Equation is the three-dimensional equivalent virtual work statement.

$$\int_{\Omega} \delta\epsilon^{\mathrm{T}}\boldsymbol{\sigma}\,\mathrm{d}\Omega - \int_{\Omega} \delta\mathbf{u}^{\mathrm{T}}\mathbf{b}\,\mathrm{d}\Omega - \int_{\Gamma} \delta\mathbf{u}^{\mathrm{T}}\mathbf{t}\,\mathrm{d}\Gamma = 0 \tag{2.1}$$

The first term of Eq.2.1 is the variation of the strain energy of the system:

$$\delta U = \int_{\Omega} \delta\epsilon^{\mathrm{T}}\boldsymbol{\sigma}\,\mathrm{d}\Omega \tag{2.2}$$

For linear elastic materials without residual stress and strain we have:

$$\boldsymbol{\sigma} = \mathbf{D}\epsilon \tag{2.3}$$

Using Eq.2.3 we can write U as:

$$U = \frac{1}{2}\int_{\Omega} \epsilon^{\mathrm{T}}\mathbf{D}\,\epsilon\,\mathrm{d}\Omega \tag{2.4}$$

The next two terms of Eq.2.1 represent variation of the potential energy of the external loads:

$$\delta W = -\delta\left(\int_{\Omega} \mathbf{u}^{\mathrm{T}}\mathbf{b}\,\mathrm{d}\Omega + \int_{\Gamma} \mathbf{u}^{\mathrm{T}}\mathbf{t}\,\mathrm{d}\Gamma\right) \tag{2.5}$$

$$W = -\int_{\Omega} \mathbf{u}^{\mathrm{T}}\mathbf{b}\,\mathrm{d}\Omega - \int_{\Gamma} \mathbf{u}^{\mathrm{T}}\mathbf{t}\,\mathrm{d}\Gamma \tag{2.6}$$

We can write variation of the total potential energy as:

$$\delta\Pi = \delta(U + W) = 0 \tag{2.7}$$

The above statement means that for equilibrium to be ensured the total potential energy must be stationary for variations of the admissible displacements.The variation with respect to displacements with the finite number of parameters $u \approx \mathbf{N}\,\tilde{\mathbf{u}}$ is now written as:

$$\Pi = \frac{1}{2}\tilde{\mathbf{u}}^{\mathrm{T}}\mathbf{K}\tilde{\mathbf{u}} + \tilde{\mathbf{u}}^{\mathrm{T}}\mathbf{f} \tag{2.8}$$

$$\frac{\delta\Pi}{\delta\tilde{\mathbf{u}}} = \begin{Bmatrix} \frac{\delta\Pi}{\delta\tilde{\mathbf{u}}_1} \\ \frac{\delta\Pi}{\delta\tilde{\mathbf{u}}_2} \\ \vdots \\ \frac{\delta\Pi}{\delta\tilde{\mathbf{u}}_n} \end{Bmatrix} \mathbf{K}\tilde{\mathbf{u}} + \mathbf{f} = \mathbf{0} \tag{2.9}$$

in which K and f are given by:

$$\mathbf{K} = \int_{\Omega} \mathbf{B}^{\mathrm{T}}\mathbf{D}\,\mathbf{B}\,\mathrm{d}\Omega \tag{2.10}$$

$$f = -\int_{\Omega} \mathbf{N}^{\mathrm{T}}\mathbf{b}\,\mathrm{d}\Omega - \int_{\Gamma} \mathbf{N}^{\mathrm{T}}\mathbf{t}\,\mathrm{d}\Gamma \tag{2.11}$$

In the Eq.2.10, B is the appropriate matrix relating displacement and strain fields:

$$\epsilon = \boldsymbol{S}\,\mathbf{u} = \boldsymbol{S}\,\mathbf{N}\tilde{\mathbf{u}} = \mathbf{B}\tilde{\mathbf{u}} \tag{2.12}$$

where $\boldsymbol{S}$ is a suitable linear differential operator.

2.2 Displacement field approximation

After formulating the problem using variational principles, it is time to approximate displacement field. There are multiple choices for element types in finite element method. Each element type has certain geometry and interpolating shape

functions. Finite elements can be classified into three categories as simplex, complex, and multiplex elements depending on the geometry of the element and the order of the polynomial used in the interpolation function. The simplex elements are those for which the approximating polynomial consists of constant and linear terms. Four-node Tetrahedral is a 3-D simplex element. The complex elements may have the same shapes as the simplex elements but will have additional nodes. For example Ten-node Tetrahedral has interpolating polynomial including terms up to quadratic terms. The multiplex elements are those whose boundaries are parallel to the coordinate axes. Eight-node Hexahedral is an example of multiplex elements.

2.2.1 Tetrahedral elements

In this section the formulation of the tetrahedral elements with 3 degree of freedom per node are derived. Tetrahedral element family starts with the linear four-node tetrahedral following by the quadratic ten-node tetrahedral. All tetrahedral elements have a complete set of the polynomial order they represent. Two types of Tetrahedral elements are shown in Figure 2.1.

2.2.1.1 TET4

In the TET4[1] element, displacement field is approximated by first order linear functions:

$$
\begin{aligned}
u &= N_1 x_1 + N_2 x_2 + N_3 x_3 + N_4 x_4 \\
v &= N_1 y_1 + N_2 y_2 + N_3 y_3 + N_4 y_4 \\
z &= N_1 z_1 + N_2 z_2 + N_3 z_3 + N_4 z_4 \\
1 &= N_1 + N_2 + N_3 + N_4
\end{aligned}
\tag{2.13}
$$

[1] 4-node Tetrahedral

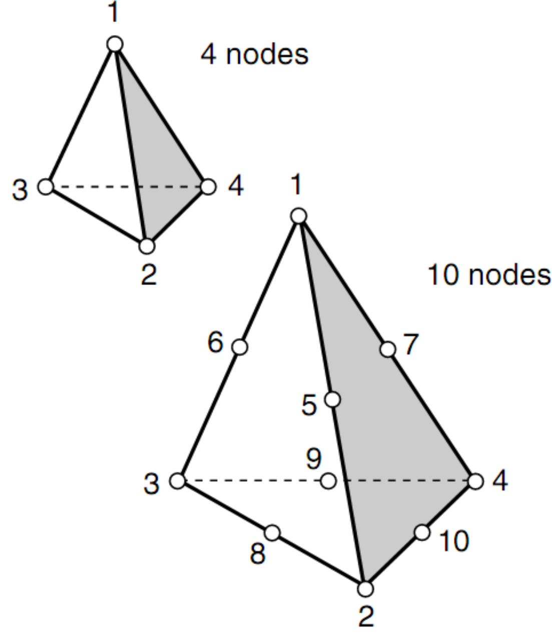

Figure 2.1. Tetrahedral elements.

Where N_k are natural functions defined by:

$$N_k = \frac{a_k + b_k x + c_k y + d_k z}{6V} \qquad k = 1, 2, 3, 4 \tag{2.14}$$

$$6V = \det \begin{vmatrix} 1 & x_1 & y_1 & z_1 \\ 1 & x_2 & y_2 & z_2 \\ 1 & x_3 & y_3 & z_3 \\ 1 & x_4 & y_4 & z_4 \end{vmatrix} \tag{2.15}$$

V is the volume of the element. Displacement fields from Eq.2.13 are rewritten as functions of (x, y, z):

$$\begin{aligned} u(x,y,z) &= N_1(x,y,z)x_1 + N_2(x,y,z)x_2 + N_3(x,y,z)x_3 + N_4(x,y,z)x_4 \\ v(x,y,z) &= N_1(x,y,z)y_1 + N_2(x,y,z)y_2 + N_3(x,y,z)y_3 + N_4(x,y,z)y_4 \\ w(x,y,z) &= N_1(x,y,z)z_1 + N_2(x,y,z)z_2 + N_3(x,y,z)z_3 + N_4(x,y,z)z_4 \end{aligned} \tag{2.16}$$

Using nodal coordinates we find the unknown coefficients in the natural function as follow:

$$a_1 = \det\begin{vmatrix} x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \\ x_4 & y_4 & z_4 \end{vmatrix} \qquad a_2 = \det\begin{vmatrix} x_3 & y_3 & z_3 \\ x_4 & y_4 & z_4 \\ x_1 & y_1 & z_1 \end{vmatrix}$$

$$a_3 = \det\begin{vmatrix} x_4 & y_4 & z_4 \\ x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \end{vmatrix} \qquad a_4 = \det\begin{vmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{vmatrix} \tag{2.17}$$

$$b_1 = -\det\begin{vmatrix} 1 & y_2 & z_2 \\ 1 & y_3 & z_3 \\ 1 & y_4 & z_4 \end{vmatrix} \qquad b_2 = -\det\begin{vmatrix} 1 & y_3 & z_3 \\ 1 & y_4 & z_4 \\ 1 & y_1 & z_1 \end{vmatrix}$$

$$b_3 = -\det\begin{vmatrix} 1 & y_4 & z_4 \\ 1 & y_1 & z_1 \\ 1 & y_2 & z_2 \end{vmatrix} \qquad b_4 = -\det\begin{vmatrix} 1 & y_1 & z_1 \\ 1 & y_2 & z_2 \\ 1 & y_3 & z_3 \end{vmatrix} \tag{2.18}$$

$$c_1 = -\det\begin{vmatrix} x_2 & 1 & z_2 \\ x_3 & 1 & z_3 \\ x_4 & 1 & z_4 \end{vmatrix} \qquad c_2 = -\det\begin{vmatrix} x_3 & 1 & z_3 \\ x_4 & 1 & z_4 \\ x_1 & 1 & z_1 \end{vmatrix}$$

$$c_3 = -\det\begin{vmatrix} x_4 & 1 & z_4 \\ x_1 & 1 & z_1 \\ x_2 & 1 & z_2 \end{vmatrix} \qquad c_4 = -\det\begin{vmatrix} x_1 & 1 & z_1 \\ x_2 & 1 & z_2 \\ x_3 & 1 & z_3 \end{vmatrix} \tag{2.19}$$

$$d_1 = - \quad \det \begin{vmatrix} x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \\ x_4 & y_4 & 1 \end{vmatrix} \qquad d_2 = - \quad \det \begin{vmatrix} x_3 & y_3 & 1 \\ x_4 & y_4 & 1 \\ x_1 & y_1 & 1 \end{vmatrix}$$

$$d_3 = - \quad \det \begin{vmatrix} x_4 & y_4 & 1 \\ x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \end{vmatrix} \qquad d_4 = - \quad \det \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} \tag{2.20}$$

$$\underset{3\times1}{\{\vec{U}\}} = \begin{Bmatrix} u(x,y,z) \\ v(x,y,z) \\ w(x,y,z) \end{Bmatrix} = \underset{3\times12}{[N]} \ \underset{12\times1}{\{\vec{Q}^{(e)}\}} \tag{2.21}$$

$$[N] = \begin{bmatrix} N_1 & 0 & 0 & N_2 & 0 & 0 & N_3 & 0 & 0 & N_4 & 0 & 0 \\ 0 & N_1 & 0 & 0 & N_2 & 0 & 0 & N_3 & 0 & 0 & N_4 & 0 \\ 0 & 0 & N_1 & 0 & 0 & N_2 & 0 & 0 & N_3 & 0 & 0 & N_4 \end{bmatrix} \tag{2.22}$$

Equation 2.22 represent a symbolic matrix of shape functions and $\{\vec{Q}^{(e)}\}$ is the nodal displacement vector.

$$\{\vec{Q}^{(e)}\} = \begin{Bmatrix} u_1 \\ v_1 \\ w_1 \\ \vdots \\ w_4 \end{Bmatrix} \tag{2.23}$$

Matrix $[B]$ is the appropriate derivative matrix relating nodal displacement to the strain field vector.

$$\underset{6\times 1}{\{\vec{\epsilon}\}} = \begin{Bmatrix} \epsilon_{xx} \\ \epsilon_{yy} \\ \epsilon_{zz} \\ \epsilon_{xy} \\ \epsilon_{yz} \\ \epsilon_{zx} \end{Bmatrix} = \begin{Bmatrix} \partial u/\partial x \\ \partial v/\partial y \\ \partial w/\partial z \\ \frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \\ \frac{\partial v}{\partial z} + \frac{\partial w}{\partial y} \\ \frac{\partial w}{\partial x} + \frac{\partial u}{\partial z} \end{Bmatrix} = \underset{6\times 12}{[B]}\ \underset{12\times 1}{\{\vec{Q}^{(e)}\}} \tag{2.24}$$

$$[B] = \frac{1}{6V} \begin{bmatrix} b_1 & 0 & 0 & b_2 & 0 & 0 & b_3 & 0 & 0 & b_4 & 0 & 0 \\ 0 & c_1 & 0 & 0 & c_2 & 0 & 0 & c_3 & 0 & 0 & c_4 & 0 \\ 0 & 0 & d_1 & 0 & 0 & d_2 & 0 & 0 & d_3 & 0 & 0 & d_4 \\ c_1 & b_1 & 0 & c_2 & b_2 & 0 & c_3 & b_3 & 0 & c_4 & b_4 & 0 \\ 0 & d_1 & c_1 & 0 & d_2 & c_2 & 0 & d_3 & c_3 & 0 & d_4 & c_4 \\ d_1 & 0 & b_1 & d_2 & 0 & b_2 & d_3 & 0 & b_3 & d_4 & 0 & b_4 \end{bmatrix} \tag{2.25}$$

$$\vec{\sigma} = [D]\vec{\epsilon} \tag{2.26}$$

$$\vec{\sigma}^{\mathrm{T}} = \begin{Bmatrix} \sigma_{xx} & \sigma_{yy} & \sigma_{zz} & \sigma_{xy} & \sigma_{yz} & \sigma_{zx} \end{Bmatrix} \tag{2.27}$$

For isotropic elastic materials matrix $[D]$ is given by:

$$[D] = \frac{E}{(1+\nu)(1-2\nu)} \begin{bmatrix} 1-\nu & \nu & \nu & 0 & 0 & 0 \\ \nu & 1-\nu & \nu & 0 & 0 & 0 \\ \nu & \nu & 1-\nu & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1-2\nu}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1-2\nu}{2} & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1-2\nu}{2} \end{bmatrix} \tag{2.28}$$

The stiffness matrix of the element in global coordinate system can be obtained as:

$$[K^{(e)}] = \iiint_{V^{(e)}} [B]^{\mathrm{T}}[D][B]\,\mathrm{dV} \tag{2.29}$$

2.2.2 Hexahedral elements

We consider the simplest hexahedron element having eight corner nodes with three degrees of freedom per node. This element is also known as Brick with eight nodes. As shown in Figure 2.2, the natural coordinates are r s and t with the origin of the system taken at the centroid of the element. In the natural coordinates, the element is a cube, although in the global Cartesian coordinate system it may be an arbitrarily warped and distorted six-sided solid. The relation between global and natural coordinate is:

$$\begin{Bmatrix} x \\ y \\ z \end{Bmatrix} = [N] \begin{Bmatrix} x_1 \\ y_1 \\ z1 \\ \vdots \\ z_8 \end{Bmatrix} \tag{2.30}$$

$$[N] = \begin{bmatrix} N_1 & 0 & 0 & N_2 & \cdot & 0 \\ 0 & N_1 & 0 & 0 & \cdot & 0 \\ 0 & 0 & N_1 & 0 & \cdots & N_8 \end{bmatrix} \tag{2.31}$$

$$N_i(r,s,t) = \frac{1}{8}(1+rr_i)(1+ss_i)(1+tt_i) \qquad\qquad i = 1,2,\cdots,8 \tag{2.32}$$

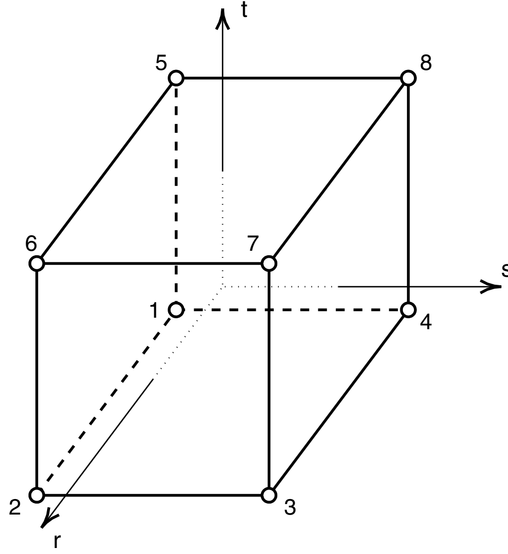

Figure 2.2. Hexahedral element.

Displacement field is approximated by Eq.2.33 where (u_i, v_i, w_i) denote the displacements of node i for $i = 1, 2, \ldots, 8$.

$$\underset{3\times1}{\{\vec{U}\}} = \begin{Bmatrix} u \\ v \\ w \end{Bmatrix} = [N] \begin{Bmatrix} u_1 \\ v_1 \\ w_1 \\ \vdots \\ w_8 \end{Bmatrix} = [N]\{\vec{Q}^{(e)}\} \tag{2.33}$$

$$\vec{\epsilon} = \underset{6\times24}{[B]}\underset{24\times1}{\{\vec{Q}^{(e)}\}} \tag{2.34}$$

$$\underset{6\times24}{[B]} = \begin{bmatrix} [B_1] & [B_2] & \cdots & [B_8] \end{bmatrix} \tag{2.35}$$

$$
\underset{6\times 3}{[B_i]} = \begin{bmatrix} \frac{\partial N_i}{\partial x} & 0 & 0 \\ 0 & \frac{\partial N_i}{\partial y} & 0 \\ 0 & 0 & \frac{\partial N_i}{\partial z} \\ \frac{\partial N_i}{\partial y} & \frac{\partial N_i}{\partial x} & 0 \\ 0 & \frac{\partial N_i}{\partial z} & \frac{\partial N_i}{\partial y} \\ \frac{\partial N_i}{\partial z} & 0 & \frac{\partial N_i}{\partial x} \end{bmatrix} \qquad i = 1, 2, \ldots, 8 \tag{2.36}
$$

By applying the chain rule of differentiation we have:

$$
\begin{Bmatrix} \frac{\partial N_i}{\partial r} \\ \frac{\partial N_i}{\partial s} \\ \frac{\partial N_i}{\partial t} \end{Bmatrix} = \begin{bmatrix} \frac{\partial x}{\partial r} & \frac{\partial y}{\partial r} & \frac{\partial z}{\partial r} \\ \frac{\partial x}{\partial s} & \frac{\partial y}{\partial s} & \frac{\partial z}{\partial s} \\ \frac{\partial x}{\partial t} & \frac{\partial y}{\partial t} & \frac{\partial z}{\partial t} \end{bmatrix} \begin{Bmatrix} \frac{\partial N_i}{\partial x} \\ \frac{\partial N_i}{\partial y} \\ \frac{\partial N_i}{\partial z} \end{Bmatrix} = [J] \begin{Bmatrix} \frac{\partial N_i}{\partial x} \\ \frac{\partial N_i}{\partial y} \\ \frac{\partial N_i}{\partial z} \end{Bmatrix} \tag{2.37}
$$

$$
\underset{3\times 3}{[J]} = \begin{bmatrix} \frac{\partial x}{\partial r} & \frac{\partial y}{\partial r} & \frac{\partial z}{\partial r} \\ \frac{\partial x}{\partial s} & \frac{\partial y}{\partial s} & \frac{\partial z}{\partial s} \\ \frac{\partial x}{\partial t} & \frac{\partial y}{\partial t} & \frac{\partial z}{\partial t} \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{8}(\frac{\partial N_i}{\partial r}x_i) & \sum_{i=1}^{8}(\frac{\partial N_i}{\partial r}y_i) & \sum_{i=1}^{8}(\frac{\partial N_i}{\partial r}z_i) \\ \sum_{i=1}^{8}(\frac{\partial N_i}{\partial s}x_i) & \sum_{i=1}^{8}(\frac{\partial N_i}{\partial s}y_i) & \sum_{i=1}^{8}(\frac{\partial N_i}{\partial s}z_i) \\ \sum_{i=1}^{8}(\frac{\partial N_i}{\partial t}x_i) & \sum_{i=1}^{8}(\frac{\partial N_i}{\partial t}y_i) & \sum_{i=1}^{8}(\frac{\partial N_i}{\partial t}z_i) \end{bmatrix} \tag{2.38}
$$

The derivatives of interpolation functions are:

$$
\left.\begin{aligned} \frac{\partial N_i}{\partial r} &= \frac{1}{8}r_i(1+ss_i)(1+tt_i) \\ \frac{\partial N_i}{\partial s} &= \frac{1}{8}s_i(1+rr_i)(1+tt_i) \\ \frac{\partial N_i}{\partial t} &= \frac{1}{8}t_i(1+rr_i)(1+ss_i) \end{aligned}\right\} \qquad i = 1, 2, \ldots, 8 \tag{2.39}
$$

$$
\begin{Bmatrix} \frac{\partial N_i}{\partial x} \\ \frac{\partial N_i}{\partial y} \\ \frac{\partial N_i}{\partial z} \end{Bmatrix} = [J]^{-1} \begin{Bmatrix} \frac{\partial N_i}{\partial r} \\ \frac{\partial N_i}{\partial s} \\ \frac{\partial N_i}{\partial t} \end{Bmatrix} \qquad i = 1, 2, \ldots, 8 \tag{2.40}
$$

Now that we derived the derivatives of the shape functions, we can populate matrix [B] and the element stiffness matrix given Eq.2.41. Since the matrix [B] is expressed

in natural coordinates, it is necessary to carry out the integration in Eq.2.41, in natural coordinates. The Gaussian quadrature has been proven to be the most efficient method of numerical integration for this class of problems. Using the two-point Gaussian quadrature, yields to sufficiently accurate results.

$$[K^{(e)}] = \iiint\limits_{V^{(e)}} [B]^{\mathrm{T}}[D][B]\,\mathrm{d}V \tag{2.41}$$

$$\mathrm{d}V = \mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z = \det[J]\mathrm{d}r\,\mathrm{d}s\,\mathrm{d}t \tag{2.42}$$

$$[K^{(e)}] = \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} [B]^{\mathrm{T}}[D][B]\det[J]\,\mathrm{d}r\,\mathrm{d}s\,\mathrm{d}t \tag{2.43}$$

2.3 Solving an FEM problem

When the finite element method is used for the solution of equilibrium problems, we get a set of simultaneous linear equations. The governing finite element equations for various types of field equilibrium problems can be expressed in matrix form as follows:

$$[A]\{x\} = \{b\} \tag{2.44}$$

$$\underset{n\times n}{[A]} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix}, \qquad \underset{n\times 1}{\{x\}} = \begin{Bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{Bmatrix}, \qquad \underset{n\times 1}{\{b\}} = \begin{Bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{Bmatrix} \tag{2.45}$$

In finite element analysis, the size of the matrix [A] will be very large. The solution of some of the practical problems involves matrices of order 10,000 or more. The methods available for solving systems of linear equations can be divided into two types: direct and iterative. Direct methods are those that, in the absence of round-off and other errors, will yield the exact solution in a finite number of elementary

arithmetic operations. In practice, sometimes the direct methods do not give good solutions because a computer works with a finite precision. The errors arising from round-off and truncation may lead to extremely poor or even useless results. The fundamental method used for direct solutions is Gaussian elimination. The Choleski method is also a direct method for solving a linear system that makes use of the fact that any square matrix [A] can be expressed as the product of an upper and a lower triangular matrix. Iterative methods are those that start with an initial approximation and that by applying a suitably chosen algorithm lead to successively better approximations. When the process converges, we can expect to get a good approximate solution. The main advantages of iterative methods are the simplicity and uniformity of the operations to be performed. Matrices associated with linear systems are also classified as dense or sparse. Dense matrices have very few zero elements, whereas sparse matrices have very few nonzero elements. Fortunately, in most finite element applications, the matrices involved are sparse and symmetric.

CHAPTER 3

SPARSE MATRIX

A matrix populated primarily with zeros is called a sparse matrix. Sparse matrices are vastly used in commercial computing software packages. The idea of sparse matrix storage is to store non-zeros only. This way we use computer memory efficiently and we avoid unnecessary processing of zeros. Sparse matrix operations have a key role in the solution procedure of finite element analysis.

In general application of variational principles, we build a mathematical model by discretizing the specified domain. Then we use formulation derived from variational calculus to form local matrices for each cell individually. Afterward we assemble all the local matrices into a global matrix, and turn the problem into a system of equations. The matrix arising from such discretization is very sparse due to loose coupling of cells. In the case of a problem in structural mechanics, we call this matrix the stiffness matrix of the system. The sparsity of stiffness matrix depends on the generated grid and the type of the elements used. More specifically the number of non-zeros per row in the global matrix depends on the number of cells which share the same node, and the size of the local stiffness matrix.

When the problem size scales by N, the stiffness matrix size scales by order of N^2. Fortunately the sparse nature of the stiffness matrix, allow the required memory space to scale with N.

3.1 Sparse matrix storage

Sparse matrices can be stored in multiple ways. Each storage format has its own computational properties. Choosing the right method plays a key role in performance of the future matrix operations. Familiarity with different sparse matrix storage is essential to proceed with the rest of this thesis. Hereby I demonstrate common well known sparse matrix storage schemes plus an altered version, specifically chosen for GPU computing application.

3.1.1 Coordinate

Coordinate format is the simplest storage scheme for sparse matrices. The data structure consists of three arrays:(1) a real array containing all the real values of nonzero elements in A in any order;(2) an integer array containing their row indices; and (3) a second integer array containing their column indices. All three arrays are of the same length equal to the number of nonzero elements [18]. An example matrix and its COO storage is provided in Figure 3.1.

3.1.2 Compressed sparse row

Compressed sparse row format is very similar to the COO format. The data structure consists of the same arrays. The first array contains real values of nonzero elements in order, row by row. The second array contains column indices of each nonzero value. The third array contains pointers to the beginning and the end of each row. Figure 3.2 demonstrates CSR storage scheme for an example sparse matrix. CSR format use less memory space compared to COO by avoiding to store redundant information. CSR is also a preferred choice due to better performance in typical matrix computations [18].

$$A = \begin{bmatrix} 1 & 2 & 0 & 3 \\ 0 & 4 & 5 & 0 \\ 6 & 0 & 7 & 8 \\ 0 & 0 & 0 & 9 \end{bmatrix}$$

$$data = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{bmatrix}$$
$$rows = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 2 & 2 & 2 & 3 \end{bmatrix}$$
$$cols = \begin{bmatrix} 0 & 1 & 3 & 1 & 2 & 0 & 2 & 3 & 3 \end{bmatrix}$$

Figure 3.1. Coordinate storage format.

$$A = \begin{bmatrix} 1 & 2 & 0 & 3 \\ 0 & 4 & 5 & 0 \\ 6 & 0 & 7 & 8 \\ 0 & 0 & 0 & 9 \end{bmatrix}$$

$$vals = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{bmatrix}$$
$$cols = \begin{bmatrix} 0 & 1 & 3 & 1 & 2 & 0 & 2 & 3 & 3 \end{bmatrix}$$
$$rows = \begin{bmatrix} 0 & 3 & 5 & 8 & 9 \end{bmatrix}$$

Figure 3.2. Compressed sparse row storage format.

$$A = \begin{bmatrix} 1 & 2 & 0 & 3 \\ 0 & 4 & 5 & 0 \\ 6 & 0 & 7 & 8 \\ 0 & 0 & 0 & 9 \end{bmatrix}$$

$$data = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & * \\ 6 & 7 & 8 \\ 9 & * & * \end{bmatrix} \qquad cols = \begin{bmatrix} 0 & 1 & 3 \\ 1 & 2 & * \\ 0 & 2 & 3 \\ 3 & * & * \end{bmatrix}$$

Figure 3.3. Ell storage format.

3.1.3 Ellpack-Itpack

This storage format is a general scheme popular on vector machines. The assumption in this scheme is that the maximum of nonzero elements per row Nd is small [18]. The data structure consists of two rectangular arrays of dimensions $n*Nd$. The first array contains all the nonzero elements of matrix A. For those rows which number of nonzero elements are less than Nd, the row is completed by zeros. The second array contains column position of each entry in the first array. Figure 3.3 represents an example matrix stored in *Ell* format.

3.1.4 Compressed sparse row block

Compressed sparse row block format is an altered version of the popular CSR scheme. The main goal of developing CSRB was to achieve higher performance in multi degree of freedom problems compared to CSR method. In a single DOF problem, when we generate local stiffness matrix, coupling between nodes is represented

$$A = \left[\begin{array}{cc|cc|cc|cc}
7 & 3 & 9 & 2 & 0 & 0 & 2 & 1 \\
1 & 4 & 8 & 6 & 0 & 0 & 4 & 5 \\
\hline
0 & 0 & 3 & 7 & 5 & 2 & 0 & 0 \\
0 & 0 & 1 & 9 & 8 & 6 & 0 & 0 \\
\hline
8 & 5 & 0 & 0 & 1 & 3 & 6 & 2 \\
3 & 7 & 0 & 0 & 2 & 1 & 9 & 3 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 5 & 6 \\
0 & 0 & 0 & 0 & 0 & 0 & 2 & 8
\end{array}\right]$$

$$vlas = [\,7\ 3\ 1\ 4\quad 9\ 2\ 8\ 6\quad 2\ 1\ 4\ 5\quad 3\ 7\ 1\ 9\quad \ldots$$

$$5\ 2\ 8\ 6\quad 8\ 5\ 3\ 7\quad 1\ 3\ 2\ 1\quad 6\ 2\ 9\ 3\quad 5\ 6\ 2\ 8\,]$$

$$cols = \begin{bmatrix} 0 & 1 & 3 & 1 & 2 & 0 & 2 & 3 & 3 \end{bmatrix}$$

$$rows = \begin{bmatrix} 0 & 3 & 5 & 8 & 9 \end{bmatrix}$$

Figure 3.4. Compressed sparse row block storage format.

by one number. However in multi-DOF problems, this coupling happens to be a square matrix. The idea behind compressed sparse row block is to store data block by block, row by row. The CSRB storage format is very similar to CSR in data structure. It consists of three vectors. The first vector *vals* contains all non-zero values in the sparse matrix. The next vector *cols* stores the column indices of nonzero blocks of each row. The last vector *rows* stores the pointers to the beginning of each row of blocks. CSRB stores less number of column indices compared to CSR method.

This will improve indirect memory access time on the GPU. CSRB storage scheme is presented in Figure 3.4 for a block sparse matrix with 2×2 blocks.

3.2 Matrix vector multiplication parallel schemes

Matrix-vector multiplication (Ax = y) is the most computationally intensive operation used by iterative methods when solving sparse linear systems with simple preconditioning. For example, the Jacobi preconditioned conjugate gradient (PCG) method has one MVM operation per iteration which is the most time consuming part of the solution. It is a logical choice to focus on improving the performance of this operation for parallel computing. Two different approaches are implemented in this thesis to parallelize matrix vector multiplications. *Assembled* method is used for linear systems and $assemble\text{-}free$ is developed for systems with nonlinear stiffness matrix.

3.2.1 Assembled

The assembled method is the common matrix vector multiplication, applicable to linear systems iterative solvers. The name refers to the stiffness matrix being assembled before the multiplication takes place. In the solution procedure of a linear finite element analysis, we assemble the global stiffness matrix once, and then store it in our choice of sparse matrix format.

3.2.2 Assembly-free

The assembly-free matrix vector multiplication method applies to systems with nonlinear stiffness matrix. During the solution procedure of a nonlinear iterative solver, it is required to update the stiffness matrix, multiple times. Similar to linear system iterative solvers, matrix vector operations are also required in the solution

procedure. These two steps are the most time consuming parts of the solution process. Although considerable improvement in speed can be achieved by only utilizing aforementioned assembled matrix vector multiplication method, but the nonlinear iterative solver still suffers from assembling the global stiffness matrix at every iteration step. There is no escape for updating the stiffness matrix in a nonlinear system; however parallel algorithms can be utilized to reduce the computation time. There are two known solutions to this problem. In this thesis both solutions are developed and compared. In the first solution, assembly process of the stiffness matrix is parallelized and assembled parallel MVM technique is used in the iterative solver. In the second one, the assembly process is combined with matrix vector multiplication into one assembly-free parallel kernel.

3.3 Iterative solver

Applying finite element method to linear elastic structural problem results in linear system of equation in the form of $Ax = b$. When using elements such as TET4 and HEX8 for discretizing a 3-D problem, large number of elements are required in order to get an accurate solution. As mentioned in chapter 2, this approach will result in a system of equations with a very large and sparse matrix and iterative methods are the suitable choice for solving such systems. In iterative solution method, the solver tries to generate a better approximation of the solution vector in every other iteration. The approximation of the solution produces a residual vector. As the approximation gets closer to the exact solution, the residual values become smaller. Ideally the exact solution is achieved when the residual vector is zero, but due to the limited precision of real numbers in computers, achieving zero residual is not practical. After calculating the residual vector, the solver extracts a total residual

estimate out of the residual vector and uses this number as a convergence criterion. A tolerance is usually chosen for the stopping criterion of the algorithm.

3.3.1 Jacobi preconditioning

The rate of the convergence of the solution in iterative methods depends on the properties of the system matrix A. Preconditioned solvers, take advantage of the fact that the solution of the system defined by Eq.3.1 is identical to the solution of the original system. Matrix M is called the preconditioning matrix in Eq.3.1.

$$M^{-1}A\,x = M^{-1}\,b \tag{3.1}$$

Appropriate preconditioning of a system of equations can lead to faster convergence of the solution without jeopardizing the result. Using preconditioning technique introduces a calculation overhead in general cases. As an example, in preconditioned conjugate gradient method, this calculation overhead is one matrix-by-matrix multiplication and one matrix-by-vector operation, per iteration. The appropriately preconditioned system converges in fewer iterations compared to the original system, however the iteration cost is definitely higher and therefor care must be taken. Choosing the appropriate preconditioning matrix is more of an advanced topic in matrix algebra and iterative solvers and it is out of the scope of this thesis.

We utilized Jacobi preconditioning for our implementation because of its simplicity. Jacobi preconditioning matrix is a diagonal matrix containing the main diagonal of the original systems matrix A. Since the preconditioning matrix (M) is diagonal, inverse of M is simply another diagonal matrix containing inverse of the diagonal elements of matrix M. The Jacobi preconditioning method may not have the highest reduction in the number of required iterations, however, the fact that the matrix M^{-1} is diagonal, drops the calculation overhead which preconditioning

methods generally introduce to the solver. More precisely, the extra calculations are only one matrix-by-vector operation and one vector product, per iteration.

CHAPTER 4

GPU IMPLEMENTATION

4.1 Preconditioned conjugate gradient iterative solver

The Conjugate gradient algorithm is one of the best known iterative solvers for sparse symmetric positive definite linear systems. Conjugate gradient methods are well studied by mathematicians and they are categorized under Krylov subspace methods [18].The preconditioned conjugate gradient (PCG) method has been chosen as the iterative solver for the linear elastic structural problem in hand. The preconditioned conjugate gradient algorithm [18] [19] steps are listed under *Algorithm* 1. To achieve higher performance for the PCG method, one inner product per iteration is eliminated simply by reordering steps in the algorithm. An additional inner product operation is used to calculate the norm of the residual vector to gauge solution convergence. In order to study the rate of convergence of the PCG method for different grid sizes we also need to make the norm of the residual non-dimensional with respect to the number of unknowns which have direct relation with number of nodes. All vector quantities in the appear in pairs in the original algorithm for ease of understanding. It is not necessary to reserve extra memory for these quantities in the computer implementation. Instead of keeping both vectors in the memory, we use one vector and simply overwrite old values in the vector operations. Optimum implementation of PCG method is listed under *Algorithm* 2.

In GPU implementation of PCG iterative solver, memory transfers between host and device should be minimized. As mentioned earlier, matrix-vector multiplication is a common time consuming step in iterative methods. The Jacobi PCG method has

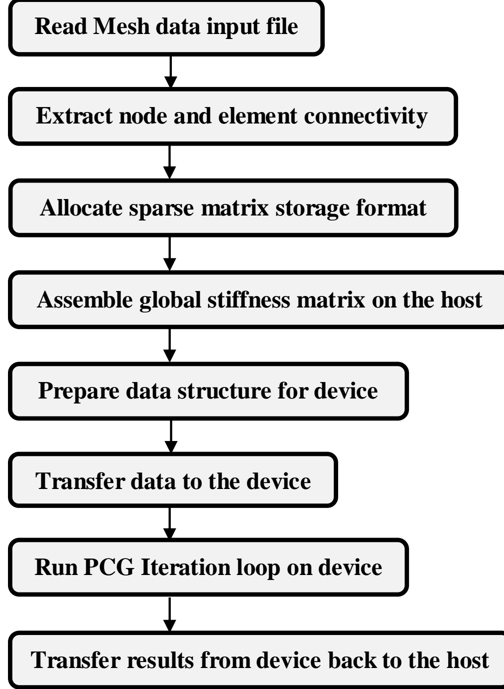

Figure 4.1. GPU preconditioned conjugate gradient flowchart.

one Matrix-vector multiplication ($A\,p$) per iteration. The first solution to accelerate the solver speed is to use the GPU device as a coprocessor for the host. In this case the device is used to carry on the matrix-vector multiplication and the CPU is responsible for the rest of the algorithm. Simply electing to utilize the GPU only for Matrix-vector operations, results in excessive number of memory transfers and hence poor solver performance. But if we also do the inner product operations on the GPU, there is no need to transfer data back and forth between host and device. Bolz et al [20] found that for conjugate gradient solver, increased performance could be realized by utilizing GPU to compute inner products as well as matrix-vector products. We chose standard NVIDIA cuBLAS library to carry out our inner products and level 1 blas operations [21]. Fig.4.1 presents the program flow of the PCG method implemented for the GPU.

Algorithm 1:Original PCG algorithm [18]

Compute $r_0 = b - Ax_0$, $z_0 = M^{-1}r_0$, and $p_0 = z_0$

For $j = 0, 1, ...,$ until convergenc Do

$\alpha_j = (r_j, z_j)/(Ap_j, p_j)$

$x_{j+1} = x_j + \alpha_j p_j$

$r_{j+1} = r_j - \alpha_j p_j$

$z_{j+1} = M^{-1}r_{j+1}$

$\beta_j = (r_{j+1}, z_{j+1})/(r_j, z_j)$

$p_{j+1} = z_{j+1} + \beta_j p_j$

$\alpha_i = (r_i, r_o)/(Ap_i, r_o)$

EndDo

Algorithm 2:Modified Implementation of PCG Algorithm

$r = Ax$

$r = b - r$

$z = M^{-1}r$

$p = z$

For $j = 0, 1, ...$ until convergence Do

If $j = 0$ Then

$temp_1 = (r, z)$

Else

$temp_1 = (r, z)$

$\beta = temp_1/temp_0$

$p = z + \beta p$

EndIf

$temp_2 = (p, A\,p)$

$\alpha_j = temp_1/temp_2$

$x = x + \alpha\, p$

$r = r\ - \alpha\, temp_2$

$z = M^{-1} r$

EndDo

4.2 Sparse matrix storage

Two different sparse matrix structures are implemented for the GPU device. The first is the compressed sparse row (CSR) method and the second is compressed sparse row block (CSRB) method. Details about the both data structures are explained in chapter 3. In order to increase data access efficiency in the matrix vector multiplication kernel both data structures are padded. In this process all the rows are padded with extra zeros to have the same length. The padding changes the CSR and CSRB into Ellpack storage scheme. The main reason behind using Ellpack method is the fixed row size. Fixed row length results in less indirect addressing and faster memory access. More details for each storage method are presented in the corresponding sub sections.

4.2.1 CSRB-Ellpack

Compressed sparse row block matrix storage is utilized to store the stiffness matrix of a 3-D linear elastic problem with 3 degree of freedoms per node. Each block is a 3×3 square matrix. In the global stiffness matrix, each row of blocks represents one node. A block with $row = i$ and $column = j$ indices, represents the coupling between *node* i and *node* j in all 3 degrees of freedom. The length of a row of blocks represents number of neighboring nodes including *node* i itself.

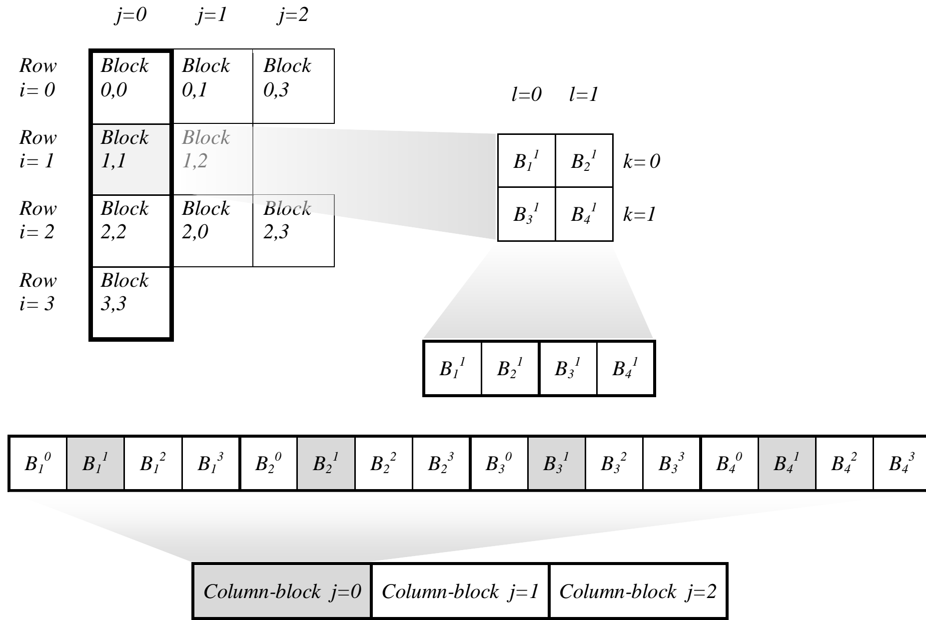

Figure 4.2. GPU compressed sparse row block data structure.

These neighbor counts are stored in a vector called NC. According to vector NC values, *node* i has $NC(i)$ blocks to be stored. The order we store the data is very important for memory access time considerations. Stride memory access pattern has to be avoided as a general principle. Optimum memory access is achieved when the stride is unit. The following principles are considered for storing the data:

- The first block of each row is the diagonal block
- Off-diagonal blocks are sorted by column index for each row
- Blocks are stored in a vector using a weaving method compatible with the GPU parallel memory access pattern

Figure 4.2 represents the order of storing blocks for the same example matrix in section 3.1.4 using weaving mechanism. Figure 4.3 shows the corresponding $vals$, $rows$ and $cols$ vectors for the padded CSRB and $vals$ after the weaving process. The example matrix is repeated for convenience. An appropriate reordering scheme is presented

for storing the data values. Weaving or reordering the data is depicted in Figure 4.2. In the weaving scheme, blocks are stored in groups of column-blocks. As mentioned earlier, the first block of each row is the diagonal block of the corresponding row number. All diagonal blocks are stored in the column-block $j = 0$. Eq.4.1 calculates the size of column blocks. Since all the rows are padded to have fixed number of blocks, all of the column-blocks have the same size.

$$column\text{-}block\ size = blocksize \times n \qquad n = number\ of\ nodes \tag{4.1}$$

The first element of each diagonal block is the first value to be stored. As it is shown in Figure 4.2, the first n values stored in the column-block array are the first elements of diagonal block in row order. In the case of 3 degree of freedoms, each column block consists of $9 \times n$ values. Eq.4.2 gives the memory location and the actual position of component (k, l) of the 2-D block (i, j).

$$\begin{aligned} global\ component\ location : row &= i \times dof + k \\ column &= cols[rows[i] + j] \times dof + l \\ location\ in\ vals : &(k \times dof + l) \times i + n \times blocksize \times j \end{aligned} \tag{4.2}$$

Storing the stiffness matrix with weaving scheme, enforces unit stride memory access and therefor it has the maximum performance. More detail about stride memory access pattern is provided in the GPU matrix-vector multiplication section.

4.3 Matrix vector multiplication parallel schemes

Parallel matrix-vector operation kernels are bandwidth limited. It means the memory bandwidth between GPU global memory and the computing cores in SMs, is the bottle neck of the performance. Hence any reduction in memory access results in higher performance. CSRB matrix format only stores row and column pointers of

$$A = \left[\begin{array}{cc|cc|cc|cc} 7 & 3 & 9 & 2 & 0 & 0 & 2 & 1 \\ 1 & 4 & 8 & 6 & 0 & 0 & 4 & 5 \\ \hline 0 & 0 & 3 & 7 & 5 & 2 & 0 & 0 \\ 0 & 0 & 1 & 9 & 8 & 6 & 0 & 0 \\ \hline 8 & 5 & 0 & 0 & 1 & 3 & 6 & 2 \\ 3 & 7 & 0 & 0 & 2 & 1 & 9 & 3 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 5 & 6 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & 8 \end{array}\right]$$

$$cols = \begin{bmatrix} 0 & 1 & 3 & 1 & 2 & 2 & 0 & 3 & 3 \end{bmatrix}$$

$$rows = \begin{bmatrix} 0 & 3 & 5 & 8 & 9 \end{bmatrix}$$

$$\begin{aligned} vlas = [\, & 7\ 3\ 1\ 4 \quad 9\ 2\ 8\ 6 \quad 2\ 1\ 4\ 5 \quad \ldots \\ & 3\ 7\ 1\ 9 \quad 5\ 2\ 8\ 6 \quad 0\ 0\ 0\ 0 \quad \ldots \\ & 1\ 3\ 2\ 1 \quad 8\ 5\ 3\ 7 \quad 6\ 2\ 9\ 3 \quad \ldots \\ & 5\ 6\ 2\ 8 \quad 0\ 0\ 0\ 0 \quad 0\ 0\ 0\ 0 \quad] \end{aligned}$$

$$\begin{aligned} vlas\ after\ weaving = [\, & 7\ 3\ 1\ 5 \quad 3\ 7\ 3\ 6 \quad 1\ 1\ 2\ 2 \quad 4\ 9\ 1\ 8 \quad \ldots \\ & 9\ 5\ 8\ 0 \quad 2\ 2\ 5\ 0 \quad 8\ 8\ 3\ 0 \quad 6\ 6\ 7\ 0 \quad \ldots \\ & 2\ 0\ 6\ 0 \quad 1\ 0\ 2\ 0 \quad 4\ 0\ 9\ 0 \quad 5\ 0\ 3\ 0 \quad] \end{aligned}$$

Figure 4.3. GPU compressed sparse row block example matrix.

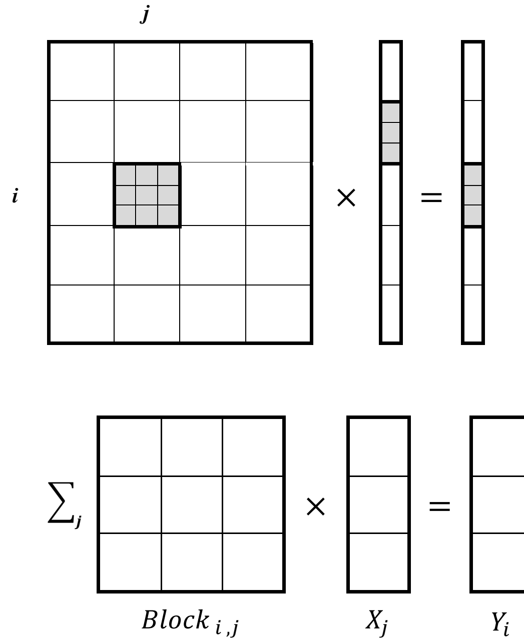

Figure 4.4. Block MVM.

blocks which not only save memory storage but it is the key to better performance in the matrix vector multiplication.

4.3.1 Assembled

Matrix-vector multiplication procedure is investigated for optimum implementation on the GPU. As being said, MVM is one of the most computationally intensive operations used by iterative methods when solving sparse linear systems. A GPU kernel is designed specially to handle CSRB MVM operations. In the FEM solution procedure we are interested in operations on the stiffness matrix. Stiffness matrix is stored in a sparse matrix format which unifies all degrees of freedom associated with each node in blocks. Figure 4.4 depicts the multiplication for a single block. The GPU CSRBMVM kernel assigns one node to one parallel thread to do the multiplication and return the results for all DOF associated with that node. As it is

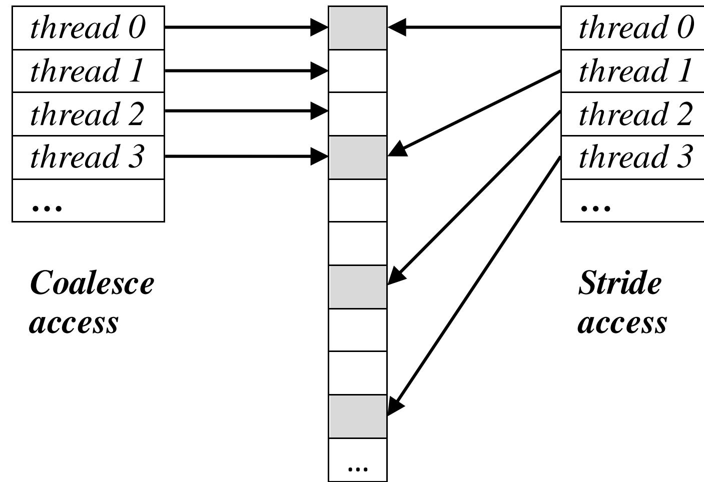

Figure 4.5. Stride and coalesce memory access.

shown in Figure 4.4, for a 3 DOF problem, same values of vector x have to be fetched from memory for each line of a block. Putting stiffness matrix in CSRB matrix format and assigning nodes instead of lines to the MVM kernel, eliminates redundant memory access to the vector x. Also CSRB matrix format reduces the time required for resolving row and column indices compared to CSR. This is due to less memory reads from $rows$ and $cols$ vector. Another way to interpret this is that the CSRB GPU kernel has higher instruction intensity compared to a CSR kernel. The ratio between instructions and memory read in CSRB is higher than the regular CSR. The parallel GPUMVM kernel tends to diverge in execution due to variable number of non-zero blocks per row. This means in each threadblock, all threads have to wait for the longest row to finish multiplying. One way to get around the thread block divergence is to schedule the nodes with the same row length together. It requires a simple reordering of the nodes based on number neighboring nodes. But scheduling nodes in a scattered pattern would have a penalty in accessing vector x.

When developing bandwidth limited GPU kernels, global coalesced memory access is the most important factor to consider. Coalesced memory access in simple words, means there is no gap or stride between requested data. Typically in parallel codes, multiple threads make requests to access different memory locations at the same time. If these requests happen to point to a series of memory locations, memory access will be so called coalesce. Un-coalesced memory access usually happens when there is a stride in the access pattern. Figure 4.5 depicts the two patterns.

In the CSRBMVM CUDA code we have to access global memory three times during execution. The first access is reading stiffness matrix data. We addressed this by reordering and padding our CSRB data structure similar to an Ellpack structure to assure global memory access coalescing. The second global memory access is reading vector x. Each parallel thread has to deal with multiple blocks and has to read 3 values from vector x per block. In general there is no guarantee to have coalesce reads from vector x for each row. However most of the mesh generators use re-numbering techniques to reduce stiffness matrix bandwidth. In CSRB case when we execute multiple threads which are accessing memory locations that are physically close to each other, the Fermi GPU will automatically line them up and manage the memory access the best possible way. Therefore memory access to vector x could be considered partially coalesce as long as we keep the neighboring nodes together or in other words keep the matrix bandwidth as low as possible. Back to the divergence of thread executions, we decided to avoid reordering the nodes. Since the kernel is bandwidth limited and not instruction limited, we believe the penalty of scattered memory access to vector x is higher than execution divergence. And finally the third global memory access is writing results back to vector y. Since each CUDA block multiplies consecutive rows at once and synchronize all threads, writing to vector y in global memory is fully coalesce.

4.3.2 Assembly-free

Assembly-free matrix multiplication refers to a procedure in which matrix multiplication is combined with calculation of stiffness matrix into one step. This unified task has higher performance in certain case studies compared to separated assembly and multiplication. The size of the global stiffness matrix in FEM problems is tied to the number of unknowns. For large FEM simulations, the size of the stiffness matrix becomes a limiting factor on memory resources. Assembly-free kernels reduce the memory space requirement per DOF and thus increase the capacity of the device. In the assembly process of stiffness matrices, local matrices are generated using the input mesh data and each local matrix is mapped to the correct corresponding locations in the global matrix. In concept, the stiffness matrix represents the same data provided in the mesh inputs in an expanded usable data structure. This expansion of data requires computations and extra memory space for storage. The assembly-free kernel takes mesh data as the input matrix, instead of the global stiffness matrix and therefore less memory space is required to do the same matrix-vector operation for the same problem size.

In a simple linear FEM problem, stiffness matrix is used many times during the solution process and hence storing the matrix avoids redundant calculations. For example the PCG iterative solver needs to access stiffness matrix for each matrix vector operation. On the other hand there are nonlinear FEM problems which require updating the stiffness matrix per iteration. Since the data is not reused, there is no need to store the whole matrix. In this case the assembly-free kernel has higher performance than the regular method.

The Assembly-free method is developed for simplex tetrahedral elements with 4 nodes. The GPU kernel parallel model maps one element to every thread. Each thread calculates the local stiffness matrix. Then it calculates the partial multiplication

result and store the result in a global result vector. Storing the partial results in the global result vector is similar to assembly process. To get correct results, one must be careful to avoid race condition. The race condition happens when multiple threads try to access the same global memory location simultaneously. This can happen if multiple threads try to insert the partial result of the same node number into the result vector. The general remedy to overcome race conditions are atomic operations (section 1.1.3.3). Since the atomic operations serialize memory access to the global memory, the performance will drop significantly and therefore the coloring technique instead. The coloring technique produces packets containing conflict-free elements. Two elements are conflict free when they have no common node. Each matrix-vector multiplication is broken down into packets of elements with the same color. Color packs are launched on separate GPU grids using sequential kernel calls. Utilizing coloring technique over atomic operations on global memory has proven to have better performance. The coloring step is done on the CPU as a preprocessing step. The element data is reordered to provide better memory access on the GPU. Coalesced global memory access is ensured by using extra memory to store redundant nodal coordinates. By using extra memory space we organize element data in separate packets. These independent packets provide coalesced global memory access and scalable parallel algorithm. The element data packets include node numbers and nodal coordinates for each element. The amount of extra memory usage is provided in the equation below:

$$\begin{aligned} Original\ memory\ for\ nodal\ coordinates &= n \times DOF \\ Extra\ memory\ for\ nodal\ coordinate &= m \times (nodes\ per\ element) \times DOF \\ n = number\ of\ nodes \qquad & m = number\ of\ elements \end{aligned} \tag{4.3}$$

CHAPTER 5

RESULTS

In this section the GPU is applied to the solution of systems of equations resulting from finite element discretization. The relative performance of the GPU to the CPU is reported as well as the GPU computing performance indices. All calculations are done in double precision. For the serial timing, C codes were compiled with the optimization flag "-$O3$" and run using a single core on a quad-core Intel i5 2.66GHz processor with 4GB of system memory. GPU computations were done in double precision utilizing CUDA C codes. An NVIDIA Quadro 5000 GPU with 352 cores and 2.5GB of memory was used as the GPU.

5.1 PCG on GPU

The cases considered here are composed of 8-node hexahedral elements for linear elastic problems. The local element stiffness matrices are formed in the conventional way explained in chapter 2. The local stiffness matrices are assembled into a global matrix with CRSB format and then boundary conditions are enforced. The aforementioned steps are completed on the CPU of the host computer while the solution of the system of equations is performed on the GPU using the PCG method described in previous chapters. Before executing the PCG function, the global stiffness matrix and force vector are copied to the global memory on the GPU. Timing functions are used to measure time required to perform a single matrix vector product and the time to perform a single PCG iteration. Similar timing functions were used in the serial version of the code running on the host CPU.

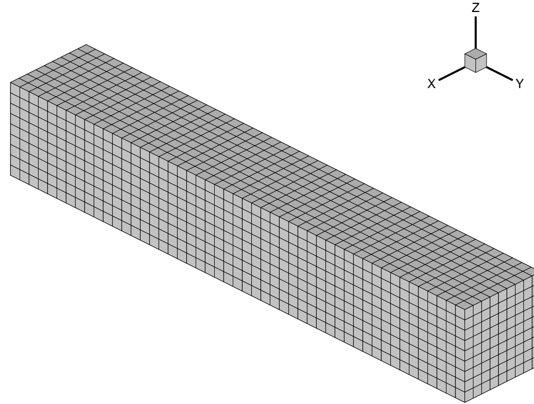

Figure 5.1. Mesh for cantilever beam.

Table 5.1. Meshes used for cantilever beam case

Grids	Total DOF	Nodes	Elements	Non-zeros blocks
1	3000	1000	624	19942
2	30000	10000	9019	233632
3	500,400	166,800	152,627	4,246,528
4	1,081,344	360,448	332,661	9,228,544
5	1,520,640	506,880	485,100	13,292,020
6	2,010,624	670,208	642,033	17,586,400

We define the speedup, which is a measure of performance of the GPU relative to the CPU, with the formula below:

$$speedup = \frac{cpu\ execution\ time}{gpu\ execution\ time} \tag{5.1}$$

The speedup is used to assess the GPU performance for a simple geometry cases. A cantilever beam is considered which is fixed at one end and has a uniform shear load applied at the other. This problem has a known exact solution and can be meshed with different resolutions quite easily. An example mesh is shown in Figure 5.1.

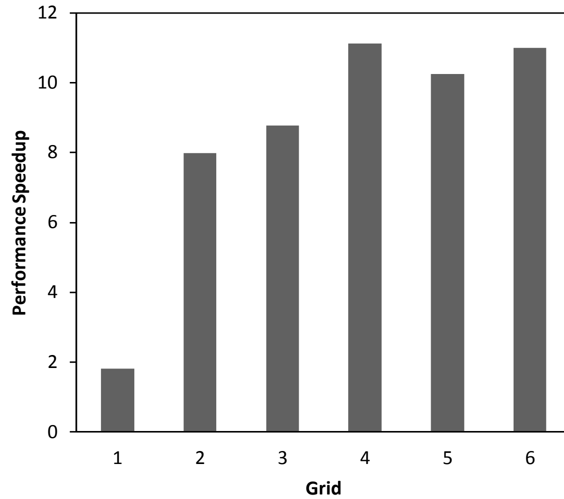

Figure 5.2. Speedup for PCG on GPU using full-matrix CSRB on both GPU and CPU for different mesh sizes.

The speedup is compared for the six different mesh sizes given in Table 5.1. The sizes range from 3000 to more than 2 million degrees of freedom (DOF). The speedup for the PCG solver for different grid sizes is shown in Figure 5.2. Note that in this case, both the GPU and CPU code use full matrix CSRB, although the global stiffness matrix is symmetric. Figure 5.3 compares the GPU speedup for the case where more conventional symmetric CSRB storage is used on the CPU side. The PCG results demonstrate the ability of the GPU CUDA code to significantly outperform the serially executed C code in both cases. For grid sizes above 1 million DOF, the CUDA implementation of the PCG algorithm shows the maximum and fairly constant speedup in performance. The memory bandwidth is the performance limiting factor. The small variation in performance gain for different grid sizes is normal due to parallel nature of the code. A parallel kernel call on the GPU schedules a grid of threadblocks. The speedup is sensitive to the total number of threadblocks to be launched. The GPU distribute the threadblocks among available computing resources which distinguish its heterogeneous architecture. A peak in speedup chart

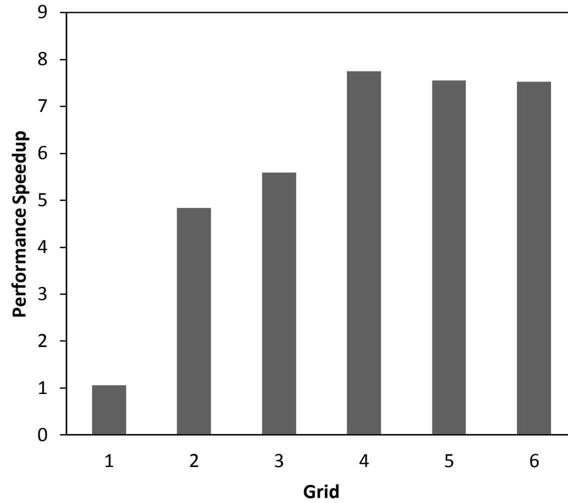

Figure 5.3. Speedup for PCG on GPU using full-matrix CSRB on GPU and symmetric CSRB on CPU for different mesh sizes.

appears if the problem size scales perfectly to the number of target GPU streaming multiprocessors.

5.2 Matrix-vector operation

The GPU results for this section are divided into two groups according to the algorithm used.

5.2.1 Assembled

The impact of CSRB verses non-block CSR on MVM operations is considered in this section. The same grids from Table.5.1 are used to generate the matrices. In this case, MVM speedup is defined as

$$speedup = \frac{CSR\ gpu\ execution\ time}{CSRB\ gpu\ execution\ time} \tag{5.2}$$

which effectively compares CSRB performance relative to CSR. Results are shown in Figure 5.4. For larger meshes, the CSRB, results in MVM operations that are five times faster than CSR MVM operations, which is most likely due to more efficient

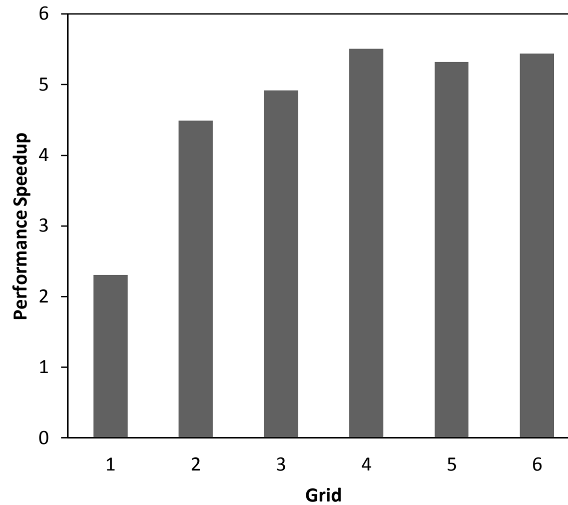

Figure 5.4. Speedup of MVM for CSRB vs CSR on GPU for different mesh sizes.

memory access explained in previous chapters. According to the results it is clear that the choice of CSRB is essential for getting maximum performance from the GPU hardware when solving matrices resulting from finite element discretization of multidimensional problems.

5.2.2 Assembly-free

To demonstrate the assembly-free multiplication method, the performance of 4 different scenarios are compared:

- Serial assembly and MVM on the CPU
- Serial assembly on the CPU and parallel MVM on the GPU
- Parallel assembly and MVM on the GPU
- Assembly-free MVM on the GPU

Multiple grid sizes were used to carry out the computations. Table .5.2 provides grid statistics of a rectangular beam structure meshed with 4-node tetrahedral elements.

Table 5.2. Meshes used for assembly-free performance study

Grids	Total DOF	Nodes	Elements	Non-zeros blocks
1	30,000	10,000	40,095	162,028
2	240,000	80,000	359,195	1,405,268
3	810,000	270,000	1,257,295	4,869,708
4	1,920,000	640,000	3,034,395	11,695,348
5	3,000,000	1,000,000	4,789,995	18,410,988
6	10,143,000	3381000	16,398540	62,819,236

In all cases, the matrix is stored in CSRB format and corresponding MVM code is utilized. In the first scenario, the CPU assembles the matrix and stores it on system memory. Then it calculates the matrix-vector operation. In the second scenario the matrix is assembled on the CPU in the same way and then the data is reordered and transferred to the GPU for calculating MVM in parallel. In the third case a parallel assembly algorithm is developed for the GPU. The matrix is first stored in the GPU global memory and then assembled MVM kernel is utilized. In the last case, mesh data is transferred to the GPU, and the assembly-free MVM kernel is used. The performance of all cases is measured using CPU and GPU time functions for one matrix-vector operation. In the last case the mesh data transfer from host to device is excluded from timing. Figure 5.5 demonstrates the relative performance of different scenarios while Table 5.3 lists actual computation time for each grid.

5.3 Conclusion

This work demonstrates the feasibility of using consumer GPUs to solve large sparse systems arising from 3-D finite element discretization of multidimensional equations. The approach exploits the sparse block structure of the global stiffness matrix to accelerate matrix vector multiplication and achieve higher performance compared

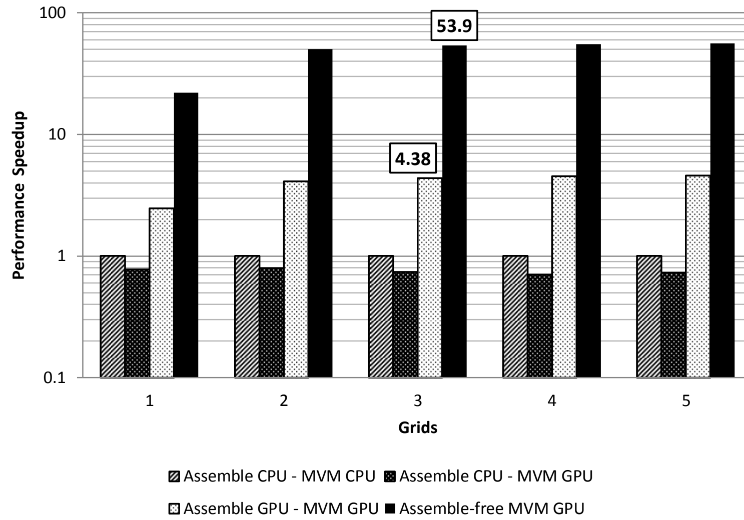

Figure 5.5. Assembly-free method performance chart.

Table 5.3. Details of computation time in millisecond

Grids	Assembly CPU	MVM CPU	GPU data preparation and transfer	MVM GPU	Assembly GPU	Assemble-Free MVM GPU
1	40	2	14	0.35	17	1.9
2	460	18	138	2.7	115.4	9.5
3	1660	65	665	9.7	393.5	32
4	4100	156	1900	22.7	943	77
5	6615	246	2712	35	1491	122.5
6	-	-	-	-	-	418

to CPU serial implementation. Compressed sparse row block multiplication kernels are developed to handle calculations. CSRB-MVM kernel is 5x faster than CSR on the GPU for field problems with 3 unknowns per node. Implementation of Jacobi preconditioned conjugate gradient iterative solver on the GPU with CSRB storage demonstrates up to 7x speedup in comparison with the fastest serial version of the code on CPU using symmetric compressed sparse row block structures.

Parallel assembling process and matrix vector multiplication are combined into one assembly-free kernel for 4-node tetrahedral element. The kernel achieved up to 50x performance speedup over CPU serial version and up to 12x performance gain over separated assembly and multiplication on the GPU. Using assembly-free method, matrix vector multiplication of a stiffness matrix with more than 10 million degrees of freedom was successfully done on the GPU with 2.5 GB memory while for the assembled method the largest problem size was 3 million DOF.

5.4 Future work

For the future work on the subject comparison of the matrix-vector operation of CSRB with standard Nvidia sparse library is suggested. Development of more sophisticated algorithms to handle unstructured grids matrix-vector operations are possible. Implementing different iterative solvers and using better preconditioning will definitely provide even faster solvers with fewer iterations to converge. Developing a nonlinear solver utilizing the provided assembly-free kernel would demonstrate the applicability of the method. Developing a faster assembly-free kernel is possible utilizing the shared memory architecture of the GPU. Texture memory spaces could be utilized to provide more efficient memory access for all the kernels.

REFERENCES

[1] NVIDIA, *CUDA C programming guide v4.0*, 2011.

[2] P. Zaspel and M. Griebel, "Solving incompressible two-phase flows on multi-gpu clusters," *Computers & Fluids*, 2012.

[3] J. Appleyard and D. Drikakis, "Higher-order cfd and interface tracking methods on highly-parallel mpi and gpu systems," *Computers & Fluids*, vol. 46, no. 1, pp. 101–105, 2011.

[4] I. Kampolis, X. Trompoukis, V. Asouti, and K. Giannakoglou, "Cfd-based analysis and two-level aerodynamic optimization on graphics processing units," *Computer Methods in Applied Mechanics and Engineering*, vol. 199, no. 9-12, pp. 712–722, 2010.

[5] G. Joldes, A. Wittek, and K. Miller, "Real-time nonlinear finite element computations on gpu-application to neurosurgical simulation," *Computer methods in applied mechanics and engineering*, vol. 199, no. 49-52, pp. 3305–3314, 2010.

[6] D. Komatitsch, G. Erlebacher, D. Goddeke, and D. Michéa, "High-order finite-element seismic wave propagation modeling with mpi on a large gpu cluster," *Journal of Computational Physics*, vol. 229, no. 20, pp. 7692–7714, 2010.

[7] J. Xue, X. Jiao, Y. Deng, H. Qian, D. Zeng, G. Li, and Z. Yu, "Massively parallel finite element simulator for full-chip sti stress analysis," in *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*. IEEE, 2010, pp. 1196–1201.

[8] Z. Taylor, M. Cheng, and S. Ourselin, "High-speed nonlinear finite element analysis for surgical simulation using graphics processing units," *Medical Imaging, IEEE Transactions on*, vol. 27, no. 5, pp. 650–663, 2008.

[9] A. Dziekonski, A. Lamecki, and M. Mrozowski, "Gpu acceleration of multilevel solvers for analysis of microwave components with finite element method," *Microwave and Wireless Components Letters, IEEE*, no. 99, pp. 1–1, 2011.

[10] M. Papadrakakis, G. Stavroulakis, and A. Karatarakis, "A new era in scientific computing: Domain decomposition methods in hybrid cpu-gpu architectures," *Computer Methods in Applied Mechanics and Engineering*, 2011.

[11] Y. Cai, G. Li, H. Wang, G. Zheng, and S. Lin, "Development of parallel explicit finite element sheet forming simulation system based on gpu architecture," *Advances in Engineering Software*, 2011.

[12] A. Cevahir, A. Nukada, and S. Matsuoka, "High performance conjugate gradient solver on multi-gpu clusters using hypergraph partitioning," *Computer Science-Research and Development*, vol. 25, no. 1, pp. 83–91, 2010.

[13] L. Buatois, G. Caumon, and B. Lévy, "Concurrent number cruncher: An efficient sparse linear solver on the gpu," in *High performance computing and communications: third international conference, HPCC 2007, Houston, USA, September 26-28, 2007: proceedings*, vol. 4782. Springer-Verlag New York Inc, 2007, p. 358.

[14] NVIDIA, "Next generation cuda compute architecture." [Online]. Available: http://www.nvidia.com/content/PDF/fermi_white_papers

[15] ——, *CUDA C Best Practice Guide v4.0*, 2011.

[16] O. Zienkiewicz, R. Taylor, and J. Zhu, *The finite element method: its basis and fundamentals*. Elsevier Butterworth-Heinemann, 2005, vol. 1.

[17] S. Rao, *The finite element method in engineering.* Butterworth-Heinemann, 2010.

[18] Y. Saad, *Iterative Methods for Sparse Linear Systems.* Society for Industrial and Applied Mathematics, 2003.

[19] "An introduction to the conjugate gradient method without the agonizing pain."

[20] J. Bolz, I. Farmer, E. Grinspun, and P. Schroder, "Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid," *ACM Transactions on Graphics (TOG)*, vol. 22, no. 3, pp. 917–924, July 2003.

[21] NVIDIA, *CUBLAS Library*, 2011.

BIOGRAPHICAL STATEMENT

Ashkan Akbariyeh was born in Tehran, Iran, in 1987. He received his B.S. degree from Iran University of Science and Technology in 2010 and his M.S. degree from The University of Texas at Arlington in 2012, both in Mechanical Engineering. From 2009 to 2010 he was working in the field of multibody dynamics computer simulation in Tehran. In 2011 he joined the CFD lab at MAE department in Arlington to pursue computational mechanics. His current research interest is developing device compatible accurate parallel algorithms for FEA solution acceleration.