Scientific Computation on Graphics Processing Unit using CUDA

Submitted in partial fulfillment of the requirements
of the degree of

Master of Technology

by

Pradip Narendrakumar Panchal
(Roll No. 09307406)

Under the guidance of
Prof. Sachin Patkar

DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY BOMBAY
2011

Dissertation Approval

The dissertation entitled

Scientific Computation on Graphics Processing Unit using CUDA

by

Pradip Narendrakumar Panchal
(Roll No : 09307406)

is approved for the degree of

Master of Technology

Examiner	Examiner
Guide	Chairman

Date: ____________

Place: ____________

Declaration

I declare that this written submission represents my ideas in my own words and where other's ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

____

(Signature)

____

(Name of the student)

____

(Roll No.)

Date: ________________

Abstract

The Partial Differential Equations (PDEs) play major role in mathematical modeling of problems in engineering and science. The engineering disciplines such as electro-magnetics and fluid dynamics use PDEs heavily and development of products in these engineering fields employs computational intensive numerical methods. These computational intensive methods takes reasonable amount of time on state of the art computers. Nowadays a Graphics Processing Unit (GPU) offers state of the art parallel computing resources and alternative for supercomputing power on desktop computer. In this report various numerical methods for PDEs have been implemented on GPU hardware using NVIDIA's Compute Unified Device Architecture (CUDA). The simulation of propagation of Gaussian pulse in 2-dimension is implemented on GPU using finite difference time domain method. The computation of eigenvalue and eigenvector involving inverse-iteration, Lanczos and bisection method is implemented on GPU for time dependent Schrödinger equation. The Navier-Stokes equations is solved by finite difference method. Various iterative methods for systems of linear equation in finite difference methods are accelerated on GPU. An unstructured grid based finite volume solver is implemented on GPU. Various optimization and renumbering techniques are employed to achieve the average speed-up of 27x for double precision over CPU implementation.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Partial differential equations (PDEs) are used for mathematical modeling of problems in engineering and science. The analytical methods or exact solution of partial differential equation are good for simple problems in real-life. The analytical methods are not suitable for large problems with complex solution region, mixed type or time-varying dependent boundary conditions, either inhomogeneous or anisotropic medium.

Numerical Methods or approximate solutions of PDEs are suitable where PDE is non-linear or application of analytical method is difficult. Numerical methods for PDE have been explored rapidly with the development of digital computers. State of the art digital computers have been employed to solve computational intensive application in the field of computational electromagnetic and Computational Fluid Dynamics (CFD). Following methods are among the most commonly used in engineering applications.

1. Finite Difference Methods (FDM) - based on approximating differential operators with difference operators

2. Finite Element Methods (FEM) - based on a discretization of the

space of solutions

3. Finite Volume Method (FVM) - based on dividing the domain in many small domains

4. method of lines - reduces the PDE to a large system of ordinary differential equations

Most of the above techniques employed discretization of the solution region in to element, volume or nodes and constructing the linear systems of equations. Normally the numerical program spends most of it's execution time in solving these linearized equations. Thus efficiency of solution method as whole depends mainly on the efficiency of the linear equation solvers. Some of the linear equation solvers have been implemented in parallel computing resources and simulation time has been reduced considerable. Nowadays Graphics Processing Unit (GPU) has been employed for parallel computing. During this work several computational intensive problems are identified in Computational Fluid Dynamics (CFD), Computational Electromagnetic and eigenvalue-value problem for Schrödinger equation. Following chapters discuss techniques of solving PDEs and various computation aspects on GPU. Second chapter discusses the Compute Unified Device Architecture (CUDA). Third chapter discusses the simulation of Navier-Stokes equations for laminar flows of viscous, incompressible fluids by finite difference method. Fourth chapter discusses unstructured grid based finite volume solver for compressible, inviscid fluid using AUSM$^+$-UP technique. Fifth chapter discusses Finite Difference Time Domain (FDTD) method for Maxwell equation. Sixth chapter discusses the eigenvalue value problem for Schrödinger equation in quantum mechanics.

Chapter 2

Graphics Processing Units

Earlier Graphics Processing Units (GPUs) were used to render the abstract geometric objects created by various programs running on the CPU so they could be displayed to the user. The GPU includes programmable shader units namely vertex shader to process vertices's of 3-dimensional objects supplied by user and Fragment or Pixel Shader to apply colors from textures as well as additional complex effects like scene lightning to each fragment/pixel. Both of these shading units are programmable and parallel. With advance of semiconductor industry above highly parallel processing units are placed in one single central unit, called Unified Shading Units combining all the features on a higher abstraction level. These Unified Shading Unit consists of a group of parallel processing units (often called multiprocessors), each composed of several stream processors along with special function units for memory access, instruction fetching and scheduling. While the first shader worked in a single-instruction multiple-data (SIMD) fashion, later on support for branching was added So the new streaming processors are best described as single-program multiple-data (SPMD) units [1].

2.1 CUDA Hardware Architecture

A GPU is an example of a Single Instruction, Multiple Data (SIMD) multiprocessor. In the CUDA programming model, compute-intensive tasks of an application are grouped into an instruction set and passed on to the GPU such that each thread core works on different data but executes the same instruction [2]. The CUDA memory hierarchy is almost same to the one for a conventional multiprocessor. Closer to the core, the local registers allow fast arithmetic and logical operations. The shared memory, seen by all the cores of a single multiprocessor, can be compared to a first-level cache (L1), as it provides a memory closer to the processors that will be used to store data that tend to be used over time by any core. The difference in CUDA is that the programmer is responsible for the management of this GPU cache. The last level in this hierarchy is the global memory, the RAM of the device shown in Fig. 2.1. It can be accessed by any processor of the GPU, but for a higher latency cost. Threads can actually perform simultaneous scatter or simultaneous gather operations if those addresses are aligned in memory and called coalesced memory access [2]. Coalesced memory access is crucial for superior kernel performance as it hides the latency of the global memory. Each multiprocessor also has read-only constant cache and texture cache. The constant cache can be used by the threads of a multiprocessor when trying to read the same constant value at the same time. Texture cache on the other hand is optimized for 2-D spatial locality and should be preferred over global device memory when coalesced read cannot be achieved [2].

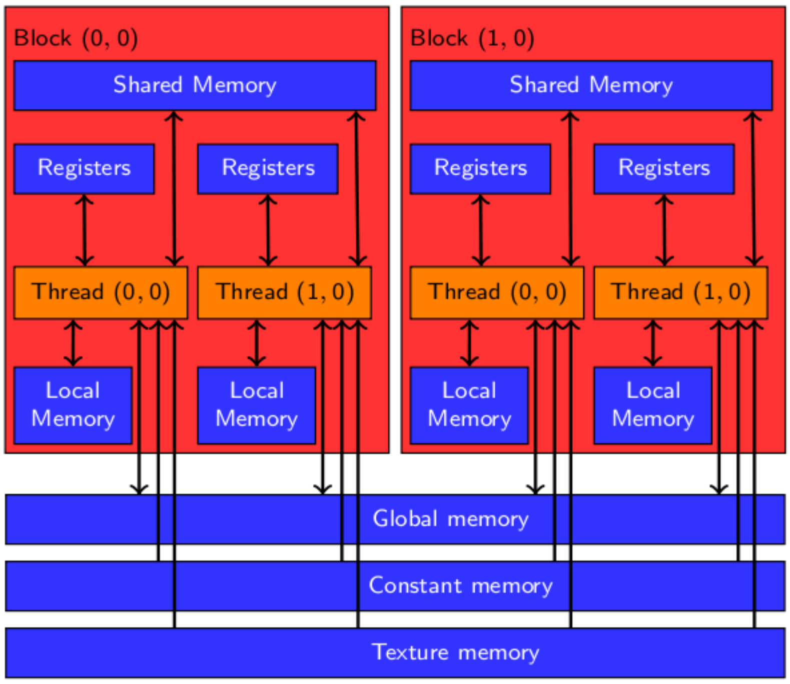


Figure 2.1: CUDA enabled Graphics Processing Unit's memory model

2.2 CUDA Programming Model

The computation core of the CUDA programming is the kernel, a set of instruction having some inherent parallelism, which is passed on to the GPU and executed by all the processor units, using different data streams. Each kernel is launched from the host side (CPU), and it is mapped to a thread grid on the GPU. Each grid is composed of thread blocks. All the threads from a particular block have access to the same shared memory and can synchronize together. On the other hand, threads from different blocks cannot synchronize and can exchange data only through the global (device) memory. A single block can only contain a limited number of threads, depending on the device model. But different blocks can be executed in parallel. Blocks executing the same kernel are batched together into a grid. Blocks are managed by CUDA and executed in parallel in a batch mode. The programmer needs to define the number of threads per block and the grid size (number of blocks) before launching the kernel. The CUDA programming model is supported by CUDA Runtime API (Application Programming Interface) and CUDA Driver API [2]. These API provides various high level and low level functions to dynamically allocate memory, free memory, transfer the data between host and device or between devices in both direction. It includes instruction for synchronizing the execution of kernels and threads within a kernels. In addition to the above, NVIDIA provides CUBLAS and CUFFT library to handle various of linear algebraic and Fast Fourier Transformation operations respectively.

The challenge for a CUDA software developer is then, not only the parallelization of the code, but also the optimization of the memory accesses by making the best use of the shared memory and the coalesced access to the global (device) memory.

Part I

Computational Fluid Dynamics on GPU Hardware

Chapter 3

CFD Solver based on Finite Difference Method

Computational Fluid Dynamics (CFD) methods are concerned with the solution of equations of motion of the fluid as well as with the interaction of the fluid with solid bodies for real-life problems. The real-life fluid problems are difficult to solve by analytical methods. The mathematical equations (systems of differential equations or integral equations), modeling of real-life fluids problems, are discretized in grid or finite-dimensional spaces. Then the underlying continuous equations are solved approximately. The data of approximate solution is processed by visualization techniques for interpretation and co-related with experimental data if available. With the development of digital computers the numerical computation of fluid dynamics have evolved fast, but most of the CFD programs runs slow on state of the art supercomputers. Recently introduced programmable GPUs have provided very good parallel computing resources to desktop computers at much cheaper rate than supercomputers. This chapter discusses the numerical simulation of laminar flows of viscous, incompressible fluids by mathematical model named Navier-Stokes equations and next chapter discusses inviscid compressible fluid flow by AUSM$^+$-UP techniques based Finite Volume Method (FVM).

3.1 Numerical Simulation of Navier-Stokes Equations

The laminar flows of viscous, incompressible fluids is mathematically modeled by Navier-Stokes equations. The flow of fluid in a region $\Omega \subset \mathbb{R}^N, (N \in 2, 3)$ throughout time $t \in [0, t_{end}]$ is characterized by the following quantities and systems of partial differential equations [3].

1. $\vec{u} : \Omega \times [0, t_{end}] \to \mathbb{R}^N$ velocity field,

2. $p : \Omega \times [0, t_{end}] \to \mathbb{R}$ pressure,

3. $\varrho : \Omega \times [0, t_{end}] \to \mathbb{R}$ density,

4. Momentum equation,

$$\frac{\partial}{\partial t}\vec{u} + (\vec{u} \cdot \nabla)\vec{u} + \nabla p = \frac{1}{R_e}\triangle\vec{u} + \vec{g} \tag{3.1}$$

5. Continuity equation,

$$\nabla \cdot \vec{u} = 0 \tag{3.2}$$

Here the quantity $R_e \in \mathbb{R}$ is the dimensionless Reynolds number, and $\vec{g} \in \mathbb{R}^N$ denotes body forces such as gravity acting throughout the bulk of the fluid.

3.1.1 Discretizing the Navier-Stokes Equations in 2-D

In two-dimensional case ($N = 2$) for $\vec{x} = (x, y)^T$, $\vec{u} = (u, v)^T$, $\vec{g} = (g_x, g_y)^T$, Eq. (3.1) and Eq. (3.2) become initial-boundary value problem and follow as below:

Momentum equations:

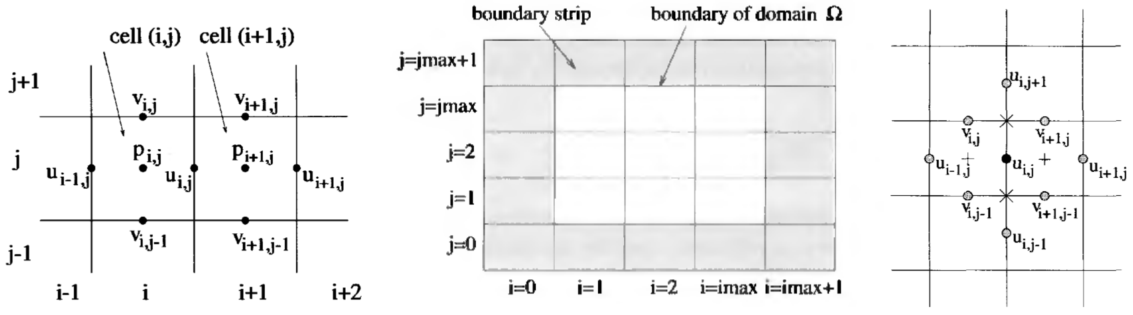
$$\frac{\partial u}{\partial t} + \frac{\partial p}{\partial x} = \frac{1}{R_e}\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) - \frac{\partial (u^2)}{\partial x} - \frac{\partial (uv)}{\partial y} + g_x \tag{3.3}$$

$$\frac{\partial v}{\partial t} + \frac{\partial p}{\partial y} = \frac{1}{R_e}\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right) - \frac{\partial (uv)}{\partial x} - \frac{\partial (v^2)}{\partial y} + g_y \tag{3.4}$$

Continuity equation:

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0. \tag{3.5}$$

The detailed derivation of Navier-Stokes equations is given in numerical



(a) Staggered grid (b) domain with boundary cells (c) discretization of the u-momentum equations

Figure 3.1: Structured grid for finite difference solver

simulation in fluid dynamics [3]. In solving the Navier-Stokes equations, the region Ω is discretized using a staggered grid, in which the different unknown variables are not located at the same grid points. Thus the discrete values of u, v and p are actually located on three separate grids; each shifted by half a grid spacing to the bottom, to the left, and to the lower left, respectively. The staggered grid, domain with boundary cells and discretized values for finite difference equation are shown in Fig. 3.1 Applying central difference and donor-cell discretization [4], we obtain the following discrete expressions.

For Eq. (3.3) for u at the midpoint of the right edge of cell (i,j), $i = 1...i_{max} - 1, j = 1...j_{max}$, we get,

$$\begin{aligned}
\left[\frac{\partial(u^2)}{\partial x}\right]_{i,j} &:= \frac{1}{\delta x}\left(\left(\frac{u_{i,j}+u_{i+1,j}}{2}\right)^2 - \left(\frac{u_{i-1,j}+u_{i,j}}{2}\right)^2\right)\\
&\quad +\gamma\frac{1}{\delta x}\left(\frac{|u_{i,j}+u_{i+1,j}|}{2}\frac{(u_{i,j}-u_{i+1,j})}{2} - \frac{|u_{i-1,j}+u_{i,j}|}{2}\frac{(u_{i-1,j}-u_{i,j})}{2}\right) \qquad (3.6)\\
\left[\frac{\partial(uv)}{\partial y}\right]_{i,j} &:= \frac{1}{\delta y}\left(\left(\frac{(v_{i,j}+v_{i+1,j})}{2}\frac{(u_{i,j}+u_{i,j+1})}{2}\right) - \left(\frac{(v_{i,j-1}+v_{i+1,j-1})}{2}\frac{(u_{i,j-1}+u_{i,j})}{2}\right)\right)\\
&\quad +\gamma\frac{1}{\delta y}\left(\frac{|v_{i,j}+v_{i+1,j}|}{2}\frac{(u_{i,j}-u_{i,j+1})}{2} - \frac{|v_{i,j-1}+v_{i+1,j-1}|}{2}\frac{(u_{i,j-1}-u_{i,j})}{2}\right) \qquad (3.7)\\
\left[\frac{\partial^2 u}{\partial x^2}\right]_{i,j} &:= \frac{u_{i+1,j}-2u_{i,j}+u_{i-1,j}}{(\delta x)^2}, \qquad \left[\frac{\partial^2 u}{\partial y^2}\right]_{i,j} := \frac{u_{i,j+1}-2u_{i,j}+u_{i,j-1}}{(\delta y)^2} \qquad (3.8)\\
\left[\frac{\partial p}{\partial x}\right]_{i,j} &:= \frac{p_{i+1,j}-p_{i,j}}{\delta x} \qquad (3.9)
\end{aligned}$$

Similarly for Eq. (3.4) for v at the midpoint of the upper edge of cell (i,j), $i = 1...i_{max}, j = 1...j_{max} - 1$, we get,

$$\begin{aligned}
\left[\frac{\partial(v^2)}{\partial y}\right]_{i,j} &:= \frac{1}{\delta y}\left(\left(\frac{v_{i,j}+v_{i,j+1}}{2}\right)^2 - \left(\frac{v_{i,j-1}+v_{i,j}}{2}\right)^2\right)\\
&\quad +\gamma\frac{1}{\delta y}\left(\frac{|v_{i,j}+v_{i,j+1}|}{2}\frac{(v_{i,j}-v_{i,j+1})}{2} - \frac{|v_{i,j-1}+v_{i,j}|}{2}\frac{(v_{i,j-1}-v_{i,j})}{2}\right)
\end{aligned} \qquad (3.10)$$

$$\begin{aligned}
\left[\frac{\partial(uv)}{\partial x}\right]_{i,j} &:= \frac{1}{\delta x}\left(\left(\frac{(u_{i,j}+u_{i,j+1})}{2}\frac{(v_{i,j}+v_{i+1,j})}{2}\right) - \left(\frac{(u_{i-1,j}+u_{i-1,j+1})}{2}\frac{(v_{i-1,j}+v_{i,j})}{2}\right)\right)\\
&\quad +\gamma\frac{1}{\delta x}\left(\frac{|u_{i,j}+u_{i,j+1}|}{2}\frac{(v_{i,j}-v_{i+1,j})}{2} - \frac{|u_{i-1,j}+u_{i-1,j+1}|}{2}\frac{(v_{i-1,j}-v_{i,j})}{2}\right)
\end{aligned} \qquad (3.11)$$

$$\left[\frac{\partial^2 v}{\partial x^2}\right]_{i,j} := \frac{v_{i+1,j} - 2v_{i,j} + v_{i-1,j}}{(\delta x)^2}, \qquad \left[\frac{\partial^2 v}{\partial y^2}\right]_{i,j} := \frac{v_{i,j+1} - 2v_{i,j} + v_{i,j-1}}{(\delta y)^2} \tag{3.12}$$

$$\left[\frac{\partial p}{\partial y}\right]_{i,j} := \frac{p_{i,j+1} - p_{i,j}}{\delta y} \tag{3.13}$$

The parameter γ in the above formulas lies between 0 and 1 [4].

Now for discretization of the time derivatives $\delta u/\delta t$ and $\delta v/\delta t$, the time interval $[0, t_{end}]$ is discretized into equal subintervals $[n\delta t, (n+1)\delta t], n = 0..., t_{end}/\delta t - 1$. Using $Euler's$ method for first-order difference quotations,

$$\left[\frac{\partial u}{\partial t}\right]^{(n+1)} := \frac{u^{(n+1)} - u^{(n)}}{\delta t}, \qquad \left[\frac{\partial v}{\partial t}\right]^{(n+1)} := \frac{v^{(n+1)} - v^{(n)}}{\delta t} \tag{3.14}$$

where the subscript (n) denotes the time level.

3.1.2 Algorithm

Substituting the time discretization Eq. (3.14) of the terms $\frac{\partial u}{\partial t}$ and $\frac{\partial v}{\partial t}$ in the momentum equations Eq. (3.3) and Eq. (3.4), we get following equation. This equations are known as *time discretization of the momentum equations* Eq. (3.3) and (3.4)

$$u^{(n+1)} = F^{(n)} - \delta t \frac{\partial p^{(n+1)}}{\partial x}, \qquad v^{(n+1)} = G^{(n)} - \delta t \frac{\partial p^{(n+1)}}{\partial y} \tag{3.15}$$

where $F^{(n)}$ and $G^{(n)}$ are evaluated at time level n with following F and G

$$\begin{aligned} F &:= u^{(n)} + \delta t \left[\frac{1}{R_e}\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) - \frac{\partial (u^2)}{\partial x} - \frac{\partial (uv)}{\partial y} + g_x\right] \\ G &:= v^{(n)} + \delta t \left[\frac{1}{R_e}\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right) - \frac{\partial (uv)}{\partial x} - \frac{\partial (v^2)}{\partial y} + g_y\right] \end{aligned} \tag{3.16}$$

This type of discretization may be characterized as being *explicit* in the velocities and *implicit* in the pressure; i.e, the velocity field at time step t_{n+1} can be computed once the corresponding pressure is known. The pressure is determined by evaluating the continuity equation Eq. (3.5) at time t_{n+1}. Now substituting the relationship of Eq. (3.15) for the velocity field $(u^{(n+1)}, v^{(n+1)})^T$ into the continuity equation Eq. (3.5) we get a *Poisson equations for the pressure* $p^{(n+1)}$ at time t_{n+1}.

$$\frac{\partial^2 p^{(n+1)}}{\partial x^2} + \frac{\partial^2 p^{(n+1)}}{\partial y^2} = \frac{1}{\delta t}\left(\frac{\partial F^{(n)}}{\partial x} + \frac{\partial G^{(n)}}{\partial y}\right) \tag{3.17}$$

The fully discrete momentum equations will be obtained by discretization of the spatial derivatives occurring in the time-discretized momentum equations Eq. (3.15) with the help of Eq. (3.6) to (3.9) and Eq. (3.10) to (3.13),

$$\begin{aligned} u_{i,j}^{(n+1)} &= F_{i,j}^{(n)} - \frac{\delta t}{\delta x}(p_{i+1,j}^{(n+1)} - p_{i,j}^{(n+1)}); \qquad i = 1, ..., i_{max} - 1, \qquad j = 1, ..., j_{max} \\ v_{i,j}^{(n+1)} &= G_{i,j}^{(n)} - \frac{\delta t}{\delta y}(p_{i,j+1}^{(n+1)} - p_{i,j}^{(n+1)}); \qquad i = 1, ..., i_{max}, \qquad j = 1, ..., j_{max} - 1 \end{aligned} \tag{3.18}$$

The quantities F and G from Eq. (3.16) are discretized at the right and upper edges of cell (i, j), respectively

$$F_{i,j} := u_{i,j} + \delta t\Bigg[\frac{1}{R_e}\Bigg(\left[\frac{\partial^2 u}{\partial x^2}\right]_{i,j} + \left[\frac{\partial^2 u}{\partial y^2}\right]_{i,j}\Bigg) - \left[\frac{\partial (u^2)}{\partial x}\right]_{(i,j)} - \left[\frac{\partial (uv)}{\partial y}\right]_{(i,j)} + g_x\Bigg]$$

$$i = 1, ..., i_{max} - 1, \qquad j = 1, ..., j_{max}$$

$$G_{i,j} := v_{i,j} + \delta t\Bigg[\frac{1}{R_e}\Bigg(\left[\frac{\partial^2 v}{\partial x^2}\right]_{i,j} + \left[\frac{\partial^2 v}{\partial y^2}\right]_{i,j}\Bigg) - \left[\frac{\partial (v^2)}{\partial y}\right]_{(i,j)} - \left[\frac{\partial (uv)}{\partial x}\right]_{(i,j)} + g_y\Bigg]$$

$$i = 1, ..., i_{max}, \qquad j = 1, ..., j_{max} - 1 \tag{3.19}$$

Substituting the discrete quantities introduced in Eq. (3.17), Poisson equation for the pressure, we get discrete Poisson equation as below,

$$\frac{p^{(n+1)}_{i+1,j} - 2p^{(n+1)}_{i,j} + p^{(n+1)}_{i-1,j}}{(\delta x)^2} + \frac{p^{(n+1)}_{i,j+1} - 2p^{(n+1)}_{i,j} + p^{(n+1)}_{i,j-1}}{(\delta y)^2} = \frac{1}{\delta t}\left(\frac{F^{(n)}_{i,j} - F^{(n)}_{i-1,j}}{\delta x} + \frac{G^{(n)}_{i,j} - G^{(n)}_{i,j-1}}{\delta y}\right)$$

$$i = 1, ..., i_{max}, \qquad j = 1, ..., j_{max} \tag{3.20}$$

Applying the boundary conditions in [3] above equation becomes as follow,

$$\frac{\epsilon^E_i(p^{(n+1)}_{i+1,j} - p^{(n+1)}_{i,j}) - \epsilon^W_i(p^{(n+1)}_{i,j} - p^{(n+1)}_{i-1,j})}{(\delta x)^2} + \frac{\epsilon^N_j(p^{(n+1)}_{i,j+1} - p^{(n+1)}_{i,j}) - \epsilon^S_j(p^{(n+1)}_{i,j} - p^{(n+1)}_{i,j-1})}{(\delta y)^2} = \frac{1}{\delta t}\left(\frac{F^{(n)}_{i,j} - F^{(n)}_{i-1,j}}{\delta x} + \frac{G^{(n)}_{i,j} - G^{(n)}_{i,j-1}}{\delta y}\right)$$

$$i = 1, ..., i_{max}, \qquad j = 1, ..., j_{max} \tag{3.21}$$

The parameters,

$$\epsilon_i^W := \begin{cases} 0 & i = 1, \\ 1 & i > 1, \end{cases} \qquad \epsilon_i^E := \begin{cases} 1 & i < i_{max}, \\ 0 & i = i_{max}, \end{cases}$$

$$\epsilon_j^S := \begin{cases} 0 & j = 1, \\ 1 & j > 1, \end{cases} \qquad \epsilon_j^N := \begin{cases} 1 & j < j_{max}, \\ 0 & j = j_{max}, \end{cases}$$

As a result, Eq. (3.21) represents a linear system of equations containing $i_{max}j_{max}$ equations and $i_{max}j_{max}$ unknowns $p_{ij}, i = 1...i_{max}, j = 1...j_{max}$ to be solved using a suitable algorithm. For the solution of the very large, sparse linear systems of equations arising from the discretization of partial differential equations, iterative solution methods are generally applied. Various iterative solutions methods have been discussed in subsequent sections. These iterative solution methods have been ported on graphics processing unit for acceleration. For stability of numerical algorithm and elimination of oscillations, stability have been imposed on the step-sizes δx, δy and δt. An adaptive step-size control has been implemented by selecting δt for the next time step as below,

$$\delta t := \tau min\left(\frac{R_e}{2}\left(\frac{1}{\delta x^2} + \frac{1}{\delta y^2}\right)^{-1}, \frac{\delta x}{|u_{max}|}, \frac{\delta y}{|v_{max}|}\right) \tag{3.22}$$

The factor $\tau \in [0, 1]$ is a safety factor.

The basic algorithm [3] is shown below,

Algorithm 1 :- Algorithm for solving Navier-Stokes equation

```
1: Set $t := 0, n := 0$
2: Assign initial values to $u$, $v$, $p$
3: while $t < t_{end}$ do
4:     Select $\delta t$ (according to Eq. (3.22)) if step-size control is used)
5:     Set boundary values for $u$ and $v$
6:     Compute $F^{(n)}$ and $G^{(n)}$ according to Eq. (3.19)
7:     Compute the right-hand side of the pressure equation Eq. (3.20)
8:     Set $it := 0$
9:     while $it < it_{max}$ and $\| r^{it} \| > eps$ (resp., $\| r^{it} \| > eps \| p^o \|$ ) do
10:        perform an iterative methods for system of linear equations (SOR cycle according
           to Eq. (3.29))
11:        Compute the residual norm Eq. (3.30) for the pressure equation $\| r^{it} \|$
12:        $it := it + 1$
13:    end while
14:    Compute $u^{(n+1)}$ and $v^{(n+1)}$ according to Eq. (3.18)
15:    $t := t + \delta t$
16:    $n := n + 1$
17: end while
```

3.2 Iterative Methods

Iterative solution methods work by applying a series of operations to an approximate solution to the linear system (i.e. $Ax = b$), with the error in the approximate solution being reduced by each application of the operations. Semantically, each application of the operations is an iteration, and the scheme iterate towards a solution. Iterative methods may require less memory and be faster than direct methods (i.e. Gaussian elimination, LU decomposition), and handle special structures (such as sparsity) in a simple way. Iterative methods mostly consists of

1. Stationary or simple methods (or classical iterative methods)

e.g. Jacobi, Gauss-Seidel, Successive Over-relaxation (SOR), and Symmetric Successive Over-relaxation (SSOR)

2. Krylov subspace methods e.g. Conjugate Gradient (CG), Bi-conjugate Gradient (BiCG), Generalized Minimal Residual (GMRES)
3. Multi-grid schemes

The stationary methods finds a splitting $A = M - K$ with non-singular M and iterates $x^{(k+1)} = M^{-1}(Kx^{(k)} + b) = Rx^{(k)} + c$. The iteration $x^{(k+1)} = Rx^{(k)} + c$ converges to the solution $x = A^{-1}b$ if and only if the spectral radius $\rho(R) < 1$, the spectral radius of an n by n matrix G is defined by

$$\rho(G) = \max\{|\lambda| : \lambda \in \lambda(G)\},$$

3.2.1 The Jacobi Method

It is the simplest iterative scheme and defined for matrices that have non-zero diagonal elements. For general n system $Ax = b$,

```
$for\ i = 1 : n$
    $x_i^{(k+1)} = \left( b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right) \Big/ a_{ii}$        (3.23)
$end$
```

Where, $x_j^{(k)}$ is the j^{th} unknown in x during the k^{th} iteration, $i = 1, 2, ..., n$ and $k = 0, 1, 2, ...;$
$x_i^{(0)}$ is the initial guess for the i^{th} unknown in x,
a_{ij} is the coefficient of A in the i^{th} row and j^{th} column,
b_{ij} is the i^{th} value in b.
Here Jacobi iteration does not use the most recently available information

when computing $x^{(k+1)}$. Thus it is highly suitable for parallel computing. It converges with a constant factor every $O(n^2)$.

In matrix term, splitting is done as $M = D$ and $K = L + U$ and

$$x_i^{(k+1)} = D^{-1}((L+U)x^{(k)} + b) \tag{3.24}$$

Where,

$x^{(k)}$ is the k^{th} iterative solution to x, $k = 0, 1, 2, ...;$

$x^{(0)}$ is the initial guess at x,

D is the diagonal of A,

L is the strictly lower triangular portion of A,

U is the strictly upper triangular portion of A,

b is the right-side vector.

3.2.2 The Gauss-Seidel Method

It is defined for matrices that have non-zero diagonal elements. For general n system $Ax = b$,

```
for $i = 1 : n$
    $x_i^{(k+1)} = \left( b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right) \Big/ a_{ii}$    (3.25)
end
```

Here iteration use the most recently available information when computing $x^{(k+1)}$. So it is faster than the Jacobi method. It is not suitable for parallel computing. In matrix term, splitting is done as $M = D - L$ and $K = U$ and

$$x^{(k+1)} = (D - L)^{-1}(Ux^{(k)} + b) \tag{3.26}$$

3.2.3 The Successive Over-relaxation Method

It is the variant of Gauss-Seidel Method. The Gauss-Seidel Method is slow if the spectral radius of $M^{-1}K$ is close to unity. With $\omega \in \mathbb{R}$ a modification of the Gauss-Seidel step gives the Successive Over-relaxation method.

```
for $i = 1 : n$
    $x_i^{(k+1)} = (1-\omega)x_i^{(k)} + \omega\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}\right) \Big/ a_{ii}$    (3.27)
end
```

In matrix terms, splitting is done as $M_\omega = D + \omega L$ and $K_\omega = (1-\omega)D - \omega U$ and the SOR step is given by

$$x^{(k+1)} = (D + \omega L)^{-1}(((1-\omega)D - \omega U)x^{(k)} + \omega b) \tag{3.28}$$

Here the parameter ω decides the convergence rate, if $\omega = 1$ then it is the Gauss-Seidel method. if $\omega > 1$ then the convergence rate is faster than the Gauss-Seidel method and called Over-relaxation method. Similarly if $\omega < 1$ then the convergence rate is slower and called under-relaxation method.

3.2.4 Red-Black Gauss-Seidel Method

The Red-Black Gauss-Seidel method can be considered as a compromise between Jacobi and Gauss Seidel. It is like a red-black ordering in a checkerboard pattern. In this method, red-black coloring scheme is applied on the staggered grid. First update the red cells from k to $k + 1$, which only depends on black cells at n. Similarly then update black cells from k to $k+1$, which depends on the red cells at $n+1$. Thus this method is highly suitable for parallel computation with the most recently available information when computing $x^{(k+1)}$.

3.3 Implementation of Iterative Methods on GPU

For viscous, incompressible laminar flow described in previous section, the discrete Poisson pressure equation takes following form by applying Successive Over-relaxation (SOR) methods as described in previous sections,

```
$for\ it = 1 : it_{max}$
    $for\ i = 1 : i_{max}$
        $for\ j = 1 : j_{max}$
            $p_{i,j}^{it+1} = (1-\omega)p_{i,j}^{it} + \frac{\omega}{\left(\frac{\epsilon_i^E+\epsilon_i^W}{(\delta x)^2} + \frac{\epsilon_j^S+\epsilon_j^N}{(\delta y)^2}\right)} \left(\frac{\epsilon_i^E p_{i+1,j}^{it} + \epsilon_i^W p_{i-1,j}^{it+1}}{(\delta x)^2} + \frac{\epsilon_j^S p_{i,j-1}^{it+1} + \epsilon_j^N p_{i,j+1}^{it}}{(\delta y)^2} - rhs_{i,j}\right)$
        $end$
    $end$
$end$                                                              (3.29)
```

$$r_{i,j}^{it} := \frac{\epsilon_i^E(p_{i+1,j}^{it} - p_{i,j}^{it}) - \epsilon_i^W(p_{i,j}^{it} - p_{i-1,j}^{it})}{(\delta x)^2} + \frac{\epsilon_j^N(p_{i,j+1}^{it} - p_{i,j}^{it}) - \epsilon_j^S(p_{i,j}^{it} - p_{i,j-1}^{it})}{(\delta y)^2} - rhs_{i,j}$$
$$i = 1, ..., i_{max}, \qquad j = 1, ..., j_{max} \tag{3.30}$$

The iteration is terminated either once a maximal number of steps it_{max} has been taken or when the norm of the residual has fallen below an absolute tolerance eps or a relative tolerance $eps \parallel p^0 \parallel$. Above algorithm become simple when the boundary condition is updated by the nearby pressure value to the boundary. As Gauss-Seidel method is not suitable for parallel computation, it is replaced by Jacobi and Red-Black SOR methods and shown in algorithm 2 and 3 respectively.

Algorithm 2 :- Algorithm for Jacobi iteration method

```
1: for $it = 1$ to $it_{max}$ do
2:     Update boundary condition.
3:     for $i = 1$ to $i_{max}$ do
4:         for $j = 1$ to $j_{max}$ do
5:             $temp_{i,j} = \frac{1}{\left(\frac{1}{(\delta x)^2}+\frac{1}{(\delta y)^2}\right)}\left(\frac{p_{i+1,j}+p_{i-1,j}}{(\delta x)^2}+\frac{p_{i,j+1}+p_{i,j-1}}{(\delta y)^2}-rhs_{i,j}\right)$
6:         end for
7:     end for
8:     $p_{i,j} = temp_{i,j}$
9:     Compute residue $r^{it}_{i,j}$ as per Eq. (3.30)
10: end for
```

Algorithm 3 :- Algorithm for Red-Black Successive Over-relaxation iteration method

```
1: for $it = 1$ to $it_{max}$ do
2:     Update boundary condition.
3:     for $i = 1$ to $i_{max}$ do
4:         for $j = 2 - (i\%2)$ to $j_{max}$ do
5:             $p_{i,j} = (1-\omega)p_{i,j} + \frac{\omega}{\left(\frac{1}{(\delta x)^2}+\frac{1}{(\delta y)^2}\right)}\left(\frac{p_{i+1,j}+p_{i-1,j}}{(\delta x)^2}+\frac{p_{i,j+1}+p_{i,j-1}}{(\delta y)^2}-rhs_{i,j}\right)$
6:             $j = j + 2$
7:         end for
8:     end for
9:     for $i = 1$ to $i_{max}$ do
10:        for $j = 1 + (i\%2)$ to $j_{max}$ do
11:            $p_{i,j} = (1-\omega)p_{i,j} + \frac{\omega}{\left(\frac{1}{(\delta x)^2}+\frac{1}{(\delta y)^2}\right)}\left(\frac{p_{i+1,j}+p_{i-1,j}}{(\delta x)^2}+\frac{p_{i,j+1}+p_{i,j-1}}{(\delta y)^2}-rhs_{i,j}\right)$
12:            $j = j + 2$
13:        end for
14:    end for
15:    Compute residue $r^{it}_{i,j}$ as per Eq. (3.30)
16: end for
```

3.4 Results and Analysis

Table - 3.1 shows the speed-up factor of Red-Black Successive Over-relaxation method over equivalent CPU code. Average speed-up factor of 21x is achieved for Red-Black SOR method with Over-relaxation rate (ω)= 1.7 in double precision on NVIDIA GeForce GTX280 on desktop computer with intel i7-920 processor.

Table 3.1: Speed-up results for Red-Black Successive Over-relaxation method for double precision on NVIDIA GeForce GTX 280

Numbers of Cells in x & y direction	Memory (MB)	Time (s) per iteration			Speed-up
		SOR CPU	Red-Black SOR CPU	Red-Black SOR GPU	
2000	122.0	0.25	0.27	0.01	27
3000	274.7	0.61	0.59	0.0275	21.45
4000	488.3	1.09	1.05	0.0475	22.11
5000	762.9	1.69	1.53	0.07	21.86

Table - 3.2 shows the speed-up factor of Jacobi method and Red-Black Gauss-Seidel method over equivalent CPU code in double precision. The speed-up performance of Red-Black Gauss-Seidel (GS) iteration method is less than that of Jacobi method. But the convergence rate of Red-Black Gauss-Seidel iteration method is higher than Jacobi iteration method. If we use Successive Over-relaxation method with Red-Black technique, then the convergence rate of Red-Black Successive Over-relaxation method is much higher than that of Jacobi method. Thus, Red-Black Successive Over-relaxation iteration method is more preferable for parallel computation on GPU.

Table 3.2: Speed-up results for Jacobi and Red-Black Gauss-Seidel methods for double precision on NVIDIA GeForce GTX 280

Numbers of Cells in x & y direction	Time (s) per iteration			Speed-up	
	Gauss-Seidel CPU	Jacobi CPU	Red-Black GS CPU	Jacobi	Red-Black GS
2000	0.28	0.38	0.25	38	25
3000	0.64	0.85	0.61	42.5	30.5
4000	1.06	1.58	1.09	43.05	27.17
5000	1.47	2.30	1.58	38.33	22.57

Chapter 4

Unstructured cell-centered AUSM$^+$-UP based Finite Volume Solver

The 3-dimensional simulation of internal flow field of a turbine start-up system of space launch vehicle is required for designer to select the material and propellant, calculate hardware thickness, arriving at appropriate geometry for optimum design and predicting the turbine performance. A computational fluid dynamics solver has been developed mainly for prediction and simulation of flow and thermal characteristics of the turbine startup motor to meet the thermal and mass flow requirements. Heat transfer was predicted using appropriate engineering correlations [5]. This CFD tool is based on unstructured cell-centered AUSM$^+$-UP based finite volume solver. The major requirement is to accelerate this unstructured grid based CFD tool. This chapter discusses the analysis of the code, implementation of computational intensive part to many-core GPU as well as multi-core CPU and various reordering schemes for improving coalesced memory access.

4.1 AUSM$^+$-UP-based finite volume solver

For this finite volume solver, the domain was divided into hexahedral control volumes with unstructured grid. The basic conservation equation is

$$\frac{\partial U}{\partial t} + \nabla \cdot F = S \tag{4.1}$$

where U = Vector of conservative variables,

F = Inviscid Flux vector,

S = Source term.

Integrating this equation,

$$\int \left(\frac{\partial U}{\partial t} + \nabla \cdot F \right) d\Omega = \int S d\Omega \tag{4.2}$$

$$\int \frac{\partial U}{\partial t} d\Omega + \int \nabla \cdot F d\Omega = \int S d\Omega \tag{4.3}$$

Applying Green's theorem

$$\int_\Omega \frac{\partial U}{\partial t} d\Omega + \int_\Gamma F \cdot n d\Gamma = \int_\Omega S d\Omega \tag{4.4}$$

where Γ and Ω are respectively the surface area and volume of the cell. The above equation can be written as

$$V_i \frac{\partial U_i}{\partial t} + \sum_{faces} F \cdot ds = S_i V_i \tag{4.5}$$

Here V_i is the cell volume and ds is the area of elemental sides.

For the simple explicit scheme the time stepping is done using Runge-Kutta method as below,

$$
\begin{aligned}
U_i^{(0)} &= U_i^{(n)} \\
U_i^{(1)} &= U_i^{(0)} - \alpha_1 \frac{\triangle t_i}{V_i}\left(R_i^{(0)}\right) \\
U_i^{(2)} &= U_i^{(0)} - \alpha_2 \frac{\triangle t_i}{V_i}\left(R_i^{(1)}\right) \\
U_i^{(3)} &= U_i^{(0)} - \alpha_3 \frac{\triangle t_i}{V_i}\left(R_i^{(2)}\right) \\
U_i^{(4)} &= U_i^{(0)} - \frac{\triangle t_i}{6V_i}\left(R_i^{(0)} + 2R_i^{(1)} + 2R_i^{(2)} + R_i^{(3)}\right) \\
U_i^{(n+1)} &= U_i^{(4)} \qquad (4.6)
\end{aligned}
$$

where the superscripts n and $n+1$ indicate the current and the next time steps. The values of coefficients in Runge-Kutta integration procedure are $\alpha_1 = 0.5$, $\alpha_2 = 0.5$ and $\alpha_3 = 1.0$. This method is only conditionally stable, as it is an explicit method. Local time stepping was employed for accelerating convergence. Thus, each control volume can march with its own maximum allowable time step specified by the explicit stability criteria given by

$$\triangle t_i \leq \frac{\triangle l_i}{q_i + c} \qquad (4.7)$$

where q_i is the magnitude of fluid velocity of i^{th} cell, given by

$$q_i = \sqrt{u_i^2 + v_i^2 + w_i^2}$$

and $c = \sqrt{\gamma RT}$, the sound velocity and $\triangle l_i$ is the characteristic dimension of the hexahedral element. Now the time step of the explicit solution procedure is given by,

$$\triangle t_i = CFL \times \frac{\triangle l_i}{q_i + c} \qquad (4.8)$$

where CFL is the Courant-Friedrichs-Lewy number.

In upwind schemes, the discretization of the equations on a mesh is performed according to the direction of propagation of information on that mesh, thereby incorporating the physical phenomena into the discretization schemes. In all Advection Upstream Splitting Methods (AUSM schemes), the inviscid flux is explicitly split into two parts i.e. convective and pressure terms by considering convective and acoustic waves as two physically distinct processes [6].

$$F = F^{(c)} + P \tag{4.9}$$

where $F^{(c)} = Ma \begin{bmatrix} \rho \\ \rho u \\ \rho h_t \end{bmatrix}$ and $P = \begin{bmatrix} 0 \\ p \\ 0 \end{bmatrix}$

Here the convective flux $F^{(c)}$ is expressed in terms of the convective speed M and the passive scalar quantities indicated in the brackets. The pressure flux P contains nothing but the pressure. Numerical flux $f_{i+1/2}$ can be expressed as the sum of the numerical convective flux $f^{(c)}_{i+1/2}$ and the numerical pressure flux $p_{i+1/2}$, at the interface $i + 1/2$ straddling the i^{th} and the $i + 1^{th}$ cells.

$$f_{i+1/2} = f^{(c)}_{i+1/2} + p_{i+1/2} \tag{4.10}$$

where one can further write $f^{(c)}_{i+1/2} = m_{i+1/2} \cdot \Phi_{i+1/2}$ and $P_{i+1/2} = \begin{bmatrix} 0 \\ p_{i+1/2} \\ 0 \end{bmatrix}$.

In AUSM^{+}-UP scheme, the interface fluxes are calculated based on the sign of the interface mass flow rate and pressure (evaluated using polynomial Fit) [6].

4.2 Implementation on Graphics Processing Unit

The computational intensive portion of this CFD solver consists of a loop which repeatedly computes the time derivatives of the conserved variables, given by Eq. (4.5). The classical Runge-Kutta (R-K) method is applied to integrate the non-linear coupled partial differential equation. Algorithm-4 shows various steps in the implementation of this scheme. The iteration loop is divided in to various functions as described in Algorithm-4. These functions are called as follows.

1. Time-step function :- Compute time interval δt_{min}
2. Bound-cond function :- Evaluate boundary condition for each cells at boundary
3. Fou-Inviscid function :- Compute inviscid flux (i.e density, momentum in x, y and z direction and momentum energy) using AUSM^{+}-UP technique.
4. Com-param-1 :- Compute the first step of Runge-Kutta method using constant α_1
5. Com-param-2 :- Compute the second step of Runge-Kutta method using constant α_2
6. Com-param-3 :- Compute the third step of Runge-Kutta method using constant α_3
7. Com-param-4 :- Compute the fourth step of Runge-Kutta method
8. Comp-diff :- Update the old conserved variable with newly computed conserved variable through R-K method

Algorithm 4 :- Algorithm for Cell-centered AUSM$^+$-UP based CFD Solver

```
1: Read initial value of physical parameters from file flow.in
2: Read data of hexahedral mesh from files grid.dat, bc.in and compute geometrical data for unstructured mesh
3: Initialize the physical parameters or update from file restart.in, if iteration is restarted.
4: for $iter = 1$ to no. of iterations do
5:     Compute time interval $\delta t_{min}$
6:     for STEP $k = 1$ to 4 do
7:         Apply boundary conditions.
8:         Compute inviscid parameters
9:         Compute the slope for given time interval, as per classical Runge-Kutta (R-K) method for each inviscid parameters
10:        Compute physical parameters $(u, v, w, E, T, R_o, P)$ with the help of slope variables for given time interval of classical Runge-Kutta method
11:    end for
12:    Update variable TIME with incremental time interval and R-K coefficients
13: end for
14: Write computed physical parameters $(u, v, w, P, T, R_o, Mach\ no.)$ to file flowc.out
```

Among these functions, a function Fou-Inviscid is computational intensive part of the complete solver due to random memory access. Algorithm for computing *inviscid* parameters is shown in algorithm-5. The code is suitable for parallel computation and ported to GPU. The parallel computation is based on per-element basis, with one thread per element. Various kernels are grouped to above functions depending up on the functionality.

The Fou-Inviscid function computes the inviscid flux parameters for each side of hexahedral element as well as for each common side of neighbour elements. In parallel computation each thread computes the inviscid flux parameters for each element. To avoid race condition in parallel implementation the computed flux for each neighbour element is stored in different vectors. At the end of inviscid flux computation, the inviscid flux of neighbour elements is aggregated.

Fig. 4.1 shows the relative computation time of each individual function for sequential and parallel program. Two data set of unstructured grid is used for testing the parallelized code. Grid data-1 consists of 26,901 nodes, 24,000 hexahedral elements and 5,600 ghosts cells (elements). Similarly grid data-2 consists of 564,417 nodes, 540,000 hexahedral elements and 48,200 ghosts cells. So, as element size increases, the arithmetic computation hides the memory latency and provides good acceleration.

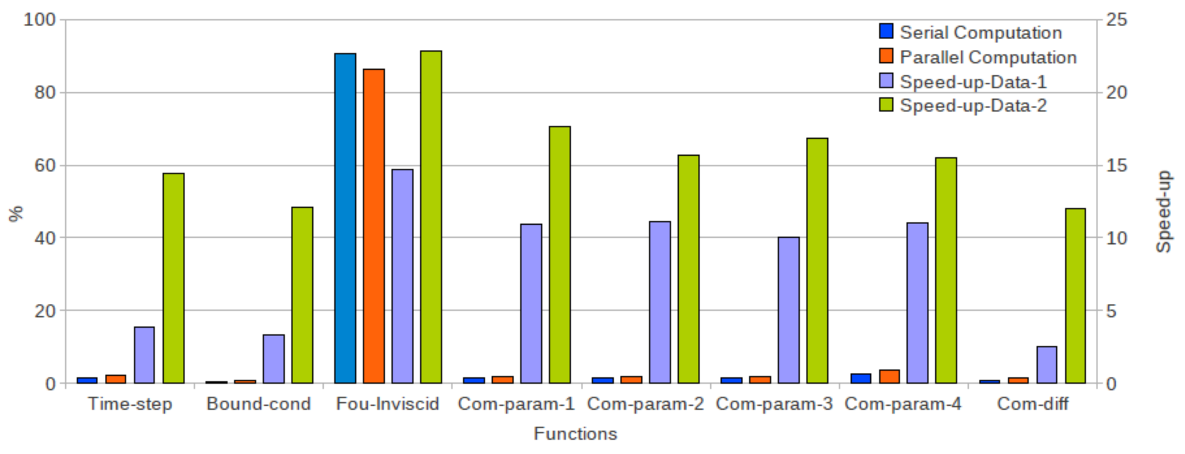


Figure 4.1: Computational time profile of an unstructured grid based CFD Solver

Apart from above technique, various performance optimization strategies are implemented in GPU computation. Optimization of memory usage to achieve maximum memory throughput and maximization of parallel execution to achieve maximum utilization are most important performance optimization strategies.

The first step in maximizing overall memory throughput for the application is to minimize data transfers with low bandwidth. This means minimizing data transfers between the host and the device, as well as minimizing data transfers between global memory and the device by maximizing use of on-chip memory: shared memory, constant cache and texture cache for NVIDIA GeForce GTX280 GPU. Shared memory is used to avoid redundant global memory access amongst threads within a block, but the GPU does not automatically make use of shared memory. So, information must be made available which specifies which global memory access can be shared by multiple threads within a block. This information is not known priori due to data dependent memory access pattern of unstructured grid based solver. With the per-element/thread based connectivity data structure, the use of shared memory is not applicable to this case. The off-chip constant memory space provides cache facility. Thus an optimized code is developed to take advantage of constant cache. The second step is the coalesced global memory access. Coalesced global memory access reduces the memory transactions and increases the instruction throughput. A better global memory access can be achieved by re-numbering the elements in unstructured grid such that the elements nearby in space remain nearby in memory. Next section discusses few renumbering strategies developed for better coalesced global memory access.

To maximize utilization the application should be structured in a way that it exposes as much parallelism as possible and efficiently maps this parallelism to the various components of the system to keep them busy most of the time. Maximization of utilization at microprocessor level is done by hiding the memory latency by arithmetic instruction. In our case the function Fou-Inviscid is not arithmetic bound. Secondly occupancy management is important to maximize parallel utilization. The number of blocks and warps residing on each multiprocessor for a given kernel call depends on the execution configuration of the call, the memory resources of the multiprocessor, and the resource requirements of the kernel. With the help of occupancy calculator, execution configuration of each kernel call is optimized. The number of registers used by a kernel can have a significant impact on the number of resident warps and occupancy. In our case by limiting maximum registers to 64 for Fou-Inviscid kernel, it's occupancy is increased to 25 %, which results 10 % improvement in speed-up. The Table - 4.1 shows the comparison of solver's computation time for both data sets.

Table 4.1: Speed-up results for finite volume CFD solver for double precision on NVIDIA GeForce GTX280

Number of elements	Time (s) per element per iteration			Speed-up	
	Single core CPU	NVIDIA GPU		Un-optimized	Optimized
		Un-optimized	Optimized		
24,000	5.21E-6	3.76E-7	3.37E-7	13.86	15.47
540,000	5.42E-6	2.26E-7	1.98E-7	24.00	27.36

Algorithm 5 :- Finding Inviscid Flux Parameters using AUSM$^+$-UP Scheme

```
1: for $i = 1$ to no. of elements do
2:     for $j = 1$ to 5 do
3:         $inviscid[i][j] = 0$
4:     end for
5: end for
6: for $i = 1$ to no. of elements do
7:     for $j = 1$ to 6 ( no. of surface of hexahedral element) do
8:         $N = j^{th}$ neighbor of $i^{th}$ element
9:         if $N > i$ then
10:            if $N$ is not a ghost cell then
11:                Compute left & right state variables and $m_{1/2}$, $p_{1/2}$
12:                if $m_{1/2} > 0$ then
13:                    Compute $inviscid[i][1]$ to $inviscid[i][5]$ and $inviscid[N][1]$ to
                       $inviscid[N][5]$ using left side variable.
14:                else
15:                    Compute $inviscid[i][1]$ to $inviscid[i][5]$ and and $inviscid[N][1]$ to
                       $inviscid[N][5]$ using right side variable.
16:                end if
17:            else
18:                Compute $inviscid[i][2]$ to $inviscid[i][4]$ and $inviscid[N][2]$ to $inviscid[N][4]$
                   using $m_{1/2}$
19:            end if
20:        end if
21:    end for
22: end for
```

4.3 Renumbering the elements in Unstructured Grid

A detailed study of algorithm suggest that memory access to the ghost cells, which are neighbour cells of the exterior or interior boundary cells of the complete control volume, is expensive and their computation is not useful further. Thus first modification is removal of the inviscid flux computation for ghost cells and their memory access. This changed new computation is called modified computation. Secondly cache-misses in computing system are directly linked to the bandwidth of the equivalent matrix system (or graph). Point renumbering to reduce bandwidths has been an important theme for many years in traditional finite element applications [7]. Each element is represented by node of a graph and the common surface between two elements is represented by an edge of the graph. With this arrangement and application of few algorithm from graph theory gives good re-numbering of the elements in unstructured grids. Fig. 4.2a show the structure of the adjacency matrix represented by the graph of elements of unstructured grid. From fig. 4.2a, it is clear that the ghost cells are far part in memory from other cells and we have to apply the renumbering scheme such that the adjacency matrix of elements of unstructured grid have minimum bandwidth. The techniques discussed here do not require spatial location of points in the unstructured grid. Following techniques require only connectivity table of the grid i.e elements and list of their neighbour's elements.

Bandwidth of an N by N symmetric positive definite matrix A, with entries $a_{i,j}$, is given by

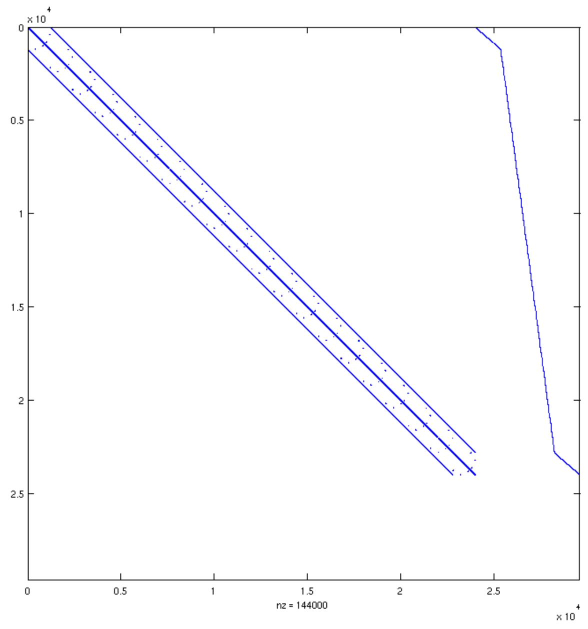
$$\beta(A) = \max\{|i-j| \mid a_{i,j} \neq 0\} \tag{4.11}$$

and number $\beta_i(A)$ is called the i-th bandwidth of A. The envelope of A, denoted by $Env(A)$ [8], is defined by

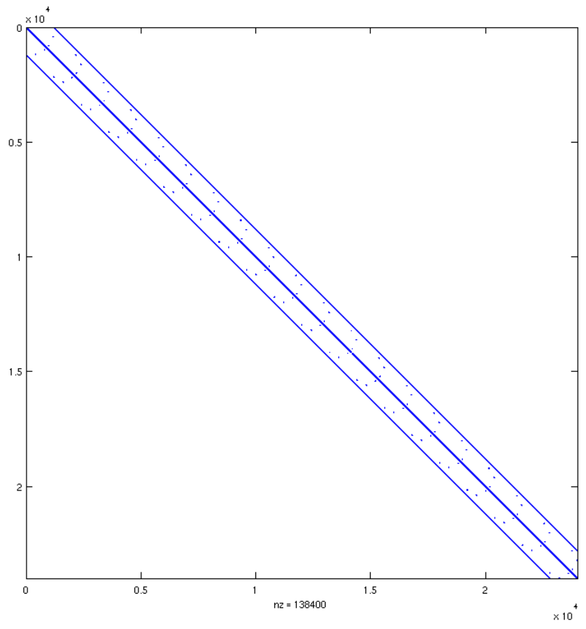
$$Env(A) = \{\{i, j\} \mid 0 < (i - j) \leq \beta_i(A),\ i \geq j\} \tag{4.12}$$

The quantity $|Env(A)|$ is called the profile or envelope size of A, and is given by

$$P_r(A) = |Env(A)| = \sum_{i=1}^{N} \beta_i(A) \tag{4.13}$$



(a) Matrix with all elements



(b) Matrix with all elements excluding ghost elements

Figure 4.2: Structure of adjacency matrix formed by the elements with and without ghost elements of unstructured grid data - 1

All of the bandwidth minimization strategies are heuristic by nature. Initially an reverse Cuthill-McKee (RCM) reordering is applied as bandwidth minimization strategy. Reverse Cuthill-McKee algorithm is most widely used profile reduction ordering algorithm and variant of the Cuthill-McKee algorithm. Cuthill-McKee algorithm is designed to reduce the bandwidth of a sparse symmetric matrix via a local minimization of the

β_i's. An ordering obtained by reversing the Cuthill-McKee ordering often turns out to be much superior to the original ordering in terms of profile reduction, although the bandwidth remains unchanged. Thus, obtained reverse ordering is called reverse Cuthill-McKee ordering (RCM) [10]. In this scheme at each stage, the node with the smallest number of surrounding unrenumbered nodes is added to the renumbering table. The profile resulting from this ordering is quite sensitive to the choice of the starting node. A good choice for a starting node will be to choose a peripheral node, that is one whose eccentricity equals the diameter of the graph as this will generate a narrow level structure where the difference in number for a node and its neighbors is minimal. Peripheral nodes are not easy to find quickly. Therefore, heuristics were devised to find "pseudo-peripheral" nodes, i.e. nodes whose eccentricities are close to the diameter of the graph. Gibbs, Poole and Stockmeyer [11] and George and Liu [10] have provided various heuristics algorithm to find the pseudo-peripheral node of given graph. The detailed algorithm of reverse Cuthill-Mckee is shown as algorithm - 6. Here the degree of a node in a graph is the number of nodes adjacent to it.

Fig. 4.2b and 4.3a shows the structure of adjacency matrix before and after applying RCM algorithm respectively. The reverse Cuthill McKee algorithm is implemented with Boost C++ library package [9]. This renumbering gives good bandwidth and profile of adjacency matrix of elements of unstructured grids.

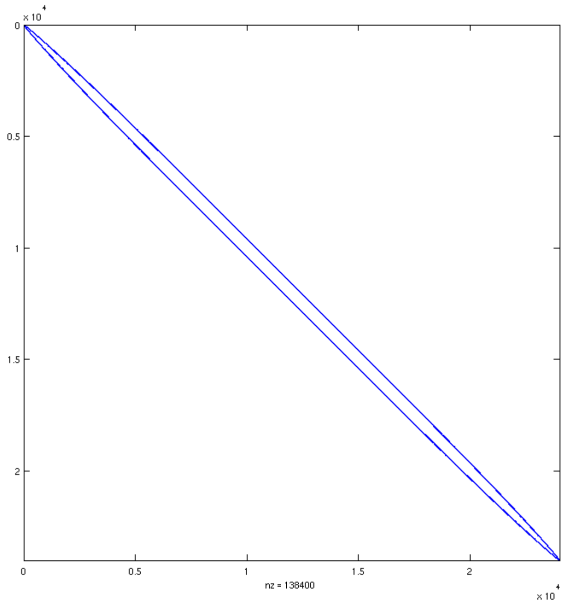
The second strategy is re-ordering the adjacency matrix by fill-reducing orderings, suited for Cholesky-based direct factorization algorithms. This METIS algorithm is based on a multilevel nested dissection algorithm and produce low fill orderings for a wide variety of matrices [12]. Fig. 4.3b

Algorithm 6 :- Reverse Cuthill-McKee Algorithm

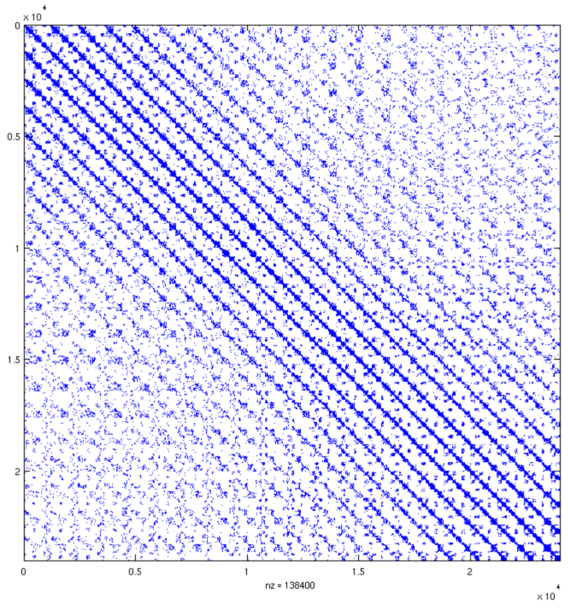
```
1: Prepare a Graph G(n) from the adjacency list of elements in unstructured grid
2: Prepare an empty queue Q and an empty result array R
3: Find a pseudo-peripheral starting node, that hasn't previously been inserted in the
   result array R. Let us name it P (for Parent)
4: Add P in the first free position of R
5: Add to the queue all the nodes adjacent with P in the increasing order of their degree
6: Extract the first node from the queue and examine it. Let us name it C (for Child)
7: If C hasn't previously been inserted in R, add it in the first free position and add to
   Q all the neighbour of C that are not in R in the increasing order of their degree
8: If Q is not empty repeat from line 6
9: If there are unexplored nodes (the graph is not connected) repeat from line 3
10: Reverse the order of the elements in R. Element R[i] is swapped with element R[n+1-i]
11: The result array will be interpreted like this: R[L] = i means that the new label of
    node i (the one that had the initial label of i) will be L
```

shows the structure of adjacency matrix after applying METIS matrix re-ordering algorithm. The renumbering by METIS algorithm gives poor bandwidth and profile of adjacency matrix of elements of unstructured grids.

The third strategy is based on matrix reordering technique used in partitioning and global routing subproblems in Very Large Scale Integrated (VLSI) design. This technique is used to reorder a binary m by n matrix Q, such that in the reordered matrix, the ones are clustered along the geometric diagonal as tightly as possible [13]. The matrix reordering technique is based on the calculation of the second largest eigenvalue and the corresponding eigenvector of a related matrix and called a spectral partitioning. An implementation of spectral partitioning involves the construction of incidence matrix of hyper-edge representing node and edge structure, cal-



(a) Matrix after RCM Algorithm



(b) Matrix after METIS Algorithm

Figure 4.3: Structure of adjacency matrix after application of RCM and METIS algorithm for unstructured grid data - 1

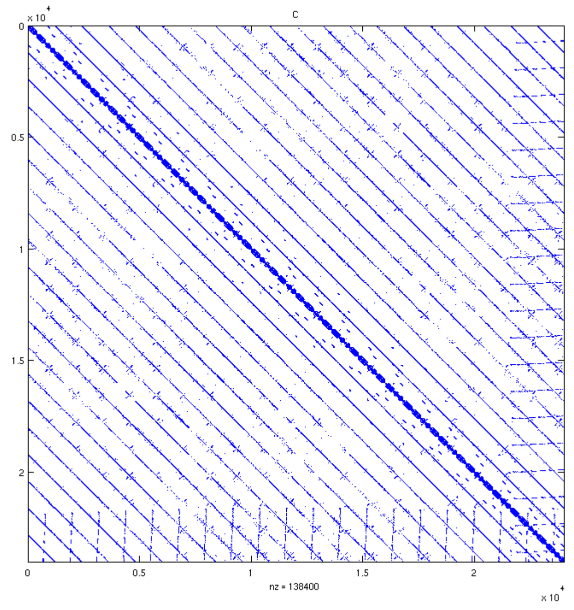
culation of second largest eigenvalue and eigenvector, and re-ordering the incidence matrix from the eigenvector. The hyper-graph is constructed by different strategies. Various hyper-graph schemes is listed below.

1. Spectral partitioning - 1: Hyper-edges are constructed from neighbour cells of a given elements excluding ghosts cells.

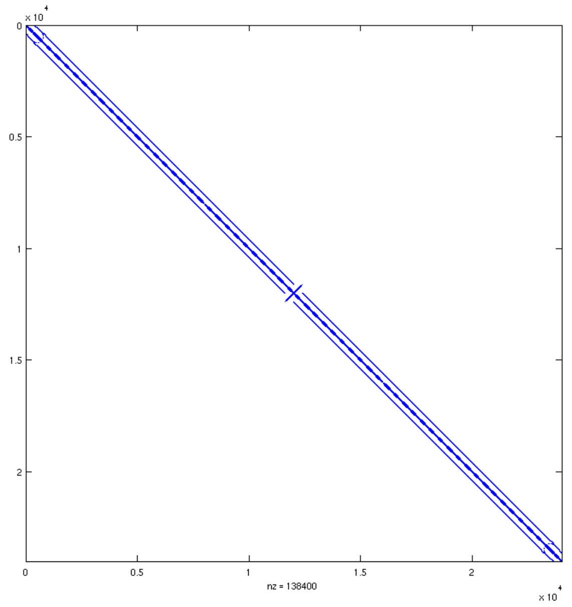
2. Spectral partitioning - 2: Hyper-edges are constructed from neighbour cells of given elements along with itself and excluding ghosts cells.

3. Spectral partitioning - 3: Hyper-edges are constructed from neighbour cells of same side and excluding ghosts cells.

4. Spectral partitioning - 4: Hyper-edges are constructed between given cell and their neighbour cell with excluding ghosts cells.



(a) Matrix after spectral partitioning - 1

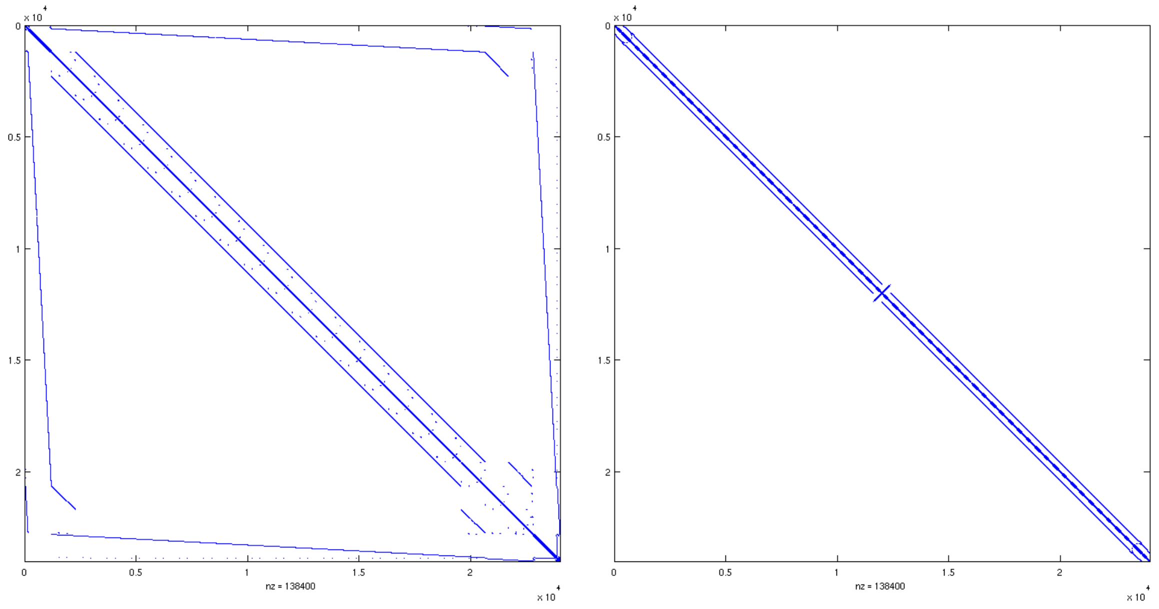


(b) Matrix after spectral partitioning - 2

Figure 4.4: Structure of adjacency matrix after application of spectral partitioning scheme 1 and 2 for unstructured grid data - 1

Hyper-graph generated by spectral partitioning scheme number 4 is similar of the hyper-graph represented by the scheme number 2, as edges of one elements with their neighbour's elements represent one hyper-edge. Fig. 4.4a, 4.4b, 4.5a and 4.5b shows the structure of adjacency matrix after reordering by spectral partitioning algorithm.

Table 4.2 shows the comparison of bandwidth and profile obtained by various renumbering schemes. The relative value of bandwidth and profile is with respect to original ordering. An reverse Cuthill-McKee reordering algorithm gives best reduction in bandwidth and profile for adjacency matrix represented by the graph of unstructured grid. While reordering by METIS algorithm gives the worst reduction in bandwidth and profile for adjacency matrix. For re-ordering by spectral partitioning algorithm, the bandwidth and profile depends upon the choice of hyper-edges of hyper-graph.



(a) Matrix after spectral partitioning - 3 (b) Matrix after spectral partitioning - 4

Figure 4.5: Structure of adjacency matrix after application of spectral partitioning scheme 3 and 4 for unstructured grid data - 1

Table 4.2: Bandwidth and profile of adjacency matrix of unstructured grid after renumbering techniques

Ordering methods	Grid data-1 (24,000 element)			Grid data-2 (540,000 elements)		
	Band-width	Relative Bandwidth	Relative Profile	Band-width	Relative Bandwidth	Relative Profile
Original	1200	1.00	1.00	10800	1.00	1.00
Reverse Cuthill-Mckee	420	0.35	0.31	2550	0.24	0.21
Spectral Partitioning-1	23701	19.75	5.50	533259	49.38	13.00
Spectral Partitioning-2	796	0.66	0.35	4996	0.46	0.23
Spectral Partitioning-3	22796	18.00	3.12	529100	49.00	3.77
Spectral Partitioning-4	796	0.66	0.35	4996	0.46	0.23
METIS	23685	19.74	5.71	536090	49.64	9.20

4.4 Implementation on Multi-core CPU

This computational fluid dynamics solver is implemented on multi-core Intel i7 processor using OpenMP parallel programming model. OpenMP is an Application Programming Interface (API) extension to the C, C++ and Fortran languages to write parallel programs for shared memory machines [30]. OpenMP supports the fork-join programming model. Under this approach, the program starts as a single thread of execution, just like a sequential program. The thread that executes this code is referred to as the initial thread. Whenever an OpenMP parallel construct is encountered by a thread while it is executing the program, it creates a team of threads (this is called the fork), becomes the master of the team, and collaborates with the other members of the team to execute the code dynamically enclosed by the construct. At the end of the construct, only the original thread, or master of the team, continues; all others terminate (this is called the join). Each portion of code enclosed by a parallel construct is called a parallel region.

OpenMP is based on the shared-memory model; hence, by default, data is shared among the threads and is visible to all of them. Also data can be shared or private to each thread. Shared data is accessible by all, while private data is accessed only by the thread that owns it. In OpenMP data transfer is transparent to the programmer and implicit synchronization takes place. OpenMP has small set of explicit synchronization. Synchronizing, or coordinating the actions of, threads is sometimes necessary in order to ensure the proper ordering of their accesses to shared data and to prevent data corruption. OpenMP provides directives or constructs, library functions, and environment variables to create and control the execution of parallel programs. Few of constructs and library functions are mainly parallel construct, work-sharing constructs (i.e. loop, sections, single construct), data-sharing, no-wait, and schedule clauses as well as synchronizing constructs (i.e. barrier, critical, atomic, locks and master construct). This constructs make parallel program much powerful and easy to implement. The independent computational intensive part of the code is parallelized using the loop work-sharing constructs.

Fig. 4.6 and 4.7 shows the computational performance measurement of the solver for data-2 on Intel i7-920 processor with GNU GCC compiler 4.1.2, running one to nine threads. An average speed-up of 3.10x and 3.3x is achieved for double and single precision respectively for both unstructured grid data. The maximum speed-up is achieved for 7-thread parallel computation on Intel's i7 processor.

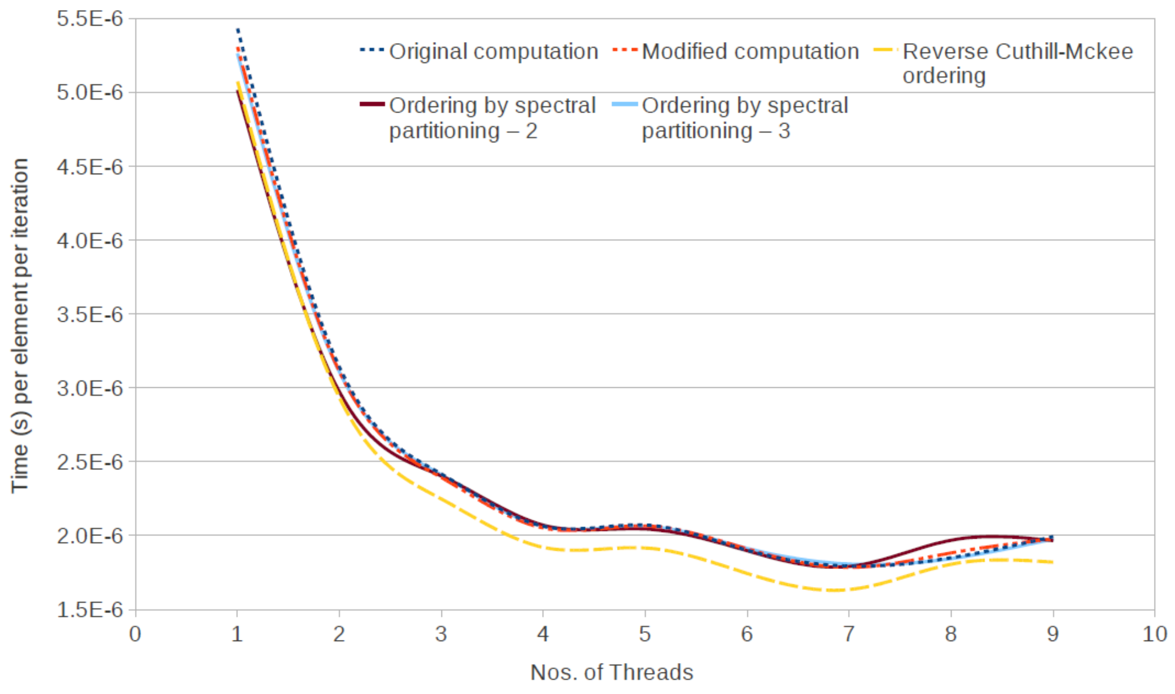


Figure 4.6: Multi-threaded computational performance of unstructured grid based CFD Solver on i7-920 processor - 1

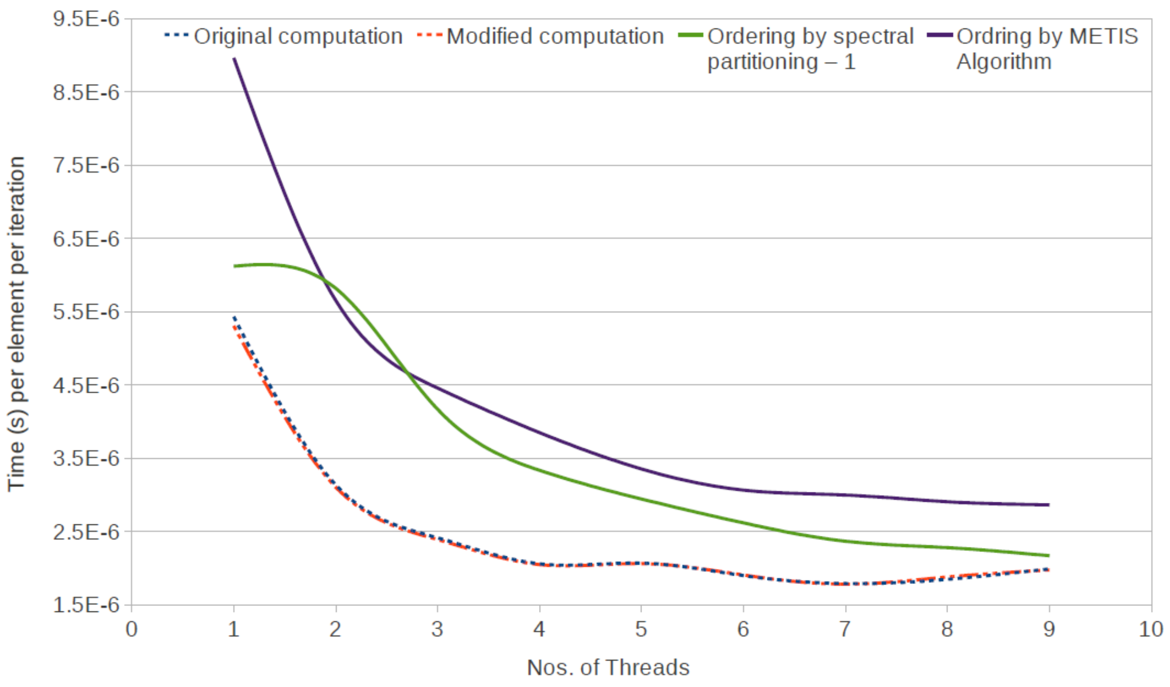


Figure 4.7: Multi-threaded computational performance of unstructured grid based CFD Solver on i7-920 processor - 2

4.5 Results and Analysis

Table - 4.3 shows the speed-up for unstructured grid based finite volume CFD solver for double precision over equivalent openMP code running on one core after reordering the elements in unstructured grid. An reverse Cuthill-McKee reordering algorithm gives lower running time than original ordering with modified computation for sequential code on i7-920 processor. Unstructured grid data-1 gives least running time on GPU, but unstructured grid data-2 gives running time very close to that of original ordering with modified computation. While other re-ordering gives poor running times than that of original ordering and inconsistent as well as unpredictable with their bandwidth and profile properties. This confirms that better global memory access is achieved by ordering the elements such that nearby elements in space remain nearby in memory and have least bandwidth and profile. NVIDIA's latest Fermi compute architecture having on-chip cache and off-chip cache will offer good performance to unstructured grid based CFD solver with reverse Cuthill-McKee ordering. Secondly it is observed that modified computation, i.e. without ghosts cell's inviscid flux computations, gives better performance than that of original computation with ghosts cell's inviscid flux computations. This is due to the reduced global memory access.

Table 4.3: Speed-up results of CFD solver after renumbering schemes for double precision on NVIDIA GeForce GTX280

Ordering methods	Grid data-1			Grid data-2		
	Time (s) per element per iteration		Speed-up	Time (s) per element per iteration		Speed-up
	CPU	GPU		CPU	GPU	
Original Computation	5.35E-6	3.36E-7	15.91	5.43E-6	1.98E-7	27.40
Modified Computation	5.34E-6	3.31E-7	16.09	5.31E-6	1.96E-7	27.02
Reverse Cuthill-McKee	5.03E-6	2.83E-7	17.80	5.07E-6	2.00E-7	25.31
Spectral-Partitioning-1	4.94E-6	2.87E-7	17.18	6.12E-6	2.06E-7	29.71
Spectral-Partitioning-2	5.45E-6	3.67E-7	14.84	5.02E-6	2.75E-7	18.26
Spectral-Partitioning-3	5.24E-6	3.34E-7	15.71	5.47E-6	1.98E-7	27.69
METIS	6.17E-6	4.13E-7	14.94	8.96E-6	3.56E-7	25.18

Table - 4.4 shows the speed-up for unstructured grid based finite volume CFD solver for single precision over equivalent openMP code running on one core. The single precision gives better speed-up than double precision at the cost of error in results due to the reduced precision during computation. Secondly NVIDIA GeForce GTX280 have one double precision processing unit per stream multiprocessor. NVIDIA's latest Fermi compute architecture having double precision processing unit per stream processor, will provide better performance than current GPU hardware.

Table 4.4: Speed-up results of CFD solver after renumbering schemes for single precision on NVIDIA GeForce GTX280

Ordering methods	Grid data-1			Grid data-2		
	Time (s) per element per iteration		Speed-up	Time (s) per element per iteration		Speed-up
	CPU	GPU		CPU	GPU	
Original Computation	4.25E-6	1.16E-7	36.72	1.04E-5	1.71E-7	60.61
Modified Computation	4.08E-6	1.08E-7	37.68	1.03E-5	1.66E-7	62.24
Reverse Cuthill-McKee	3.55E-6	1.03E-7	34.64	9.02E-6	1.66E-7	54.26
Spectral-Partitioning-1	3.54E-6	1.57E-7	22.58	1.17E-5	3.18E-7	36.76
Spectral-Partitioning-2	4.15E-6	1.23E-7	33.72	9.08E-6	2.38E-7	38.19
Spectral-Partitioning-3	4.21E-6	1.09E-7	38.43	1.03E-5	1.63E-7	63.01
METIS	4.66E-6	1.91E-7	24.4	1.83E-5	6.06E-7	30.12

The parallel code for unstructured grid data - 2 with 540,00 elements and double precision shows an average speed-up of 27x in comparison to the OpenMP code running on one core and 9.0x in comparison to the OpenMP code running on seven cores. Similarly the parallel code for unstructured grid data - 1 with 24,000 elements with double precision shows an average speed-up of 16x in comparison to the OpenMP code running on one core and 5.3x in comparison to the OpenMP code running on seven cores.

Part II

Electromagnetic Computation on GPU Hardware

Chapter 5

Finite Difference Time Domain (FDTD) Method

The finite-difference time-domain (FDTD) formulation of electromagnetic field problem is a convenient tool for solving scattering problems. In 1966 Yee [14] proposed a technique to solve Maxwell's curl equations using the FDTD which was later developed by Teflove. The equations are solved in a leapfrog manner: the electric field is solved at a given instant in time, then the magnetic field is solved at the next instant in time, and the process is repeated over and over again. However, FDTD runs too slow for some simulations to be practical, even when carried out on supercomputers. The development of dedicated hardware to accelerate FDTD computation has been investigated. This chapter discusses the current trend of FDTD computations on GPU and a simulation of propagation of Gaussian pulse in 2-dimensions on GPU.

5.1 History of FDTD Computation on Dedicated Hardware for Acceleration

In earlier years of introduction of programmable graphics card, Krakiwsky et al. have demonstrated approximately 10x speed-up for 2-D FDTD computation on NVIDIA GeForce FX 5900 Ultra GPU [19, 20]. Valcarce et al. have describe the simulation of radio coverage prediction for two dimensions using FDTD with Convolutional Perfectly Matched Layer (CPML) with CUDA [21]. Balevic et al. have demonstrated accelerated simulations of light scattering based on FDTD with transverse magnetic (TM) mode in 2-D with CUDA on GPU [22]. Some of the other numerical solution with FDTD on GPU are referenced at Baron et al. 2005; Humphrey et al. 2006; Adams et al. 2007 [23, 24, 25].

5.2 Finite Difference Time Domain (FDTD) Technique

The finite-difference technique is based upon approximations which permit replacing differential equations by finite difference equations. These finite difference approximations are algebraic in form and they relate the value of the dependent variable at a point in the solution region to the values at some neighboring points [15] [16]. The basic steps involved are:

1. Dividing the solution region into grid of nodes.
2. Approximating the given differential equation by finite difference equivalent that relates the dependent variable at a point in the solution region to its values at the neighboring points

3. Solving the difference equations subject to the prescribed boundary conditions and initial conditions

The course of action taken in the above three steps is decided by the nature of the problem being solved, the solution region and the boundary conditions.

5.2.1 Yee's FDTD algorithm

Maxwell's equations for an isotropic medium can be written as [14] [15]

$$\nabla \times \vec{E} = -\mu \frac{\partial \vec{H}}{\partial t} \tag{5.1}$$

$$\nabla \times \vec{H} = \sigma \vec{E} + \epsilon \frac{\partial \vec{E}}{\partial t} \tag{5.2}$$

$$\frac{\partial E_x}{\partial t} = \frac{1}{\epsilon}\left(\frac{\partial H_z}{\partial y} - \frac{\partial H_y}{\partial z} - \sigma E_x\right) \tag{5.3}$$

$$\frac{\partial E_y}{\partial t} = \frac{1}{\epsilon}\left(\frac{\partial H_x}{\partial z} - \frac{\partial H_z}{\partial x} - \sigma E_y\right) \tag{5.4}$$

$$\frac{\partial E_z}{\partial t} = \frac{1}{\epsilon}\left(\frac{\partial H_y}{\partial x} - \frac{\partial H_x}{\partial y} - \sigma E_z\right) \tag{5.5}$$

$$\frac{\partial H_x}{\partial t} = \frac{1}{\mu}\left(\frac{\partial E_y}{\partial z} - \frac{\partial E_z}{\partial y}\right) \tag{5.6}$$

$$\frac{\partial H_y}{\partial t} = \frac{1}{\mu}\left(\frac{\partial E_z}{\partial x} - \frac{\partial E_x}{\partial z}\right) \tag{5.7}$$

$$\frac{\partial H_z}{\partial t} = \frac{1}{\mu}\left(\frac{\partial E_x}{\partial y} - \frac{\partial E_y}{\partial x}\right) \tag{5.8}$$

Where,

$\vec{E}$ = Electric Field Intensity,

$\vec{H}$ = Magnetic Filed Intensity,

E_x, E_y, E_z are Electric Field Intensity in x, y, z direction,

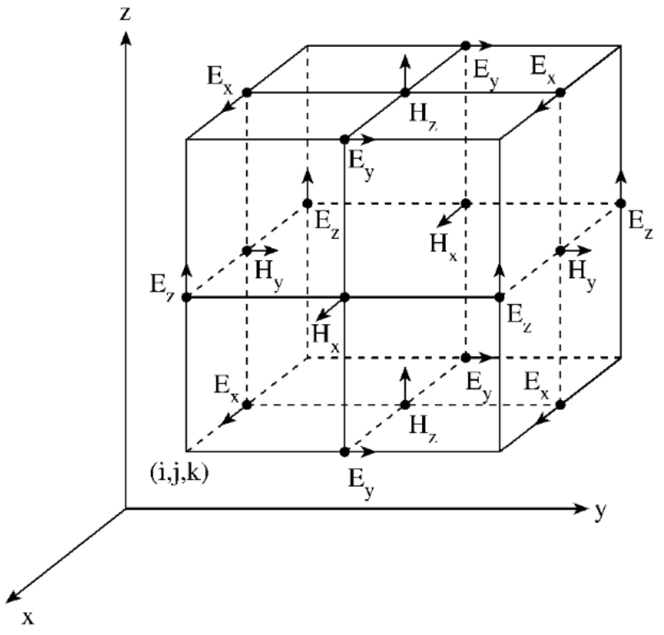


Figure 5.1: Positions of the field components in a unit cell of the Yee's lattice

H_x, H_y, H_z are Magnetic Field Intensity in x, y, z direction,

σ = Electrical Conductivity, μ = Permeability, ϵ = Permittivity

Following Yee's notation, we define a grid point in the solution region as

$$(i,j,k) = (i\triangle x,\ j\triangle y,\ k\triangle z) \tag{5.9}$$

and any function of space and time as

$$\phi^n(i,j,k) = \phi(i\delta,\ j\delta,\ k\delta,\ n\triangle t) \tag{5.10}$$

Where $\delta = \triangle x = \triangle y = \triangle z$ is the space increment, $\triangle t$ is the time increment, and $i,\ j,\ k,\ n$ are integers. Applying central difference approximation for space and time derivatives that are second order accurate to Eq. (5.9) and Eq. (5.10) we get,

$$\frac{\partial \phi^n(i,j,k)}{\partial x} = \frac{\phi^n(i+\frac{1}{2},j,k) - \phi^n(i-\frac{1}{2},j,k)}{\delta} + O(\delta^2) \tag{5.11}$$

$$\frac{\partial \phi^n(i,j,k)}{\partial t} = \frac{\phi^{n+\frac{1}{2}}(i,j,k) - \phi^{n-\frac{1}{2}}(i,j,k)}{\triangle t} + O(\triangle t^2) \tag{5.12}$$

By applying Eq. (5.11) and Eq. (5.12) to all time and space derivatives, Yee positions the components of $\vec{E}$ and $\vec{H}$ about a unit cell of the lattice as shown in the Fig. 5.1. $\vec{E}$ and $\vec{H}$ field are evaluated at alternate half time steps, such that all field components are calculated in each time step $\triangle t$.

Cell Size

The choice of cell size is critical in applying FDTD technique. It must be small enough to permit accurate results at the highest frequency of interest and yet be large enough to be implemented on computer. The cell size is directly affected by the materials present. The fundamental constraint is that the cell size must be much less than the smallest wavelength for which accurate results are desired. Size of each cell should 0.1λ or less at the highest frequency (shortest wavelength) of interest [16].

Accuracy and Stability

For the accuracy of the computed results, the spatial increment δ must be small compared to the wavelength. To ensure the stability of the finite difference scheme the increment $\triangle t$ must satisfy the following stability criteria [16].

$$u_{\max}\triangle t \leq \left[\frac{1}{\triangle x^2} + \frac{1}{\triangle y^2} + \frac{1}{\triangle z^2}\right]^{-1/2} \tag{5.13}$$

where $u_{\max}$ is the maximum wave phase velocity within the model. An electromagnetic wave propagating in free space cannot go faster than speed of light. To propagate a distance of one cell it requires minimum time of $\triangle t = \triangle x/c_o$ where $\triangle x$ is the cell size and c_o is the velocity of light. When we proceed to n dimension [17]

$$\triangle t = \frac{\triangle x}{\sqrt{n} c_o} \tag{5.14}$$

approximated here as,

$$\triangle t = \frac{\triangle x}{2 c_o} \tag{5.15}$$

Absorbing Boundary Condition

Absorbing boundary conditions (ABC) are necessary to keep outgoing E and H fields from being reflected back into the problem space. The basic assumption of FDTD is that when calculating E field we need to know the surrounding H field. But at the edge of the problem space we will not have the value to one side however we know that there are no sources outside the problem space. Therefore the fields at the edge must be propagating outwards. A more elegant ABC is the Perfectly Matched Layer (PML). The implementation of FDTD algorithm and simulations will be discussed in the next section.

5.3 Simulation of Gaussian Pulse Propagation by FDTD Method

This section deals with formulation and simulation of Gaussian pulse propagation in two dimension. The medium specific parameters are included and the absorbing boundary condition is incorporated using the perfectly matched layer (PML) proposed initially by Berenger [18].

5.3.1 Two dimensional FDTD theory and formulation

In doing two-dimensional simulation one of the two groups of three fields each, namely Transverse magnetic (TM) mode composed of E_z, H_x, and H_y or Transverse Electric mode (TE), composed of E_x, E_y and H_z is selected [14] [17]. TM mode is used for this work. Eqs. 5.1 and 5.2 are reduced with the above assumptions in the form,

$$\frac{\partial E_z}{\partial t} = \frac{1}{\epsilon}\left(\frac{\partial H_y}{\partial x} - \frac{\partial H_x}{\partial y} - \sigma E_z\right), \frac{\partial H_x}{\partial t} = -\frac{1}{\mu}\frac{\partial E_z}{\partial y}, \frac{\partial H_y}{\partial t} = \frac{1}{\mu}\frac{\partial E_z}{\partial x} \quad (5.16)$$

Taking the central difference approximations for both the temporal and spatial derivate,

$$\begin{aligned} {E_z}^{n+1}(i,j) &= \left(1 - \frac{\sigma(i,j)\triangle t}{\epsilon(i,j)}\right){E_z}^{n}(i,j) \\ &\quad + \frac{\triangle t}{\epsilon(i,j)\delta}[{H_y}^{n+1/2}(i+1/2,j) - {H_y}^{n+1/2}(i-1/2,j) \\ &\quad + {H_x}^{n+1/2}(i,j-1/2) - {H_x}^{n+1/2}(i,j+1/2)] \end{aligned} \quad (5.17)$$

$${H_x}^{n+1/2}(i,j+1/2) = {H_x}^{n-1/2}(i,j+1/2) + \frac{\triangle t}{\mu(i,j+1/2)\delta}[{E_z}^{n}(i,j) - {E_z}^{n}(i,j+1)] \quad (5.18)$$

$${H_y}^{n+1/2}(i+1/2,j) = {H_y}^{n-1/2}(i+1/2,j) + \frac{\triangle t}{\mu(i+1/2,j)\delta}[{E_z}^{n}(i+1,j) - {E_z}^{n}(i,j)] \quad (5.19)$$

The formulation of the above equation assumes that the E and H fields are interleaved in both space and time. This is the fundamental paradigm of the finite-difference time-domain (FDTD) method. Eq. (5.17), (5.18), (5.19) are very similar but ϵ and μ differ by several orders of magnitude. Thus applying change of variables [17]

$$\tilde{E} = \sqrt{\frac{\epsilon_o}{\mu_o}}E \quad (5.20)$$

The normalized '$\tilde{E}$ ' field unit is called Gaussian units. The 'Ez ' field should be understood as in the normalized units in the subsequent sections.

Substituting Eq. (5.20) to (5.17), (5.18), (5.19),

$$\begin{aligned}\tilde{E}_z^{n+1}(i,j) &= \frac{(1-q(i,j))}{(1+q(i,j))}\tilde{E}_z^{n}(i,j) \\ &+\frac{q_i(i,j)}{(1+q(i,j)}[{H_y}^{n+1/2}(i+1/2,j) - {H_y}^{n+1/2}(i-1/2,j) \\ &+{H_x}^{n+1/2}(i,j-1/2) - {H_x}^{n+1/2}(i,j+1/2)] \end{aligned} \tag{5.21}$$

$${H_x}^{n+1/2}(i,j+1/2) = {H_x}^{n-1/2}(i,j+1/2)+q_2(i,j+1/2)[\tilde{E}_z^{n}(i,j)-\tilde{E}_z^{n}(i,j+1)] \tag{5.22}$$

$${H_y}^{n+1/2}(i+1/2,j) = {H_y}^{n-1/2}(i+1/2,j)+q_2(i+1/2,j)[\tilde{E}_z^{n}(i+1,j)-\tilde{E}_z^{n}(i,j)] \tag{5.23}$$

where,

$$q(i,j) = \frac{\sigma(i,j)\triangle t}{2\epsilon_o\epsilon_r(i,j)}, \qquad q_1(i,j) = \frac{\triangle t c_o}{\delta\epsilon_r(i,j)}, \qquad q_2(i,j) = \frac{\triangle t c_o}{\delta\mu_r(i,j)} \tag{5.24}$$

5.3.2 Simulation of 2-D Gaussian Pulse Propagation in unbound medium without Absorbing Boundary Conditions (ABC)

The simulation of two dimensional unbounded medium is similar to free space simulation without any constraints. The simulation is done with Gaussian source at the center of the problem space. The permittivity and permeability of the region is taken as one. A program is implemented in C language with the Eq. (5.22), (5.23) and (5.24). Fig. 5.2 is the result for 30 time steps and the pulse has not reached the boundary yet. After 100 time steps the pulse is seen to have reached the boundary and reflected as seen in Fig. 5.3. The contour in Fig. 5.4 is neither concentric nor symmetric about the center due to reflections. The same code is implemented on GPU and good speed-up achieved. Table 5.1 shows the speed-up for various size of the cell over equivalent serial code.

Table 5.1: Speed results of FDTD algorithm without absorbing boundary condition on NVIDIA GeForce GTX280

Numbers of Cells in x & y direction	Nos. of Steps	GPU Time (s)	CPU Time (s)	Speed-up
64	1000	0.0525	0.18	3.43
160	1000	0.08	1.13	14.13
320	1000	0.1425	5.3825	37.77
480	1000	0.2825	14.605	51.70
640	1000	0.43	24.43	56.81
64	5000	0.2525	0.805	3.19
160	5000	0.37	5.6425	15.25
320	5000	0.7275	24.825	34.12
480	5000	1.4175	60.7825	42.88
640	5000	2.18	126.8125	58.17

5.3.3 Formulation in two dimensions with ABCs

The formulation in the further sections has been done using Maxwell's equation with flux density D. The overall procedure which has been adopted in previous sections remain the same and for the E and D values, normalized form of the equations are used for the change of variables. The detailed derivation can be found in [17]. The advantage of this formulation is when one wants to simulate frequency dependent media. The set of two dimensional TM mode equations implemented in C code are:

$$D_z[i][j] = D_z[i][j] + 0.5(H_y[i][j] - H_y[i-1][j] - H_x[i][j] + H_x[i][j-1]) \quad (5.25)$$

$$E_z[i][j] = q_{1z}[i][j](D_z[i][j] - sum[i][j]) \quad (5.26)$$

$$sum[i][j] = sum[i][j] + q_{1z}[i][j]E_z[i][j] \quad (5.27)$$

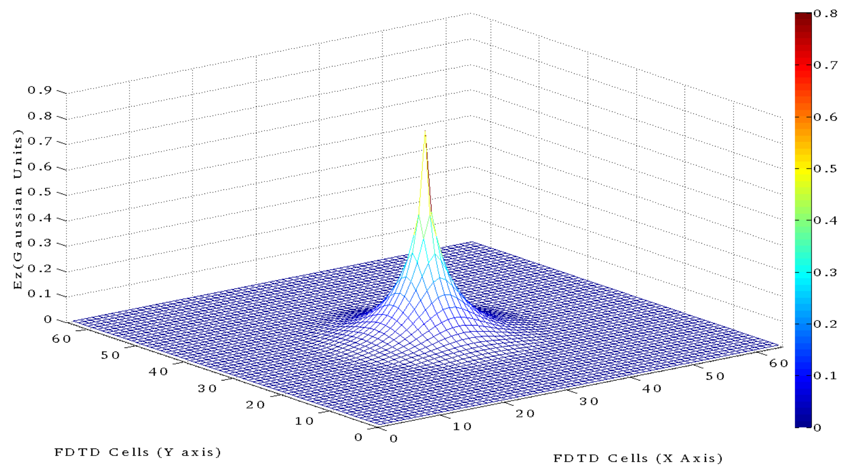


Figure 5.2: E_z field after 30 time steps with Gaussian pulse at center

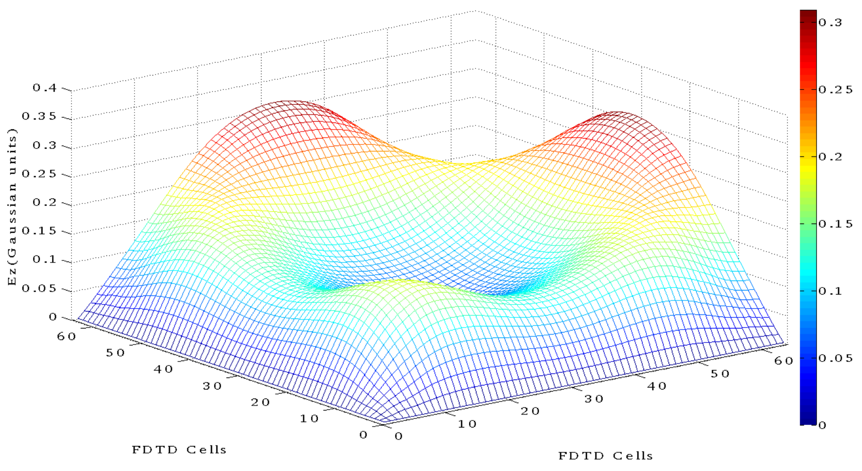


Figure 5.3: E_z field after 100 time steps with Gaussian pulse at center without ABC's

$$H_x[i][j] = H_x[i][j] + 0.5(E_z[i][j] - E_z[i][j+1]) \tag{5.28}$$

$$H_y[i][j] = H_y[i][j] + 0.5(E_z[i+1][j] - E_z[i-1][j]) \tag{5.29}$$

where,

$$q_{1z}[i][j] = \frac{1}{\epsilon_r + \frac{(\sigma * \triangle t)}{\epsilon_o}} \qquad q_{2z}[i][j] = \frac{(\sigma * \triangle t)}{\epsilon_o} \tag{5.30}$$

Absorbing boundary conditions are needed to keep outgoing electric field E and magnetic field H from being reflected back into the problem space.

A basic difficulty encountered in applying the FDTD method to scattering problems is that the domain in which the field is to be computed

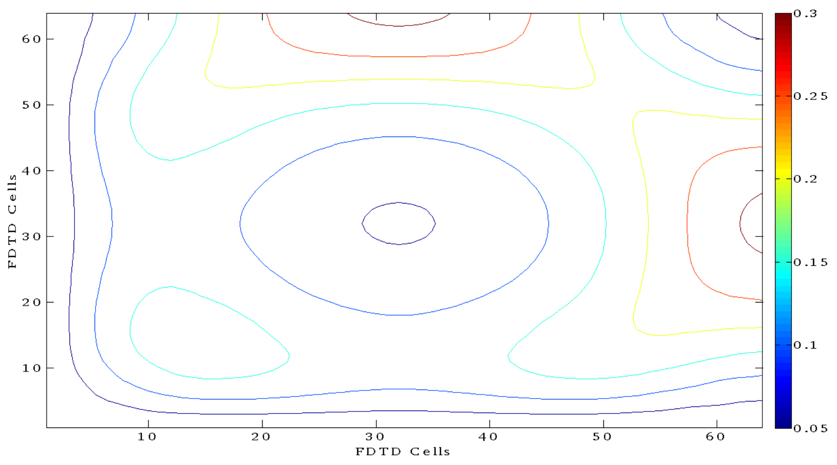


Figure 5.4: Contour of E_z field after 100 time steps with Gaussian pulse at center without ABC's

is open or unbounded. Since no computer can store an unlimited amount of data, a finite difference scheme over the whole domain is impractical. We must limit the extent of our solution region or an artificial boundary must be enforced to create the numerical illusion of an infinite space. The solution region must be large enough to enclose the scatterer, and suitable boundary conditions on the artificial boundary must be used to simulate the extension of the solution region to infinity. Outer boundary conditions of this type is called Absorbing Boundary Conditions (ABC). The accuracy of the ABC dictates the accuracy of the FDTD method. For simplicity we have consider only Berenger's Perfectly Matched Layer (PML) type of ABC. The PML has been the most widely accepted and is set to revolutionize the FDTD method.

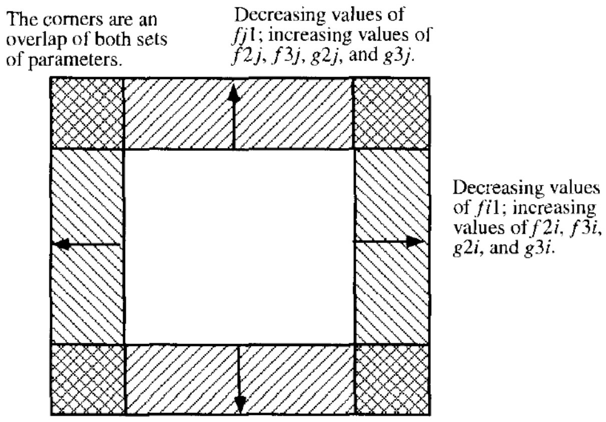
If a wave is propagating in medium-A and it strikes upon medium-B, the amount of reflection is dictated by the intrinsic impedances of the two media, which is given by

$$\Gamma = \frac{\eta_A - \eta_B}{\eta_A + \eta_B} \tag{5.31}$$

The impedance are determined by ϵ and μ of the two media:

$$\eta = \sqrt{\frac{\mu}{\epsilon}} \tag{5.32}$$

If any medium is lossy, the EM wave dies out. Hence for a certain thickness close to the boundary, the medium is modeled as lossy to avoid the reflections from being generated. This is accomplished by making ϵ and μ complex and the decay caused is represented by the imaginary terms. The detailed derivation can be found in [17, 18]. The formulation will be applicable to a certain number of cells in the boundary. Thus the thickness of PML layer can be specified in the program. Once the wave reaches the defined PML points (cells) it will get attenuated and will not be reflected back. Complete set of parameters associated with PML is defined in the developed code and by setting the values that are found empirically to be the most effective value which satisfy stability and variation are used [17]. The fig. 5.5 shows the parameters related to PML used in code.



The corners are an overlap of both sets of parameters.
Decreasing values of $fj1$; increasing values of $f2j$, $f3j$, $g2j$, and $g3j$.
Decreasing values of $fi1$; increasing values of $f2i$, $f3i$, $g2i$, and $g3i$.

Figure 5.5: Representation of parameters related to PML [17]

5.3.4 Two dimensional simulation results with PML

A code with perfectly matched layer from [17] has been implemented on GPU and compared with serial execution. The GPU has shown higher speed-up than simulation without absorbing boundary condition because memory latency has been hide with more arithmetic computation. Table 5.2 shows the speed-up of the FDTD simulation with PML. A comparison of fig. 5.6 and 5.7, shows how the reflections get eliminated with PML. The outgoing contour of fig. 5.7 is circular and only when the wave gets within eight points (PML) of the problem space, does the phase front depart from its circular nature.

Table 5.2: Speed-up results of FDTD algorithm with Perfectly Matched Layer type of ABC on NVIDIA GeForce GTX280

Numbers of Cells in x & y direction	Nos. of Steps	GPU Time (s)	CPU Time (s)	Speed-up
64	1000	0.1	0.34	3.4
160	1000	0.1325	2.28	17.21
320	1000	0.2725	15.7325	57.73
480	1000	0.5925	60.1625	101.54
64	5000	0.46	1.6925	3.68
160	5000	0.66	11.8575	17.97
320	5000	1.355	80.195	59.18
480	5000	2.97	277.7375	93.51

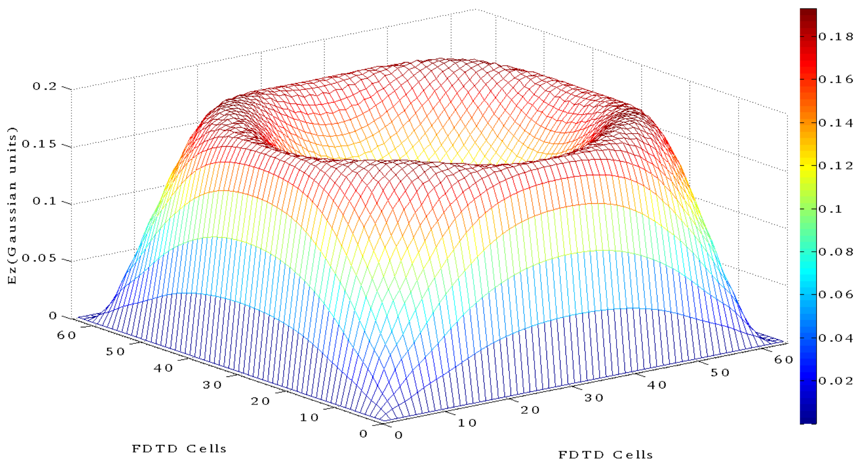


Figure 5.6: E_z field after 90 time steps with Gaussian pulse at center and 8 point PML

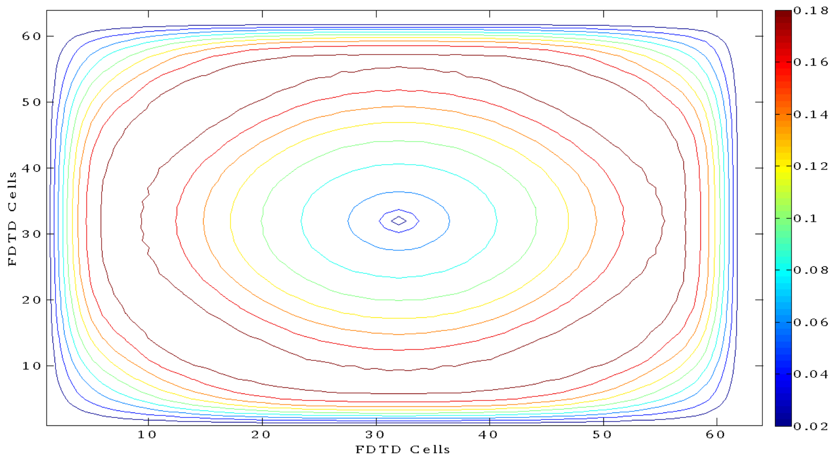


Figure 5.7: Contour of E_z field after 90 time steps with Gaussian pulse at center and 8 point PML

Part III

Eigenvalue Problem on GPU Hardware

Chapter 6

Computation of Eigenvalue and Eigenvector for Banded Sparse Matrix

A banded matrix is generated from the discretization of partial differential equation in 1-D, 2-D and 3-D of physical systems. The eigenvalue and eigenvector models the important parameter in physics and engineering. For example, the time-dependent Schrödinger equation in quantum mechanics is

$$H\psi_E = E\psi_E \tag{6.1}$$

where H, the Hamiltonian, is a second-order differential operator and ψ_E, the wave-function, is one of its eigenfunctions corresponding to the eigenvalue E, interpreted as its energy. The wave-functions associated with the bound states of an electron in a hydrogen atom can be seen as the eigenvectors of the hydrogen atom Hamiltonian as well as of the angular momentum operator. They are associated with eigenvalues interpreted as their energies and angular momentum [29]. The symmetric banded matrix have been generated from the discretization of partial differential equation of Schrödinger equation. The computation of eigenvalue and eigenvector involve the Inverse Iteration method, Lanczos method and bisection algorithm.

6.1 The Inverse Iteration Method

Inverse iteration method for selected eigenvalue and eigenvector is just the power method applied to $(A - \mu I)^{-1}$. For $(A - \mu I) \in \mathbb{R}^{n \times n}$, a nonsingular matrix, given a unit 2-norm $q^{(0)} \in \mathbb{C}^n$, following algorithm produces a eigenvalue nearest to the given μ shift [27]. Where,
$q^{(k)}$ is the k^{th} normalized eigenvector for $\lambda^{(k)}$ eigenvalue, $k = 0, 1, 2, ...;$
$r^{(k)}$ is the k^{th} residual for $\lambda^{(k)}$ eigenvalue, $k = 0, 1, 2, ...;$
c is a constant of order unity.

Algorithm 7 :- Inverse Iteration Algorithm

```
1: repeat
2:     Solve $(A - \mu I)z^{(k)} = q^{(k-1)}$
3:     $q^{(k)} = z^{(k)} / \| z^{(k)} \|_2$
4:     $\lambda^{(k)} = [q^{(k)}]^T A q^{(k)}$
5:     $r^{(k)} = (A - \mu I) q^{(k)}$
6: until $\| r^{(k)} \|_\infty \leq cu \| A \|_\infty$
```

Here the system of linear equations $(A - \mu I)z^{(k)} = q^{(k-1)}$ is solved by Conjugate Gradient (CG) algorithm. A special algorithm has been developed for multiplication of sparse banded matrix with vector and Conjugate Gradient method for banded sparse matrix [28]. This algorithm treats the diagonal, sub-diagonal and super-diagonal elements as vectors and computation involves only multiplication & addition of vectors.

6.2 The Lanczos Method

Lanczos method is used for partial tridiagonalizations of the given matrix A. Long before the tridiagonalization is complete, A's extremal eigenvalues tend to emerge. Thus Lanczos algorithm is particular useful to find a few of A's largest or smallest eigenvalues.

The Lanczos Algorithm

Given a symmetric $A \in \mathbb{R}^{n \times n}$ and $\omega \in \mathbb{R}^n$ having unit 2-norm, the following algorithm computes a k-by-k symmetric tridiagonal matrix T_k with the property that $\lambda(T_k) \subset \lambda(A)$. The diagonal and sub-diagonal elements of T_k are stored in $\alpha(1 : k)$ and $\beta(1 : k-1)$ respectively [27]. A few results

Algorithm 8 :- The Lanczos Algorithm

```
1: $v(1:n) = 0; \beta_0 = 1; k = 0;$
2: while $\beta_k \neq 0$ do
3:    if $k \neq 0$ then
4:       for $i = 1$ to n do
5:          $t = w_i;$
6:          $w_i = v_i/\beta_k;$
7:          $v_i = -\beta_k t$
8:       end for
9:    end if
10:   $v = v + A\omega;$      $k = k + 1;$
11:   $\alpha_k = w^T v;$      $v = v - \alpha_k w;$      $\beta_k = \parallel v \parallel_2$
12: end while
```

for speed-up has shown in Table 6.1. The computation has been carried out on NVIDIA GeForce GTX 280.

Table 6.1: Results of speed-up of Lanczos algorithm with double precision on NVIDIA GeForce GTX280 GPU.

Matrix Size	Nos. of Threads	GPU Time (s)	CPU Time (s)	Speed-Up
40,000 x 40,000	256	25.94	138.58	5.34
1,000,000 x 1,000,000	256	160.86	1982.03	12.32

6.3 Algorithm for eigenvalue of banded sparse matrix on GPU

NVIDIA's software development kit (SDK) contains a program for computation of all eigenvalues of tridiagonal symmetric matrix of arbitrary size using bisection algorithm with CUDA [26]. Our algorithm consists of converting banded matrix to symmetric tridiagonal using Lanczos algorithm, finding the eigenvalue of symmetric tridiagonal matrix using bisection algorithm and finding the accurate eigenvalue and eigenvector using inverse iteration method.

Algorithm 9 :- The Pseudo-Code for Computation of Eigenvalue and Eigenvector (GPU)

```
1: Get Matrix size, nos. of thread per block
2: Allocate memory dynamically for banded matrix
3: Allocate device memory and copy all variables from host to device
4: Set-up the execution configuration i.e parameters of Lanczos's kernel
5: Call kernel to compute Lanczos algorithm
6: Copy only diagonal and sub-diagonal elements to host and free device memory
7: Compute eigenvalue from bisection algorithm
8: Sort the eigenvalues in ascending orders
9: Allocate device memory and copy all elements of banded matrix to device
10: Set-up the execution configuration i.e parameters of Inverse iteration's kernel
11: for i = 1 to size of Matrix do
12:     Call kernels to compute inverse iteration algorithm for eigenvectors & eigenvalues.
13: end for
14: Free device and host memory
```

Chapter 7

Conclusions and Future works

7.1 Conclusions

During this work various scientific problems have been studied with emphasizing on numerical solution of partial differential equation. The partial differential equation can be solved in many ways depending on it's nature and boundary conditions. It is shown that substantial speed-up (≥ 20) has been achieved for GPU implementation in double as well as single precision over equivalent single core CPU implementation. An unstructured grid based finite volume solver have been implemented on many-core GPU and multi-core CPU using OpenMP programming model. It is shown that many-core graphics processing computing is much better than multi-core processor computing. Various optimization techniques, i.e GPU optimization and algorithmic strategies, have been employed for unstructured grid based finite volume solver and provided better speed-up performance. With NVIDIA's next generation Fermi compute architecture unstructured grid based scientific applications will provide better speed-up due to availability of cache at on & off chip level and double precision floating point unit per stream processors.

7.2 Future Works

We have implemented renumbering scheme for unstructured grid, which is based on the connectivity table of elements and their neighbour's elements. There are other renumbering schemes, which are based on the spatial and cell information of the unstructured grid. Implementation of those schemes may give better speed-up than our implementation. Secondly we have implemented computational fluid dynamics solver on single GPU, it will be useful to implement on multi-GPU configuration with OpenMP programming model to accelerate the finite volume solver on unstructured grid with very large elements.

References

[1] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. *GPU Computing*, Proceedings of the IEEE, 96(5), May 2008, pp. 879-899

[2] NVIDIA Corporation, *NVIDIA CUDA Compute Unified Device Architecture C Programming Guide*, Version 3.1.1, 2010

[3] Michael Grieble, Thomas Dornseifer and Tilman Neunhoeffer, *Numerical Simulation in Fluid Dynamics, A Practical Introduction*, Society for Industrial and Applied Mathematics, 1998

[4] Hirt, C., Nicholas, B., and Romero, N., 1975, *SOLA - A Numerical Solution Algorithm for Transient Fluid Flows*, Technical report LA-5852, Los Alamos, NM: Alamos National Lab.

[5] Praveen Nair, Jayachandran T., Balachandra Puranik, and Upendra V. Bhandarkar, *Simulation of Thermo-fluid Interactions in Cryogenic Stage Turbine Startup System Using AUSM$^+$-UP-based Higher-order Accurate Flow Solver*, Defence Science Journel, Vol. 59, No. 3, May 2009, pp. 215-229

[6] Meng-Sing Liou, *A sequel to AUSM, Part II: AUSM$^+$-UP for all speeds*, Journel of Computational Physics, Vol. 214, 2006, pp. 137-

170

[7] Löhner R., *A*pplied CFD Techniques: An Introduction Based on Finite Element Methods, 2nd Edition, John Wiley and Sons Ltd.

[8] G. Meurant, *Computer Solution of Large Linear Systems*, Studies in Mathematics and Its Applications, volume-28, Elsevier, 1999

[9] Boost C++ Library, *Cuthill-McKee ordering*, http://www.boost.org

[10] George, Alan and Joseph Liu, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice-Hall, 1981.

[11] Gibbs, N.E., Poole, W.G., Stockmeyer,P.K., *An algorithm for reducing the bandwidth and profile of a sparse matrix*, SIAM Journal on Numerical Analysis, vol-13, no 2, 1976, pp.236-250

[12] George Karypis, Kirk Schloegel and Vipin Kumar, *METIS - Serial Graph Partitioning and Fill-reducing Matrix Ordering*, http://glaros.dtc.umn.edu/gkhome/metis/metis/overview

[13] R. H. J. M. Otten, *Eigensolutions in top-down layout design*, IEEE Symposium on Circuits and Systems, 1982, pp. 1017-1020

[14] K. S. Yee, *Numerical Solution of Initial Boundary Value Problem Involving Maxwells Equation in Isotropic Media*, IEEE Transactions on Antennas and Propagation, vol. AP-17, Jan. 1966, pages 236-250

[15] M. N .O. Sadiku, *Numerical Techniques in Electromagnetics*, CRC - Press, Boka Raton, Third edition, 2001

[16] K. S. Kunz and J. L. Raymond, *The Finite Difference Time Domain Method for Electromagnetics*, CRC Press, 1993

[17] D. M. Sullivan, *Electromagnetic Simulation Using FDTD Method*, IEEE Press series on RF and Microwave Technology, New York, 2000

[18] J. P. Berenger, *A Perfectly Matched Layer for the absorption of Electromagnetic Waves*, Journal of Computational Physics, vol. 114, Oct. 1994, pp. 185-200

[19] Sean E. Krakiwsky, Laurence E. Turner and Michal M. Okoniewski, 2004a, *Graphics Processing Unit (GPU) Acceleration of Finite-Difference Time-Domain (FDTD) Algorithm*, Proceedings of the 2004 IEEE International Symposium on Circuits and Systems, pp. 265-268.

[20] Sean E. Krakiwsky, Laurence E. Turner and Michal M. Okoniewski, 2004b, *Acceleration of Finite-Difference Time-Domain (FDTD) using Graphics Processing Unit (GPU)*, IEEE MIT-S International Microwave Symposium Digest, vol. 2, pp. 1033-1036, Jun. 2004.

[21] Valcarce A., De La Roche, G. and Zhang, J., 2008 *A GPU approach to FDTD for radio coverage prediction*, Proceedings of the 11th IEEE International Conference on Communication Systems, pp. 1585-1590, Guangzhou, China,

[22] Balevic A., Rockstroh, L., Tausendfreund, A., Patzelt, S., Goch, G. and Simon, S., 2008b *Accelerating Simulations of Light Scattering based on Finite-Difference Time-Domain Method with General Purpose GPUs*, Proceedings of the 11th IEEE International Conference on Computational Science and Engineering, IEEE Computer Society, pp. 1585-1590, Washington, DC, USA

[23] Baron, G.S., Sarris, C.D., Fiume, E., 2005, *Fast and accurate time-domain simulations with commodity graphics hardware*, in Proceedings of the 2005 IEEE Antennas and Propagation Society International Symposium, 4A, pp. 193-196, IEEE Computer Society, Washington, DC, USA,

[24] Humphrey, J.R., Price, D.K., Durbano, J.P., Kelmelis, E.J. Martin, R.D., 2006, *High performance 2D and 3D FDTD solvers on GPUs*, in Proceedings of the 10th WSEAS International Conference on Applied Mathematics, pp. 547-550, World Scientific and Engineering Academy and Society (WSEAS), Dallas, TX, USA.

[25] Adams, S., Payne, J., Boppana, R., 2007, *Finite difference time domain (FDTD) simulations using graphics processors*, in Proceedings of the Department of Defense High Performance Computing Modernization Program Users Group Conference, pp. 334-338, IEEE Computer Society,Washington, DC, USA.

[26] Christian Lessig, *Eigenvalue Computation with CUDA*, NVIDIA Software Development Kit, October 2007.

[27] Golub H. G., Van Loan F. C., *Matrix Computations*, Third Edition, The Johns Hopkins University Press.

[28] Pradip Narendrakumar Panchal, *GPU based Multi Core Parallel Computing*, M. Tech. Seminar Report, November 2009

[29] http://en.wikipedia.org/wiki/Eigenvalue

[30] *The OpenMP API Specification for Parallel Programming*,http://openmp.org

Acknowledgements

I am deeply grateful to Prof. Sachin Patkar for his invaluable guidance and discussion on various aspect of computation. I thank Mr. Praveen Nair of Vikram Sarabhai Space Center, Thiruvananthapuram for providing unstructured grid data and algorithm for finite volume solver. I also thank Prof. S. V. Kulkarni for providing the computational intensive problem in electromagnetic field. I thank to Mr. Anirudh Maringanti and Dr. Gaurav Trivedi to help me out in understanding some of the computational concepts. I also thank Electrical Engineering Department for providing the computing environment for doing this project.

Lastly I thank my parents and family for their support and patience which helped me in fulfillment of my M. Tech project.