GPU-BASED ACCELERATION ON ACENET FOR FDTD METHOD OF ELECTROMAGNETIC FIELD ANALYSIS

by

Dachuan Sun

Submitted in partial fulfilment of the requirements
for the degree of Master of Applied Science

at

Dalhousie University
Halifax, Nova Scotia
November 2013

Dedication

This work is dedicated to my parents who gave me support and encouragement at each step of my life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Graphics Processing Unit (GPU) programming techniques have been applied to a range of scientific and engineering computations. In computational electromagnetics, uses of the GPU technique have dramatically increased since the release of NVIDIA's Compute Unified Device Architecture (CUDA), a powerful and simple-to-use programmer environment that renders GPU computing easy accessibility to developers not specialized in computer graphics.

The focus of recent research has been on problems concerning the Finite-Difference Time-Domain (FDTD) simulation of electromagnetic (EM) fields. Traditional FDTD methods sometimes run slowly due to large memory and CPU requirements for modeling electrically large structures. Acceleration methods such as parallel programming are then needed. FDTD algorithm is suitable for multi-thread parallel computation with GPU. For complex structures and procedures, high performance GPU calculation algorithms will be crucial.

In this work, we present the implementation of GPU programming for acceleration of computations for EM engineering problems. The speed-up is demonstrated through a few simulations with inexpensive GPUs and ACEnet, and the attainable efficiency is illustrated with numerical results. Using C, CUDA C, Matlab GPU, and ACEnet, we make comparisons between serial and parallel algorithms and among computations with and without GPU and CUDA, different types of GPUs, and personal computers and ACEnet. A maximum of 26.77 times of speed-up is achieved, which could be further boosted with development of new hardware in the future. The acceleration in runtime will make many investigations possible and will pave the way for studies of large-scale computational electromagnetic problems that were previously impractical. This is a field that definitely invites more in-depth studies.

LIST OF ABBREVIATIONS USED

ACEnet	Atlantic Computational Excellence Network
BEM	Boundary Element Method
CCM	custom computing machines
CPU	Central Processing Unit
CUBLAS	CU Basic Linear Algebra Subroutines
CUDA	Compute Unified Device Architecture
CUFFT	CU Fast Fourier Transform
DDA	Discrete Dipole Approximation
DRAM	dynamic random access memory
DSM	Distributed Shared Memory
EM	Electromagnetics
FDM	Finite Difference Method
FDTD	Finite-Difference Time-Domain
FEM	Finite Element Method
FMM	Fast Multi-pole Method
FPGA	field-programmable gate arrays
GPGPU	General-Purpose Graphics Processing Unit
GPU	Graphics Processing Unit
HPC	High-Performance Computing
MIMD	multiple-instruction-multiple-data
MISD	multiple-instruction-single-data
MLFMM	multi-level fast multi-pole methods
MoM	Method of Moments
NVCC	NVIDIA CUDA Compiler
PDE	partial differential equations
PTX	Parallel Thread eXecution
SFP	special function processor

SIMD	single-instruction-multiple-data
SISD	single-instruction-single-data
SM	Shared Memory
SMP	Symmetrical Multi-Processing
TFL	Thread For Line
TFP	Thread For Point
TLM	Transmission Line Matrix
TPC	Texture Processor Cluster

ACKNOWLEDGEMENTS

I extend heartfelt appreciation to my supervisor, Dr. Zhizhang Chen, who introduced me to this special and challenging research topic and motivated me throughout the entire course of this thesis. He has the attitude and the substance of a researcher – continually and convincingly conveying a spirit of adventure through exploration. I received generous support and encouragement from him and also learned a lot from him.

As well, I would like to express my gratitude to my co-supervisor, Dr. Poman So, for his insightful comments, meticulous suggestions, and careful modifications. They are very important to my work and are certainly a huge help to me.

My thanks also go to my committee members, Dr. Sergey Ponomarenko and Dr. William Phillips, for their willing participation in the defense process and their kind advice.

I am particularly grateful to all my colleagues at the RF/Microwave Wireless Research Laboratory for their long-term support and kindness, especially Shunchuan Yang. Without their help, this thesis would not have materialized.

I owe a deep thank you to my best friends, Han Zhang and Luyu Wang, for their faithful companionship and warm encouragement over the past two memorable years.

Last but not least, I would like to thank my wonderful parents. Without their love, support, patience, and persistent encouragement throughout my degree, I would not have made it. This thesis is dedicated to them.

CHAPTER 1 INTRODUCTION

This chapter introduces the research background and motivation. The thesis outline is given at the end.

1.1 RESEARCH BACKGROUND

In recent years, GPU (Graphics Processing Unit) programming techniques are increasingly being used in a range of scientific and engineering computer simulations [1]. In the area of field simulation, reports on this technique have begun to increase as well, particularly since the release of NVIDIA's CUDA (Compute Unified Device Architecture) [2], a powerful and simple-to-use programmer environment available for NVIDIA cards. CUDA makes GPU computing accessible to developers who are not necessarily experts in computer graphics.

Initially, GPUs were not designed for general purpose programming, and high level programming languages were not readily available. Programmers were thus required to learn the intricacies of specialized, low-level hardware languages. For instance, the FDTD implementations in [3], [4] and [5] are based on OpenGL. Meanwhile, CUDA is a general purpose parallel computing architecture. To program CUDA's architecture, developers can use C, which can then be run at high performance on a CUDA-enabled processor. The CUDA architecture and its associated software provide a small set of extensions to standard programming languages, like C, that enable straightforward implementation of parallel algorithms [6].

Therefore, we can focus on the task of parallelization of algorithms instead of spending all of our time on their implementation. The CPUs and GPUs are treated as separate devices that have their own memory spaces. Simultaneous computation on both CPUs and GPUs are also allowed by this configuration without contention for memory resources [7].

1.2 MOTIVATION

The computational power of graphics cards have been utilized within numerical electromagnetic problems. Among these, some work (e.g., [8] - [12]) is concerned with the problems of FDTD simulation of EM fields. The problems treated thus far, however, are rather limited in their structures and calculation procedures. If we are able to treat most of calculations and procedures only by basic FDTD equations like wave propagation calculation, it would be comparatively easy to extract the GPU's inherent acceleration capability because basic FDTD algorithm is essentially suited to the multi-thread parallel computing function of GPU. For complex problems, the type of algorithm employed and the computer circumstances being used are deeply relevant to the computation speed. In other words, for complex problems, discussions of GPU implementation algorithm are quite valuable and significant. Developments in the design of GPUs have occurred at a much faster pace than with CPUs, and powerful processing units have been designed solely for the processing of computer graphics. For instance, the current generation of GPU-based NVIDIA Tesla C1060 Computing Processors is running at approximately 1.3 GHz, with a 512 bit data and memory bandwidth of 102 GB/sec [13]. While GPU clock speed seems slow compared to modern 3.8 GHz Pentium CPUs or 3.0 GHz Core Duos, the parallelism provided by the graphics cards enables better efficiency in computations [9]. Due to this potential in faster computations, GPU programming and CUDA have caught the attention of the scientific and engineering computing community.

In this thesis, we explain how to implement GPU versions of some simulation algorithms to achieve improved acceleration performance, especially on ACEnet. The attainable efficiency is demonstrated by discussing studies of some interaction problems and implementing the algorithms on off-the-shelf inexpensive GPUs. The wrap-up shows the consistency of the results between serial and parallel algorithms. Furthermore, observable speed-up is obtained compared with traditional computation.

1.3 Thesis Outline

This thesis is organized as follows:

Chapter 2 introduces the computational EM methods and their applications, including the Finite Element Method, the Moment Method, the Finite-Difference Time-Domain Method, Yee cell, and differentials of Maxwell's equations.

Chapter 3 focuses on GPU programming and CUDA architecture. A short summary of the various types of memory available on a GPU are given here, and differences between a CUDA GPU and a CPU are discussed. CUDA's potential in reducing computation time for the FDTD method is also explored.

Chapter 4 discusses implementations of CUDA-enabled FDTD on GPUs. Memory allocation and coalescing requirements are also discussed. In this chapter, the complexity involved in programming using GPU framework is introduced, and difficulties involved in implementing two- and three-dimensional grids and blocks of threads on CUDA are explained.

Chapter 5 investigates detailed acceleration methods. Performance optimization revolves around some basic strategies, including maximizing parallel execution to achieve maximum utilization and optimizing memory usage to achieve maximum speed-up. Tests were run and accelerated using various approaches on different platforms. Thorough comparisons are provided.

Chapter 6 summarizes the thesis and suggests potential future research directions for parallel methods. As a wrap-up, this chapter presents an overall conclusion of the thesis as well as some improvements that can be made to in-depth acceleration.

CHAPTER 2 OVERVIEW OF THE FINITE DIFFERENCE TIME DOMAIN (FDTD) METHOD

2.1 COMPUTATIONAL EM METHODS AND THEIR APPLICATIONS

Since the establishment of basic Maxwell's equations of electromagnetic fields, EM and wave theory have been used in various research areas such as geology, life science, medicine, material science, and information technology. EM and wave theory are also used in other fields like radio propagation, antennas, radars, optical communications, mobile communications, geological exploration, and bio-electromagnetics [14]. In practical problems, electromagnetic analysis is often quite complex and there are usually no easy analytical solutions to the boundary value problems of Maxwell's equations. Moreover, electronic analysis is difficult, expensive, and time-consuming. Therefore, it is much more practical to explore a solution for an electromagnetic theoretical equation to obtain the numerical results of field characteristics for a specific environment.

Earlier studies of EM numerical methods focused on the electrostatic field and time-harmonic steady state of Maxwell's equations Finite Difference Method (FDM). The Finite Element Method extended from structural mechanics calculations, high-frequency asymptotic, Method of Moments, and some other integral methods. In 1966, Yee presented the Finite-Difference Time-Domain method, which could directly solve the time-domain equations [15]. In the 1970s, several computational electromagnetic methods were proposed as improvements, such as Transmission Line Matrix (TLM) [16], Discrete Dipole Approximation (DDA) [17], Fast Multi-pole Method (FMM) [18], and Boundary Element Method (BEM) [19].

Next, we will introduce several widely applied computational electromagnetic methods.

2.1.1 Finite Element Method

The Finite Element Method is based on the variational principle and is widely used in various types of physical field numerical calculation. In the 1940s, Hrennikoff [20], McHenry [21] and Courant [22] originally proposed the idea of a mesh discrete element, but their approaches differed significantly. Hrennikoff's work discretized the domain by using a lattice analogy, while Courant's approach divided the domain into finite triangular subregions to solve second-order elliptic partial differential equations (PDEs) that arise from the problem of torsion of a cylinder. Courant's contribution was evolutionary, drawing on a large body of earlier results for PDEs developed by Rayleigh, Ritz, and Galerkin [23]. The finite element method obtained its real impetus in the 1960s and 1970s with the developments of J.H. Argyris and co-workers at the University of Stuttgart, R.W. Clough and co-workers at UC Berkeley, O.C. Zienkiewicz and co-workers at the University of Swansea, and Richard Gallagher [24] and co-workers at Cornell University. A rigorous mathematical basis to the finite element method was provided in 1973 with a publication by Strang and Fix [25]. The method has since been generalized for the numerical modeling of physical systems in a wide variety of engineering disciplines, e.g., electromagnetism, heat transfer, and fluid dynamics [26].

FEM utilizes mesh interpolation to divide the continuous structure into a finite number of non-overlapping cells and to select the nodes as the interpolation points of functions to solve. Many physical phenomena in engineering and science can be described in terms of partial differential equations (PDE). In general, solving these equations by classical analytical methods for arbitrary shapes is almost impossible. The finite element method is a numerical approach by which these PDE can be solved approximately. It is widely used in diverse fields to solve static and dynamic problems of solid or fluid mechanics, electromagnetics, biomechanics, etc.

There are two key words in FEM: discretization and interpolation. First, we need to discretize the equations. The subdivision of a whole domain into simpler parts has several advantages, such as accurate representation of complex geometry, inclusion of dissimilar material properties, easy representation of the total solution, and capture of local effects [27]. We then can manage to solve the discrete system of equations, eventually followed by interpretation of the obtained results and errors analysis. Generally, the advantage of finite element analysis is that it can move us toward a solution when we have a very complicated shape undergoing very complicated loads (e.g., temperature, vibration, earthquakes, yielding, spread of plasticity), or when we wish to examine a part in greater detail (more common in mechanical engineering).

2.1.2 Moment Method

The method of moments (MoM) or boundary element method (BEM) is a numerical computational method of solving linear partial differential equations that have been formulated as integral equations (i.e., in boundary integral form). It can be applied in many areas of engineering and science, including fluid mechanics, acoustics, electromagnetics, fracture mechanics, and plasticity [28]. In 1962, Dr. Mei proposed MoM to solve integral equations [29], and Harrington made a comprehensive in-depth analysis in terms of using the moment method to solve electromagnetic problems [30] [31].

MoM has become more popular since the 1980s. It is especially efficient in terms of computational resources for problems with a small surface/volume ratio, since it requires calculating only boundary values rather than values throughout space. Conceptually, it works by constructing a "mesh" over the modeled surface. For many problems, however, MoM would be substantially less efficient than volume-discretization methods (i.e., finite element method, finite difference method, finite volume method). Boundary element formulations typically give rise to fully populated matrices, which means that the storage requirements and computational time tend to increase based on the square of the problem size. Finite element matrices, by contrast, are typically banded and the storage requirements for the system matrices grow linearly with the problem size. We can use compression techniques (e.g., multi-pole expansions or adaptive cross approximation/hierarchical matrices) to ameliorate these problems, though at the cost of added complexity and with a success-rate that depends heavily on the nature and geometry of the problem [28].

Moment method is relatively simple in terms of the analytical part, but the resulting matrix is often large sparse matrix, or full matrix, which brings in heavy computing workloads. The three-dimensional problem solving is especially hard in this case with very high memory storage requirements. Therefore, in recent years the development of MoM concentrated on an effective sparse solution of the three-dimensional structures, ensuring the accuracy of the premise. Examples of this are fast multi-pole methods (FMM) [32] and multilevel fast multi-pole methods (MLFMM) [33]. MoM is applicable to problems for which Green's functions can be calculated, which usually involve fields in linear homogeneous media. This places considerable restrictions on the range and generality of problems suitable for boundary elements [34]. The formulation can contain nonlinearities, although they generally introduce volume integrals that require the volume to be discretized before solution, removing an oft-cited advantage of MoM.

2.1.3 Finite-Difference Time-Domain Method

The FDTD method is a numerical method introduced by Yee in the 1960s [15] to solve the differential form of Maxwell's equations in time-domain. Although the method has existed for over four decades, enhancements to improve FDTD are continuously being published [35]. It is easy to understand and has an exceptionally simple implementation for a full wave solver. It is at least an order of magnitude less work to implement a basic FDTD solver than either an FEM or MoM solver. FDTD is the only technique where one person can realistically implement oneself in a reasonable time frame, but even this will be for a quite specific problem [36].

Maxwell's curl equations are discretized by the FDTD method into time and spatial domains. The electric fields are generally located at the edge of the Yee Cell, while the magnetic fields are located at the center of the Yee Cell, as shown in Figure 2.1.3.1. In three dimensions, the number of cells in one time step can easily be in orders of millions. A dimension of $100\times100\times100$ cells already yields a total of one million cells. For example, in [37], a high resolution head model of an adult male has a total of 4,642,730 Yee cells, with each cell having a dimension of 1×1×1 mm^3.

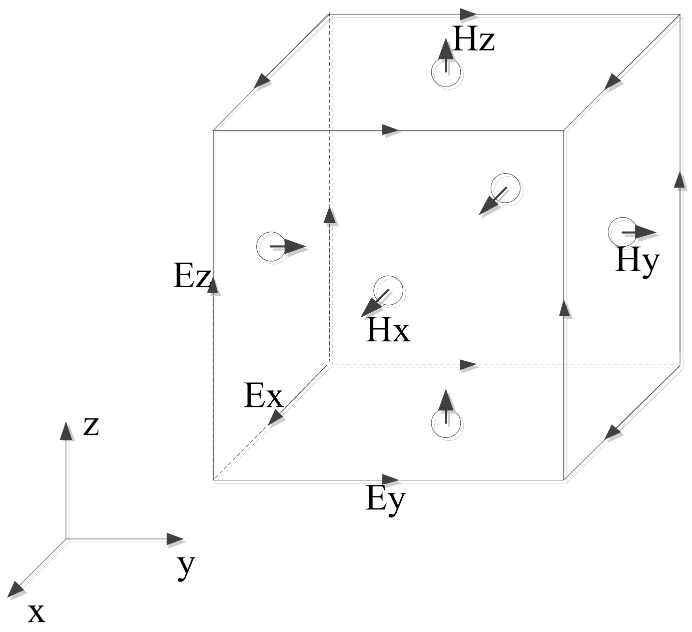


Figure 2.1.3.1: Position of the electric and magnetic fields in Yee's scheme.

FDTD is a versatile modeling technique used to solve Maxwell's equations. It is intuitive, so users can easily understand how to handle it and know what to expect. It is also a time-domain technique. When a broadband pulse (such as a Gaussian pulse) is used as the source, the response of the system over a wide range of frequencies can be obtained with a single simulation. This is useful in applications where resonant frequencies are not exactly known or anytime that a broadband result is desired. Since FDTD calculates the E and H fields everywhere in the computational domain as they evolve in time, it lends itself to providing animated displays of the electromagnetic field movement through the model. This type of display is useful in understanding what is going on in the model, and to help ensure that the model is working correctly. FDTD uses the E and H fields directly. Since most EMI/EMC modeling applications are interested in the E and H fields, it is convenient that no conversions must be made after the simulation has run to get these values [38].

2.2 Yee's FDTD Grids and Discretization of Maxwell's Equations

Electromagnetic processes in the media space meet the requirements of Maxwell's curl equations:

$$\nabla \times \boldsymbol{H} = \frac{\partial \boldsymbol{D}}{\partial t} + \boldsymbol{J} \qquad \text{(2-2-1a)}$$

$$\nabla \times \boldsymbol{E} = -\frac{\partial \boldsymbol{B}}{\partial t} \qquad \text{(2-2-1b)}$$

Combined with structural equations of isotropic media:

$$\boldsymbol{D} = \varepsilon \boldsymbol{E} \qquad \text{(2-2-2a)}$$

$$\boldsymbol{B} = \mu \boldsymbol{H} \qquad \text{(2-2-2b)}$$

$$\boldsymbol{J} = \sigma \boldsymbol{E} \qquad \text{(2-2-2c)}$$

In a Cartesian coordinate system, we can expand them to get scalar Maxwell's equations, as follows:

$$\frac{\partial H_x}{\partial t} = \frac{1}{\mu}\left(\frac{\partial E_y}{\partial z} - \frac{\partial E_z}{\partial y}\right) \qquad \text{(2-2-3a)}$$

$$\frac{\partial H_y}{\partial t} = \frac{1}{\mu}\left(\frac{\partial E_z}{\partial x} - \frac{\partial E_x}{\partial z}\right) \qquad \text{(2-2-3b)}$$

$$\frac{\partial H_z}{\partial t} = \frac{1}{\mu}\left(\frac{\partial E_x}{\partial y} - \frac{\partial E_y}{\partial x}\right) \qquad \text{(2-2-3c)}$$

$$\frac{\partial E_x}{\partial t} = \frac{1}{\varepsilon}\left(\frac{\partial H_z}{\partial y} - \frac{\partial H_y}{\partial z} - \sigma E_x\right) \qquad \text{(2-2-3d)}$$

$$\frac{\partial E_y}{\partial t} = \frac{1}{\varepsilon}\left(\frac{\partial H_x}{\partial z} - \frac{\partial H_z}{\partial x} - \sigma E_y\right) \qquad \text{(2-2-3e)}$$

$$\frac{\partial E_z}{\partial t} = \frac{1}{\varepsilon}\left(\frac{\partial H_y}{\partial x} - \frac{\partial H_x}{\partial y} - \sigma E_z\right) \qquad \text{(2-2-3f)}$$

Yee did the alternate sampling for the electromagnetic components in space, so that each electric field component is surrounded by four magnetic field components and each magnetic field component is surrounded by four electric field components. As shown in Figure 2.2.1, this is called a Yee Cell. The sampling basis is simple, intuitive, and satisfies the Gauss theorem [39]. Applying it to the scalar Maxwell's curl equations (2-2-3), we obtain numerical solutions to the electromagnetic problems.

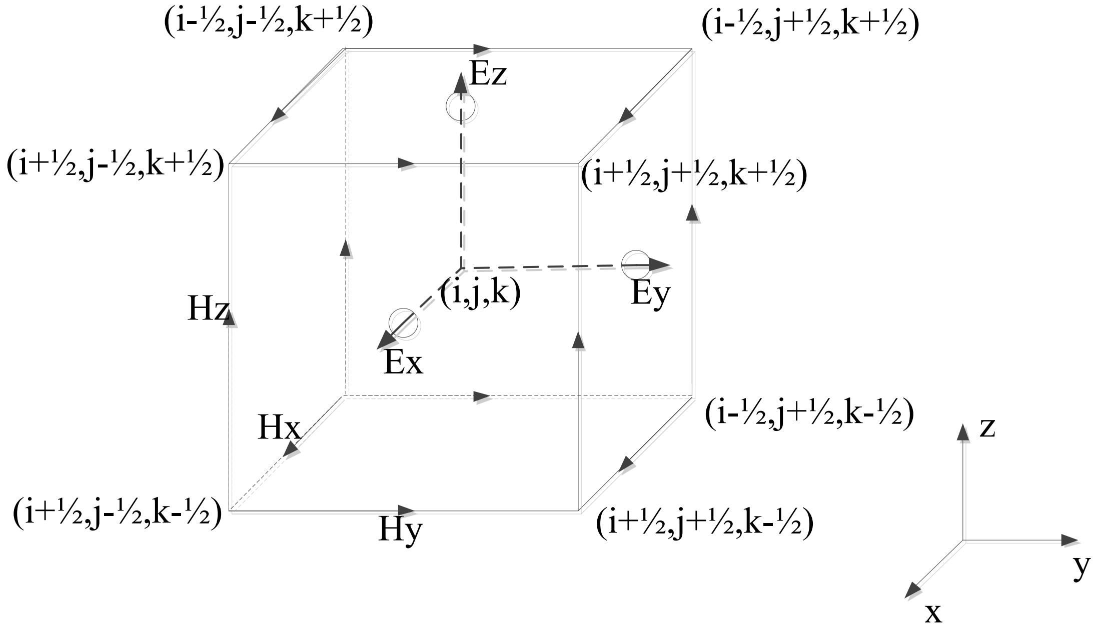


Figure 2.2.1: Yee's cell in FDTD.

If we use second-order central difference to replace the differential in (2-2-3) and do the sampling in time domain for the E and H every half-step length, we can obtain the time-domain electromagnetic field recurrence formula:

$$H_x^{n+\frac{1}{2}}\left(i,j+\frac{1}{2},k+\frac{1}{2}\right) = H_x^{n-\frac{1}{2}}\left(i,j+\frac{1}{2},k+\frac{1}{2}\right) + \frac{\Delta t}{\mu}\left[\frac{E_y^n\left(i,j+\frac{1}{2},k+1\right) - E_y^n\left(i,j+\frac{1}{2},k\right)}{\Delta z} - \frac{E_z^n\left(i,j+1,k+\frac{1}{2}\right) - E_z^n\left(i,j,k+\frac{1}{2}\right)}{\Delta y}\right] \tag{2-2-4a}$$

$$H_y^{n+\frac{1}{2}}\left(i+\frac{1}{2},j,k+\frac{1}{2}\right) = H_y^{n-\frac{1}{2}}\left(i+\frac{1}{2},j,k+\frac{1}{2}\right) + \frac{\Delta t}{\mu}\left[\frac{E_z^n\left(i+1,j,k+\frac{1}{2}\right) - E_z^n\left(i,j,k+\frac{1}{2}\right)}{\Delta x} - \frac{E_x^n\left(i+\frac{1}{2},j,k+1\right) - E_x^n\left(i+\frac{1}{2},j,k\right)}{\Delta z}\right] \tag{2-2-4b}$$

$$H_z^{n+\frac{1}{2}}\left(i+\frac{1}{2},j+\frac{1}{2},k\right) = H_z^{n-\frac{1}{2}}\left(i+\frac{1}{2},j+\frac{1}{2},k\right) + \frac{\Delta t}{\mu}\left[\frac{E_x^n\left(i+\frac{1}{2},j+1,k\right) - E_x^n\left(i+\frac{1}{2},j,k\right)}{\Delta y} - \frac{E_y^n\left(i+1,j+\frac{1}{2},k\right) - E_y^n\left(i,j+\frac{1}{2},k\right)}{\Delta x}\right] \tag{2-2-4c}$$

$$E_x^{n+1}\left(i+\frac{1}{2},j,k\right) = \frac{2\varepsilon - \Delta t\sigma}{2\varepsilon + \Delta t\sigma} E_x^n\left(i+\frac{1}{2},j,k\right) + \frac{2\Delta t}{2\varepsilon + \Delta t\sigma} \times \left[\frac{H_z^{n+\frac{1}{2}}\left(i+\frac{1}{2},j+\frac{1}{2},k\right) - H_z^{n+\frac{1}{2}}\left(i+\frac{1}{2},j-\frac{1}{2},k\right)}{\Delta y} - \frac{H_y^{n+\frac{1}{2}}\left(i+\frac{1}{2},j,k+\frac{1}{2}\right) - H_y^{n+\frac{1}{2}}\left(i+\frac{1}{2},j,k-\frac{1}{2}\right)}{\Delta z}\right] \tag{2-2-4d}$$

$$E_y^{n+1}\left(i,j+\frac{1}{2},k\right) = \frac{2\varepsilon-\Delta t\sigma}{2\varepsilon+\Delta t\sigma}E_y^n\left(i,j+\frac{1}{2},k\right)+\frac{2\Delta t}{2\varepsilon+\Delta t\sigma} \times [\frac{H_x^{n+\frac{1}{2}}\left(i,j+\frac{1}{2},k+\frac{1}{2}\right)-H_x^{n+\frac{1}{2}}\left(i,j+\frac{1}{2},k-\frac{1}{2}\right)}{\Delta y} - \frac{H_z^{n+\frac{1}{2}}\left(i+\frac{1}{2},j+\frac{1}{2},k\right)-H_z^{n+\frac{1}{2}}\left(i-\frac{1}{2},j+\frac{1}{2},k\right)}{\Delta x}]$$

(2-2-4e)

$$E_z^{n+1}\left(i,j,k+\frac{1}{2}\right) = \frac{2\varepsilon-\Delta t\sigma}{2\varepsilon+\Delta t\sigma}E_z^n\left(i,j,k+\frac{1}{2}\right)+\frac{2\Delta t}{2\varepsilon+\Delta t\sigma} \times [\frac{H_y^{n+\frac{1}{2}}\left(i,j+\frac{1}{2},k+\frac{1}{2}\right)-H_y^{n+\frac{1}{2}}\left(i,j-\frac{1}{2},k+\frac{1}{2}\right)}{\Delta x} - \frac{H_x^{n+\frac{1}{2}}\left(i,j+\frac{1}{2},k+\frac{1}{2}\right)-H_x^{n+\frac{1}{2}}\left(i,j-\frac{1}{2},k+\frac{1}{2}\right)}{\Delta y}]$$

(2-2-4f)

In the equations above, Δx, Δy, Δz are space steps at three directions in the Cartesian coordinates, respectively, while Δt is the time step. $(i+\frac{1}{2},j,k)$ and its similar forms are coordinates of Yee grid point field components, as illustrated in Figure 3.1. σ and ε are conductivity and permittivity at the current calculation point. Superscripts like $n+1$ and $n+\frac{1}{2}$ are index values of the time steps. It can be seen that each component of $\boldsymbol{E}$ and $\boldsymbol{H}$ at that time has the same difference of half a step, and thus takes the alternate recursion in the time domain, updating the EM field distribution over time.

From Figure 3.1 and Equation (3.1.4), we know that in FDTD recursion of time domain, the computation of one field value at all positions in space is conducted in accordance with a unified recursive formula. Additionally, the field value of a grid point is only related to its previous field value and its four surrounding grid points, without regard to the overall distribution of the electromagnetic field.

2.3 FDTD Enabling Hardware Platform

We know that Maxwell's curl equations in free space for one-dimension are

$$\frac{\delta E_x}{\delta t} = -\frac{1}{\varepsilon_0}\frac{\delta H_y}{\delta z} \qquad (2\text{-}3\text{-}1a)$$

$$\frac{\delta H_y}{\delta t} = -\frac{1}{\mu_0}\frac{\delta E_x}{\delta z} \qquad (2\text{-}3\text{-}1b)$$

Using the finite difference method of central approximation and rearranging [40], the electric fields and magnetic fields are calculated alternately over the entire spatial domain at time step n. This process continues over all time steps until convergence is achieved. Depending on the size of the computation domain, millions of iterations could be required to solve the differential equations. However, the benefit of using the FDTD method is that it only requires exchange of data with neighbouring cells, without regard to the overall distribution of the electromagnetic field. This provides an easy way of decomposition into multiple sub-process calculations and thus can be a natural parallelizable algorithm. Because of this, FDTD has drawn intense interest from researchers.

2.3.1 CPU-Based FDTD

In the 1980s, reports emerged regarding implementing FDTD parallel computing on large vector machines [41] [42]. Before PC networking technology was widely applied, the use of the FDTD method on mainframe computers dominated this field. However, these computers or systems were too expensive for general researchers and so the method not been extensively applied. Then, with the development of PC and network equipment, parallel computing systems using low-cost computers began to emerge. Initially from the parallel 386 CPU [43] to multicore Xeon CPUs [44], researchers have continuously been promoting the parallel models and algorithms.

Parallel computing hardware devices based on traditional CPU have shared or distributed memory parallel systems. With the popularity of multi-core processors, there are increasing numbers of distributed memory parallel systems. Parallel implementations are mostly in view of FDTD domain decomposition, which divides the whole computing area into a plurality of sub-areas according to the number of sub-computing units. Each sub-unit is responsible for processing iterations in a sub-area of the electromagnetic field. In the course of forwarding time steps, there will be synchronizations between the cells. Moreover, if the computation is implemented on a distributed memory parallel system, there have to be data transferring between the cells.

Along with the further improvement of parallel computing systems, mature parallel computing software and models on FDTD are also promoted. These include MPI [45] and PVM [46] based on message passing, OpenMP based on single computer multi-threaded programming and the distributed memory parallel systems combining the two models. W. Yu (2007) conducted a detailed study on the FDTD parallel algorithm, border data exchange, computational efficiency and other issues using parallel computer systems at the University of Pennsylvania and IBM Blue Gene supercomputer, which took MPI as the basic model. In a good network environment, the parallel efficiency was able to reach up to 90% with over 200 processors [47]. In 2004, a combination of MPI and OpenMP was proposed and achieved the implementation of FDTD parallel computing on SGI Origin 2000 [48]. In 2008, one more parallel solution was realized on hyper-threading clusters composed of Pentium four CPUs [49]. However, due to limitations in network equipment, the efficiency was lower than mainframe computers and declined further with each additional computing unit.

2.3.2 FPGA-Based FDTD

The software execution of the FDTD method has the problem of limited computational speed. An effective approach to construct parallelism is the implementation with field-programmable gate arrays (FPGAs). The FDTD algorithm is computationally intensive but has a simple computation kernel, making it suitable for implementation on FPGA co-processor boards of high performance parallel computers. FPGA is one of the programmable-logic devices where end-users can configure their own logic circuits over and over on the semiconductor chip [50]. So far, fundamental floating-point operations and their applications on FPGAs have been studied.

In 2005, FPGA-based floating-point co-processors were proposed for the matrix-vector multiplication and the matrix multiplication in [51] and [52]. Although such fundamental operations can be utilized for the FDTD method, these co-processors have an essential problem in that data transfer to FPGAs becomes bottlenecked. Since the I/O bandwidth of an FPGA chip is limited, building custom computing machines (CCMs) as a co-processor is essentially not scalable due to the I/O bottleneck in streaming data to FPGAs from external components. Durbano et al. designed 12 dedicated computing engines to do the parallel calculation, as shown in Figure 2.3.2.1 [53], while Kawaguchi et al. designed different hardware for different computing portions so that they could process in parallel for different field values. They also utilized specific PML for hardware circuits and optimized for memory modules [54].

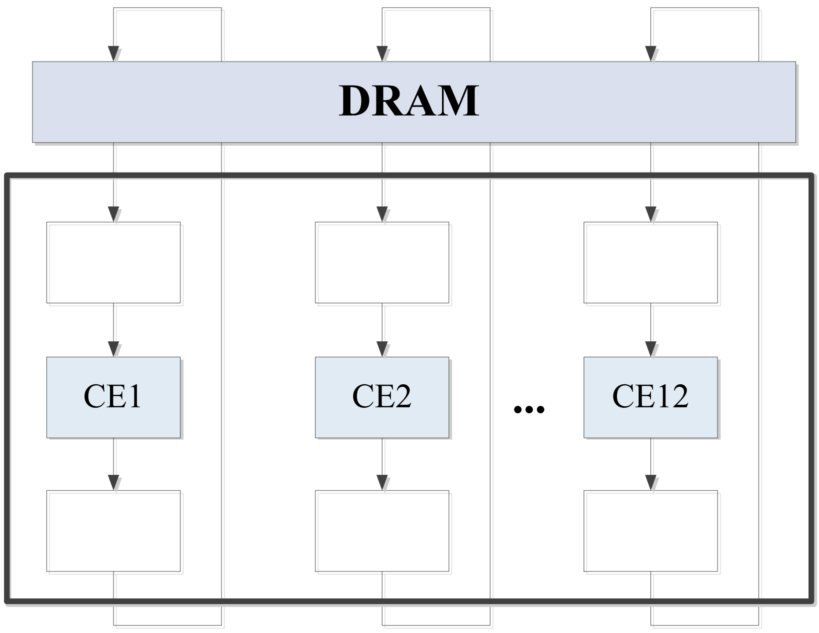


Figure 2.3.2.1: FDTD computing design using FPGA.

Apart from the relatively high cost of hardware, the technique also had several problems in terms of issues like memory access and multi-parallel. Additionally, it is difficult to implement complex FDTD using hardware programming language. Due to all of these issues, FPGA-based FDTD parallel computing systems are not widely applied.

2.3.3 GPU-Based FDTD

As one of the main components of PC, GPU has a higher floating point computing capability compared to CPU. As well, its integrated process development is even faster than CPU, which offers a cost-effective solution for scientific computing parallelization. Meanwhile, FDTD has a strong parallelizability, so FDTD parallel computing on GPU has always attracted considerable attention.

The adoption of GPU computing for EM numerical methods is a relatively new issue, even though it is featuring a positive trend. This is also because of the recent publication of the Compute Unified Device Architecture (CUDA), a powerful and simple-to-use programmer environment available for NVIDIA cards that makes GPU computing accessible to developers who are not experts at computer graphics. Several EM numerical methods are amenable to benefit from appealing GPU characteristics [55] [56] [57]. Among them, the FDTD algorithm is perhaps the best suited to be implemented in a GPU-enabled fashion. It is the simplest to describe, as it is an inherently data-parallel algorithm. It was therefore chosen in this thesis as a case study for demonstrating GPU capabilities, and for illustrating the software design and implementation efforts required for developing efficient parallel codes on a GPU.

Traditional General-Purpose Graphics Processing Unit (GPGPU) implementation of FDTD is relatively complex as it needs to map the entire calculation process to the graphics pipeline. Two-dimensional TM wave FDTD recursion of Hx and Hy proceeds as follows:

- Bind input parameters to texture units;
- Bind them to an object that needs to be rendered, which means to virtualize the whole area into a quadrilateral, and each component represents one single part;
- Load shader program and set its parameters. Write in accordance with recursive formula as the form of fragment program and load it into a storage area. One small section in Figure 2.3.3.1 is the compiled form of recursive formula. We can see the operations such as texture reading and multiply-add from it;
- Render a quadrilateral. Execute the fragment program process, conduct electromagnetic field recursion, and update Hx and Hy from the input field values;
- Read data from the texture buffer. Output the updated Hx and Hy value.

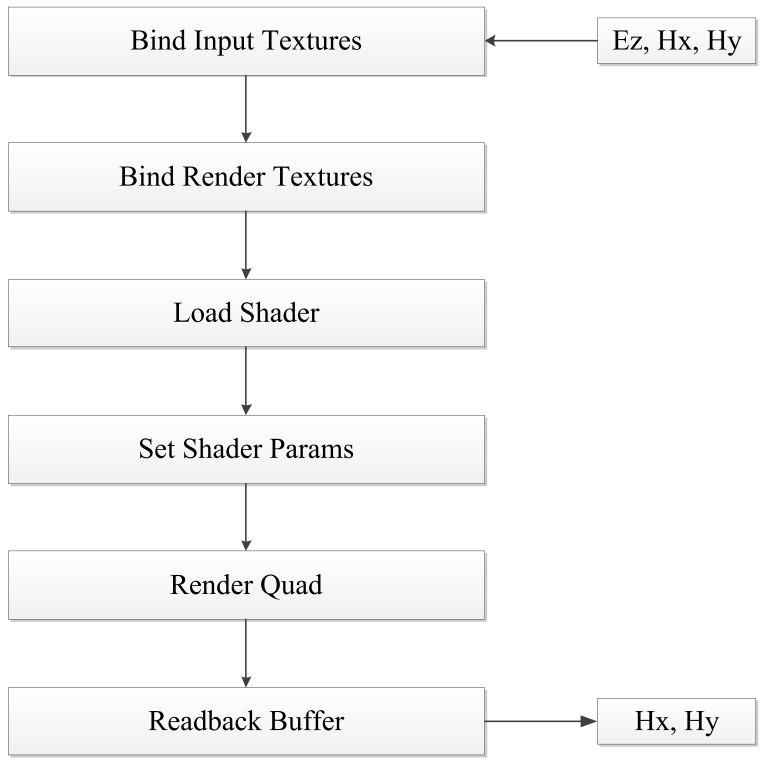


Figure 2.3.3.1: Traditional GPGPU parallel FDTD computing procedure.

This computing process involves numerous graphics concepts and the operation is complicated. In early GPU architecture, there were limitations using vertex and pixel processors to program, which did not take full advantage of the GPU floating point calculation superiority.

Since the release of NVIDIA CUDA, researchers have started to program in the novel GPU architecture without the help of graphics rendering. They only need to use pointers, memory replication, function calls, and some other familiar operations. In 2008, Sypek et al. realized fundamental FDTD parallel computation and achieved a great speed of 20 times faster for 3-D FDTD (unspecified type of GPU chip) [58]; Balevic et al. utilized 2-Dimensional block and realized FDTD based on wave equations upon GeForce 8800GTX (112sp); Valcarce et al. utilized 5 different kernels, calculated FDTD main area and PML area, realized 2-D FDTD computing under CPML absorbing boundary conditions, and worked out WiMAX signal distribution in the city.

CHAPTER 3 PARALLEL COMPUTING AND GPU ARCHITECTURE

3.1 Parallel Computing Technology

In the process of applying FDTD algorithm for electromagnetic simulation, the computed grids and time steps must be small enough so that the calculation requirements will be met with the increasing of the structure complexity. To reduce the simulation time, research has focused on parallel algorithms. FDTD has a natural characteristic of data parallelizability, which provides an excellent condition for its parallel computing implementation. To do this, we must first select the appropriate parallel system, then design the algorithm based on the parallel programming model, and finally implement it in the computing environment.

3.1.1 Hardware

Michael J. Flynn created one of the earliest classification systems for parallel (and sequential) computers and programs, now known as Flynn's taxonomy [59]. Flynn classified programs and computers by whether they were operating using a single set or multiple sets of instructions, and whether or not those instructions were using a single set or multiple sets of data. There are four major types of classification. The single-instruction-single-data (SISD) classification is equivalent to an entirely sequential program. The single-instruction-multiple-data (SIMD) classification is analogous to doing the same operation repeatedly over a large data set. This is commonly done in signal processing applications. Multiple-instruction-single-data (MISD) is a rarely used classification. While computer architectures to deal with this were devised (such as systolic arrays), few applications that fit this class materialized. Multiple-instruction-multiple-data (MIMD) programs are by far the most common type of parallel programs [60]. According to memory and computing unit distribution, computer systems can also be divided into Shared Memory (SM), Symmetrical Multi-Processing (SMP) and Distributed Shared Memory (DSM) architectures.

Conventional single-core CPU clusters and distributed computing architectures are typical DM parallel systems. As can be seen in Figure 3.1.1.1, this structure is quite suitable for parallel algorithms without computing unit communication. If the algorithm requires data communication between each computing unit, transit through other devices will be needed for messaging. Its performance can thus be greatly influenced by interconnected network devices. Meanwhile, DM architecture has an easy access to expand the scale and thus is extensively applied.

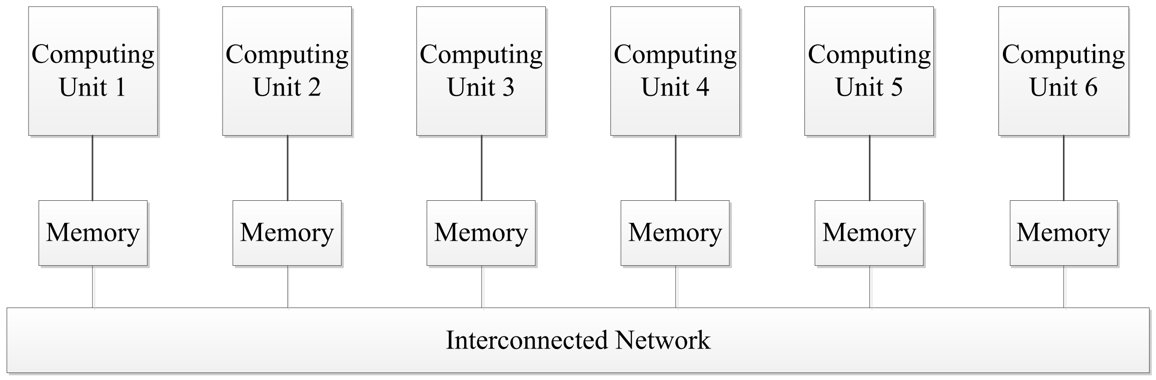


Figure 3.1.1.1: Parallel computing system: distributed memory architecture.

Multi-core CPUs and GPUs belong to shared memory parallel computing system architecture. As shown in Figure 3.1.1.2, all of its calculation units are able to access the memory in the same area via the bus of the device. There are cache devices in each calculation unit to accelerate the speed of data access. The data transferring process is not required for any algorithm, and the only thing that should be taken into account is synchronization of the calculation units.

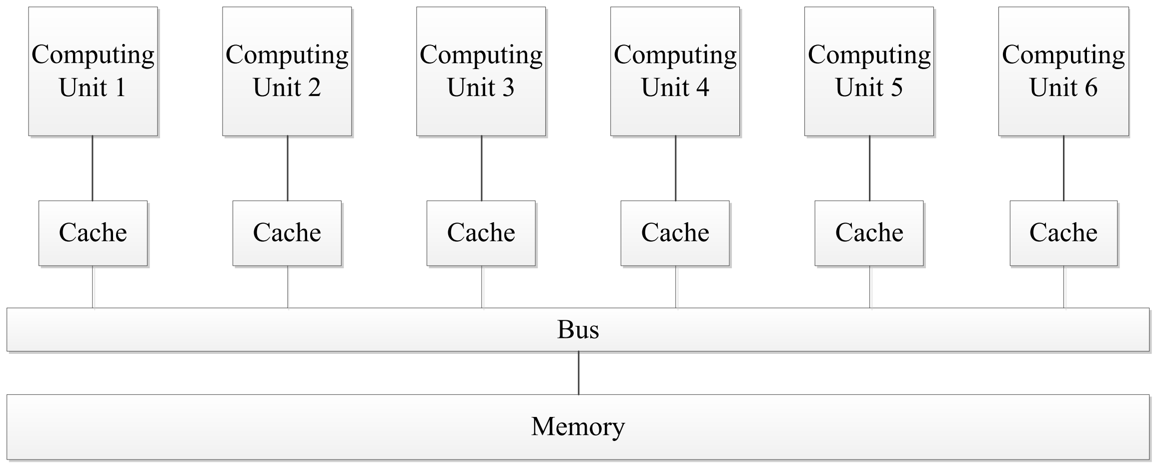


Figure 3.1.1.2: Parallel computing system: shared memory architecture.

Multi-core CPUs connected in parallel through a network and multi-GPU platform belonging to DSM architecture. As can be seen in Figure 3.1.1.3, within an SMP, each computing unit can access the same memory area. Every unit can have private or public cache, and a number of SMPs build up the DM architecture through interconnected equipment components. Between GPUs, data can be transferred via the host memory or network equipment.

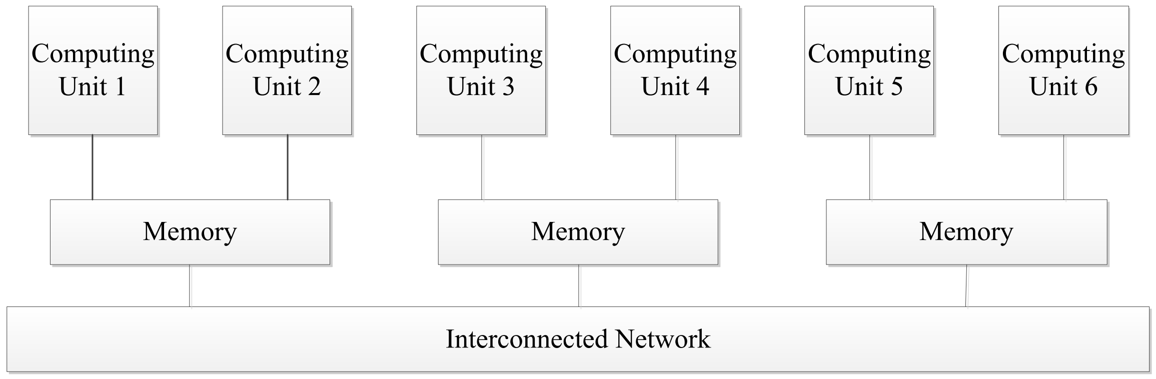


Figure 3.1.1.3: Parallel computing system: distributed shared memory architecture.

3.1.2 Programming Model

Using a parallel computing system as a hardware basis, we then select the appropriate software development model and design parallel algorithms to achieve the desired programming results. For different parallel computing systems, we can choose different development models and design different parallel algorithms. The implementation used in this thesis – Compute Unified Device Architecture (CUDA) – can also be regarded as a parallel model.

CUDA is hardware and software architecture introduced by NVIDIA in 2006 [61]. The aim in developing CUDA was to provide developers with access to the parallel computational elements of NVIDIA GPUs. CUDA architecture enables NVIDIA GPUs to execute programs written in various high-level languages such as C, FORTRAN, OpenCL and Direct Compute. The newest architecture of GPUs by NVIDIA (code-named "Fermi") also fully supports programming through the C++ language [62].

Because of advancements in technology, the processing power and parallelism of GPUs are continuously increasing. CUDA's scalable programming model makes it easy to provide this abstraction to software developers, allowing the program to automatically scale according to the capabilities of the GPU without any change in code, unlike traditional graphics programming languages such as OpenGL [63]. This is illustrated in Figure 3.1.2.1. A multithreaded program is partitioned into blocks of threads that execute independently from each other, so that a GPU with more cores will automatically execute the program in less time than a GPU with fewer cores.

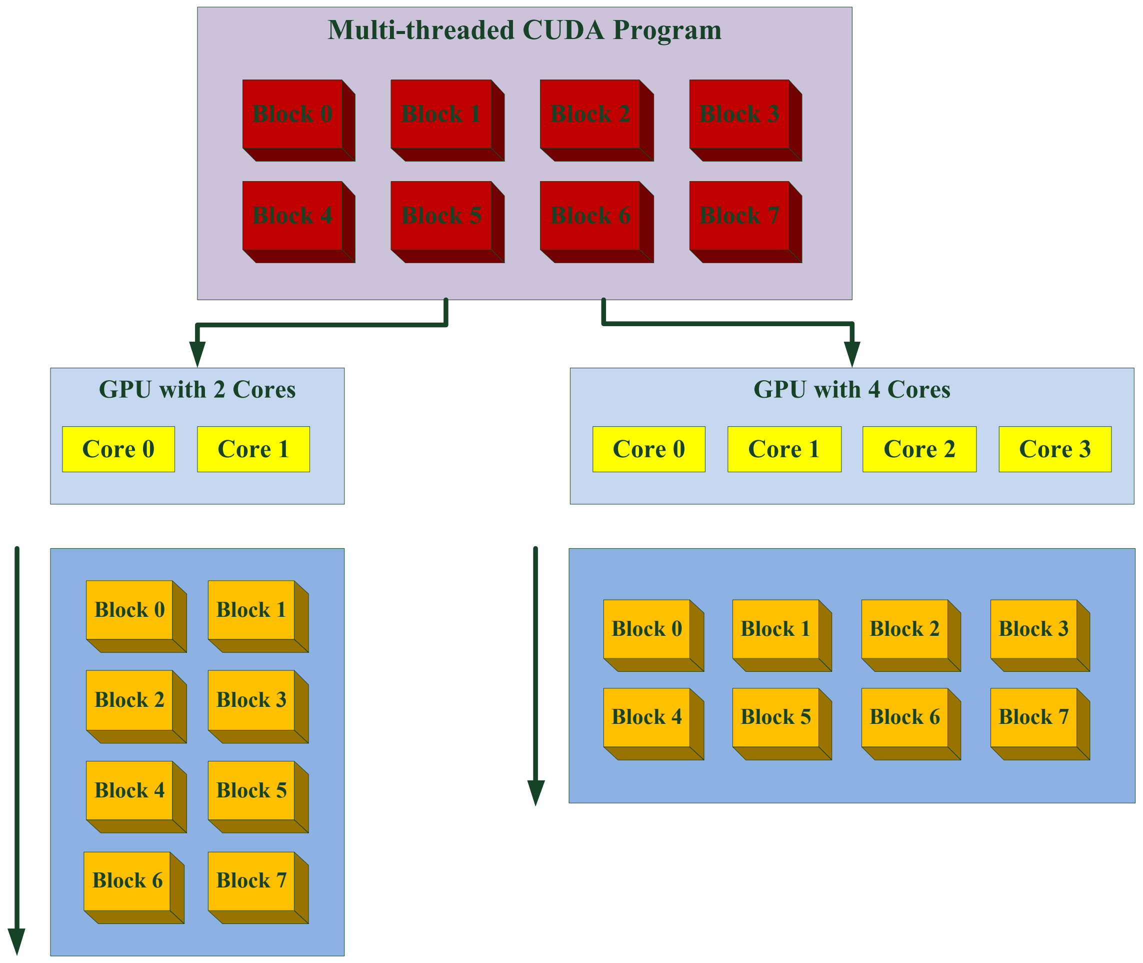


Figure 3.1.2.1: Partitioned multi-threaded program.

(Source: NVIDIA CUDA C programming guide version 4.2, 2012.)

Because the GPU and CPU both serve different purposes in a computer, their microprocessor architectures, as shown in Figure 3.1.2.2, are very different. GPU devotes more transistors to data processing. While CPUs currently have up to six processor cores (Intel Core i7-970), a GPU has hundreds. For example, the NVIDIA Tesla 20-series has 448 CUDA cores [62].

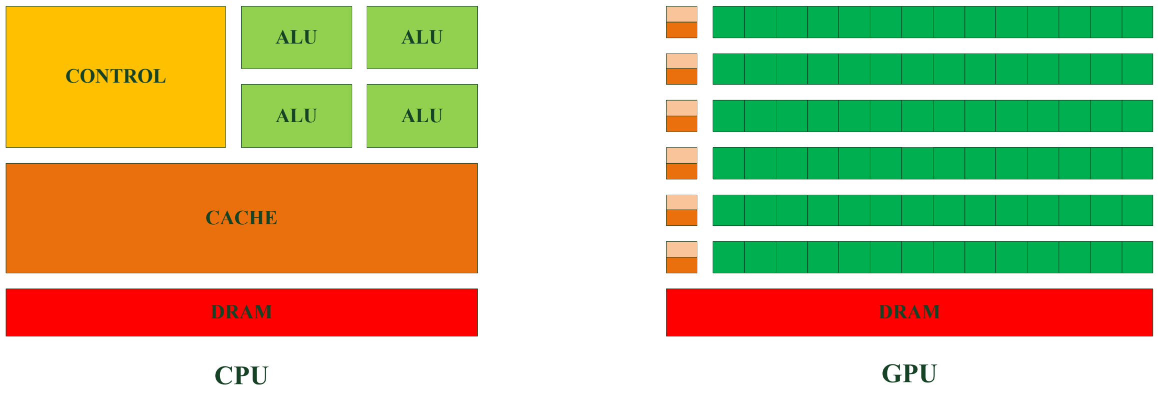


Figure 3.1.2.2: CPU and GPU microprocessor architectures.
(Source: NVIDIA CUDA C programming guide version 4.2, 2012.)

Compared to the CPU, the GPU devotes more transistors to data processing than to data caching and flow control. This allows GPUs to specialize in math-intensive, highly parallel operations compared to CPUs, which serve as multi-purpose microprocessors. Therefore, calculations of the FDTD algorithm are potentially much faster when executed on a GPU instead of a CPU. This is becoming increasingly true as graphics card vendors such as NVIDIA and AMD are now developing more graphics card for high performance computing (HPC) such as the NVIDIA Tesla [64].

CUDA has a single-instruction multiple-thread (SIMT) execution model where multiple independent threads execute concurrently using a single instruction [62]. CUDA GPUs have a hierarchy of grids, threads and blocks, as shown in Figure 3.1.2.3. Each thread has its own private memory. Shared memory is available per-block and global memory is accessible by all threads. This multi-threaded architecture model puts the focus on data calculations rather than on data caching. Thus, it can sometimes be faster to recalculate than to cache on a GPU.

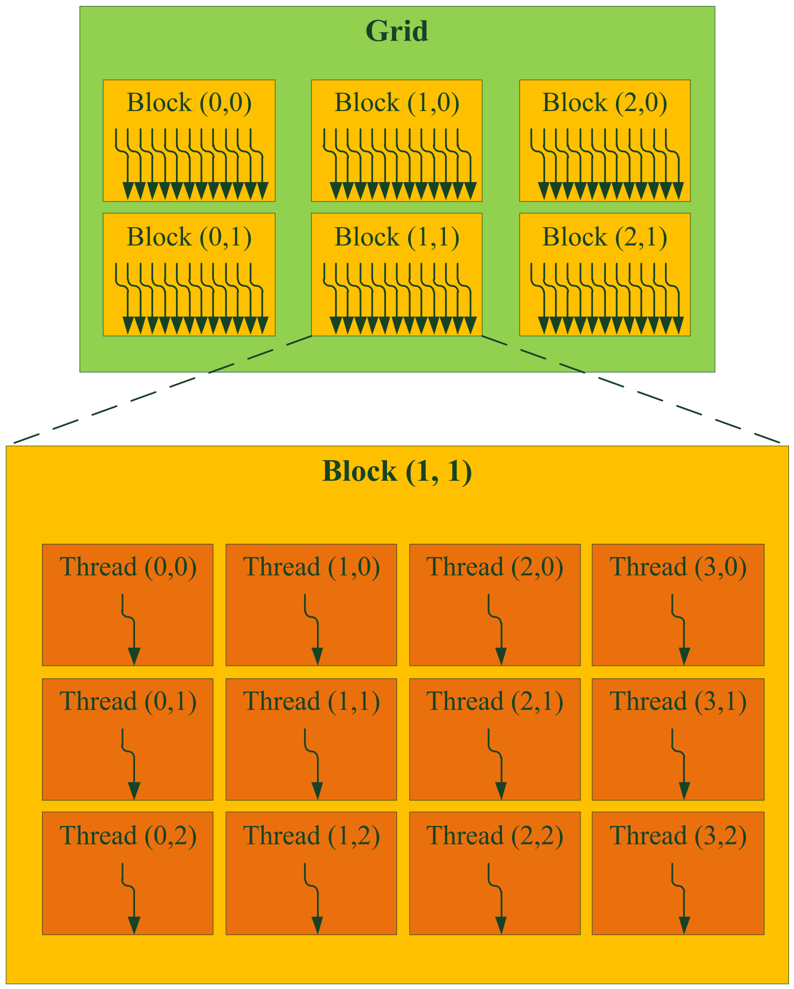


Figure 3.1.2.3: Hierarchy of threads, blocks and grid in CUDA.

(Source: NVIDIA CUDA C programming guide version 4.2, 2012.)

A CUDA program is called a kernel and is invoked by a CPU program. The CUDA programming model assumes that CUDA threads execute on a physically separate device (GPU), which is a co-processor to the host (CPU) that runs the program. CUDA also assumes that the host and device both have separate memory spaces: host memory and device memory, respectively. Because host and device both have their own separate memory spaces, there is potentially a lot of memory allocation, de-allocation and data transfer between host and device. Thus, memory management is a key issue in GPGPU computing. Inefficient use of memory can significantly increase the computation time and mask the speed-ups obtained by the data calculations.

3.1.3 Algorithms

There are many classifications of parallel algorithms. According to basic calculation objects, the algorithms can be divided into numerical and non-numerical [65]. FDTD is the study of numerical parallel algorithms, whereas comparing relationships between symbolic processing issues is a non-numerical parallel algorithm, like database operations [66].

The dependencies of parallel processes can be divided into synchronous algorithm, asynchronous algorithm, and independent algorithm. In a synchronous parallel algorithm, the performance of some computation unit tasks requires others' results. Thus, it needs a global clock to control the pace of each part so that each computing unit in various parts of the tasks can simultaneously move forward. In contrast, an asynchronous parallel algorithm does not need every computing unit to wait for the others. They either decide to wait, continue on, or terminate, based on different stages of the calculating process. In an independent parallel algorithm, all computing units are totally irrelevant and do not even start computing tasks simultaneously. For distributed computing, people often use the independent parallel algorithm.

If we focus on the computing tasks, the algorithms can be divided into coarse-grained parallel ones and fine-grained parallel ones [67]. The former is an overall task decomposition into each computing unit, and each unit has a substantial workload of computing tasks and a complex calculation procedure, e.g., the parallel algorithm based on PC clusters. In terms of design, by comparison, a fine-grained parallel algorithm does not care about the relationship between computing tasks and computing units; it is only responsible for assigning the data calculation process to the abstract threads. Single-GPU-based FDTD is a fine-grained parallel algorithm, while multi-GPU-platform FDTD contains a coarse-grained parallel algorithm.

3.2 COMPUTATION: FROM CPU TO GPU

Benchmarks that focus on floating point arithmetic (which is most often used in these engineering computations) show that GPUs can perform such computations much faster than the traditional CPUs used in today's workstations – sometimes as much as several times and could even be several orders times faster at the best performance, depending on the specific computation.

3.2.1 GPU Computation Capability

The number of floating points operations per second (flops) of a computer is one of the measures of the device's computational abilities. This is an important measure, especially in scientific calculations, as it is an indication of a computer's arithmetic capabilities.

While a high-performance CPU can have a double-precision computation capability of 140 Gflops (Intel Nehalem architecture) [63], an NVIDIA Tesla 20-series (NVIDIA Fermi architecture) GPU has a peak single-precision performance of 1.03 Tflops and a peak double-precision performance of 515 Gflops [64]. Furthermore, the computation capability of a GPU is growing at a much faster pace compared to the CPU.

Although the compute capability of a GPU is impressive when compared to a CPU, it has one significant disadvantage in scientific applications. Not all GPUs fully conform to the IEEE standard for floating point operations [68]. Although the floating point arithmetic of NVIDIA graphics cards is similar to the IEEE 754-2008 standard used by many CPU vendors, it is not quite the same, especially for double precision [61].

3.2.2 Accurate Floating-Point Representation

In computers, the natural form of representation of numbers is in binary (1's and 0's). Thus, computers cannot accurately represent real numbers. There are standards to represent floating-point numbers in computers, the most widely used of which is the IEEE 754 standard. Accuracy in representation of floating point numbers in computers is important in scientific applications.

There have been many cases where errors in floating-point representation have caused catastrophes. One example is the failure of the American Patriot Missile defence system to intercept an incoming Iraqi Scud missile at Dhahran, Saudi Arabia in 1991. This resulted in the death of 28 Americans [69]. The cause of this was determined to be the loss of accuracy from the conversion of an integer to a real number in the Patriot's computer. Other examples of catastrophes resulting from floating-point representation errors can be found at [70].

Therefore, accurate floating-point representation is important in scientific computations. Most CPU manufacturers now use the IEEE 754 floating-point standard. As developments in GPGPUs continue, GPU vendors will inevitably conform to the IEEE 754 standard for floating-point representation as well. This is evident with the newest Fermi architecture from NVIDIA, which implements the IEEE 754-2008 floating point for both single- and double-precision arithmetic [62].

3.3 GPU Memory Structure

The CUDA memory hierarchy is shown in Figure 3.3.1. These varied types of memory differ in size, access times, and restrictions. Detailed descriptions of the various memory types are available in the CUDA Programming Guide [61] and the CUDA Best Practices Guide [71].

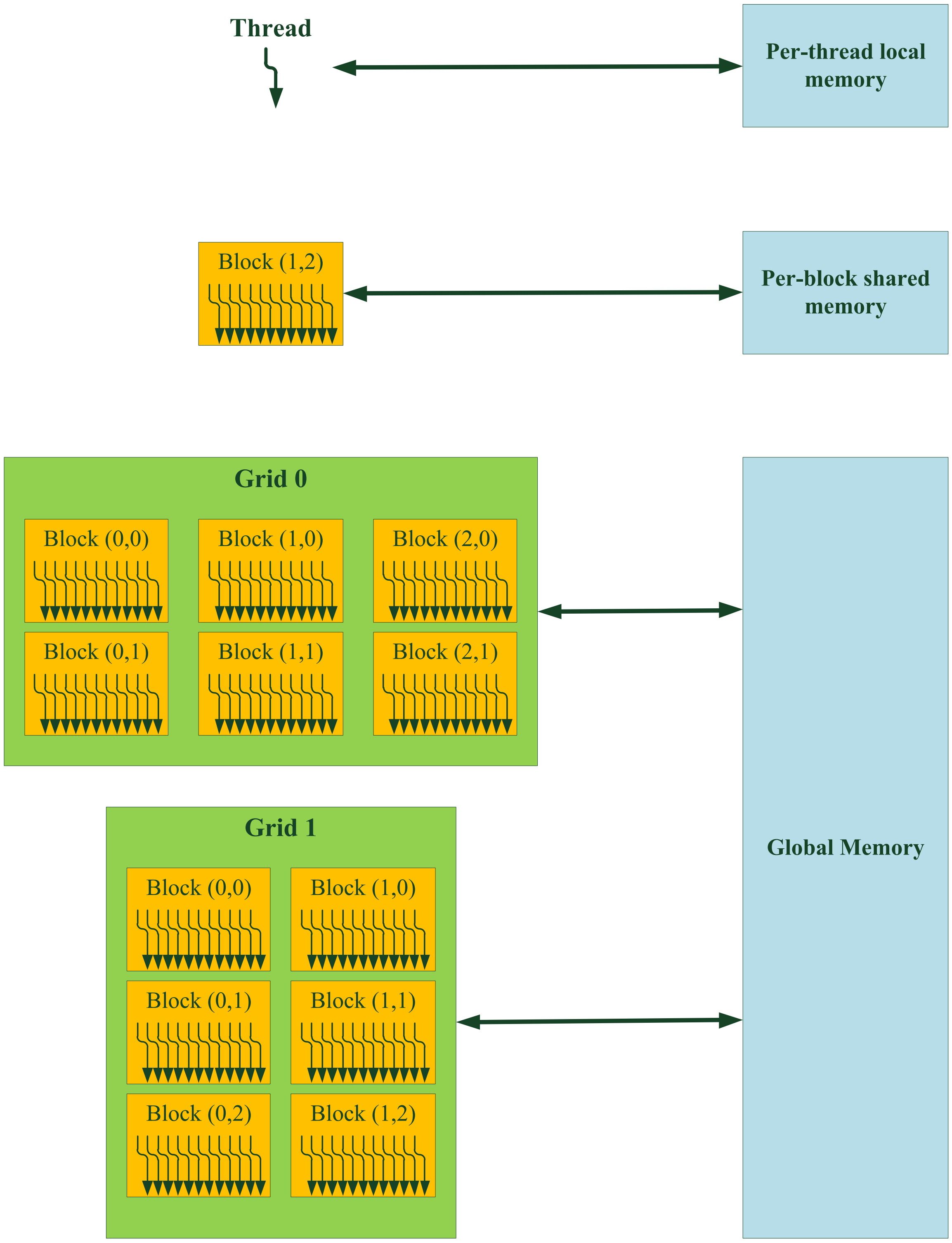


Figure 3.3.1: Hierarchy of various types of memory in CUDA.

(Source: NVIDIA CUDA C programming guide version 4.2, 2012.)

In short, global memory is the largest in size. It is located off the GPU chip and can be accessed by any thread. Because it is off-chip, its access time is the slowest amongst all types of memory. Shared memory is located on-chip, which makes memory access very fast compared to global memory. However, it is limited in size and shared memory access is only on block-level. This means that a thread cannot access shared memory that is allocated outside its block. Local memory is located off-chip and thus has a long latency. It is used to store variables when there are insufficient registers available.

Other types of memory not shown in Figure 3.3.1 are registers and constant memory. Registers are located on-chip, are scarce, and are not shared between threads. Constant memory is located off-chip but is cached. Caching makes memory accesses to constant memory fast, although it is located off-chip.

While global memory is located off-chip and has the longest latency, there are techniques available that can reduce the amount of GPU clock cycles required to access large amounts of memory at one time. This can be done through memory coalescing, which refers to the alignment of threads and memory. For example, if memory access is coalesced, it takes only one memory request to read 64-bytes of data. On the other hand, if it is not coalesced, it could take up to 16 memory requests, depending on the GPU's compute capability. This is further explained in the CUDA Best Practices Guide [71].

3.4 SUMMARY

GPUs have a parallel architecture with the capability of executing thousands of threads simultaneously. This gives the GPU an advantage over the CPU when it comes to intensive computations on large amounts of data. With the CUDA framework, developers have access to CUDA-enabled NVIDIA GPUs. This allows developers to leverage this computation capability for applications other than graphics-rendering. Along with this, CUDA provides a relatively cheap alternative to supercomputing.

CHAPTER 4 IMPLEMENTATIONS OF CUDA-ENABLED FDTD

4.1 COMPUTE UNIFIED DEVICE ARCHITECTURE

4.1.1 Hardware

The CUDA model is developed by NVIDIA from its GPU parallel programming development. Currently, it is only supported by the production of NVIDIA GPU G80 and above, including G92, GT200, GF100, GF110 of the Fermi series, etc. The market-oriented Geforce series, the graphics workstation-oriented Quadro series, and the scientific computing-oriented Tesla series graphics cards have all supported CUDA programming development. From G80, GPU began to use unified stream processors to replace the original discrete vertex shaders and pixel shaders. In this thesis, we will use GT520 and GT650 to analyze CUDA architecture and performance.

Within GT520 GPU architecture, GPU is made up of TPCs (Texture Processor Clusters). Three streaming multiprocessors (SM) are most relevant with general-purpose computing. An SM is single-instruction multiple-thread processor architecture. Similar to SIMD, it contains command transmitting units, eight streaming processors or scalar processors, two special function processors (SFP), quick-accessible shared memory, and command and constant caches.

Arithmetic operation is basically done within SPs, which contains floating-point and integer processing units and registers. Floating-point FMAD/FMUL/FADD and integer ADD/MUL/CMP/MOVE can thus be performed.

There are 30 SMs and 240 SPs in GT520. The operation frequency of SP within GTX280 graphics card settings is 1296MHz. Through its dual-issue commanding emission mechanism, there can be a floating-point multiply-add operation and a multiply operation within each clock cycle, so its theoretical floating point computing capability is up to

1296MHz×240×3Flops=933120MFlops=933.12GFlops, which indicates the amount of floating point operations per second. In comparison, the floating point operation capability of same-period high-end Intel Harpertown architecture 3.2GHz CPU is just 102GFlops. GPU simplifies the control unit and the cache function, further focusing on the floating point computing power, and is therefore more suitable for intensive computations.

The calculation process requires frequent access to the dynamic random access memory (DRAM), namely PC memory and graphics memory. Therefore, the calculation is closely related to another hardware issue: memory bandwidth. For instance, there are 8 internal storage controllers built in GT200 GPU architecture, and each can control two memory chips of 32 bits, so there are 16 GDDR3 memory chips with 32-bit width and 64MB capacity in total. This gives us a 512-bit overall storage, 1GB capacity, and 2214MHz operating frequency. It can be calculated that the GTX280 graphics memory bandwidth is 512bits×2214MHz/8=141696MB/s=141.696GB/s, while the widespread configuration of personal computers was dual channel DDR2 800MHz memory with the bandwidth of 12.8GB/s during the same period.

The CUDA-supported GPUs are divided into several versions with different computing capabilities based on their fundamental structures. This computing power, however, does not represent the speed of floating-point operations or capacity but the support of some CUDA features. For example, the graphic card GTX280 is under the architecture of GT200, and its computing capability is 1.3. Additional characteristics of computing power versions can be found in [61]. GT520 and GT650 graphic cards' specifications are shown in Table 4.1.1.1.

	GeForce GT 520	**GeForce GTX 650**
	GPU Core Details	
Core Speed	810 MHz	1058 MHz
Architecture	Fermi GF119	Kepler GK107

Notebook GPU	No	No
Release Date	14 May 2011	01 Sep 2012
	GPU Memory	
Memory	1024 MB	2048 MB
Memory Speed	900 MHz	1250 MHz
Memory Bus	64 Bit	128 Bit
Memory Type	DDR3	GDDR5
Memory Bandwidth	14.4 GB/sec	80 GB/sec
	GPU Display	
Shader Processing Units	48	384
Actual Shader Performance	96	384
Technology	40 nm	28 nm
Texture Mapping Units	8	32
Texture Rate	6.5 GTexel/s	33.9 GTexel/s
Render Output Units	4	16
Pixel Rate	3.2 GPixel/s	16.9 GPixel/s
	GPU Display Tech	
DirectX	11	11.1
Shader Model	5.0	5.0
Open GL	4.1	4.3
	GPU Power Requirements	
Max Power	29 Watts	64 Watts

Table 4.1.1.1: GeForce GT 520 and GTX 650 graphics cards brief information.

4.1.2 Programming Model

The development using the CUDA parallel programming no longer needs the aid of graphics API. Its syntax is just an extension of C language, following the C language programming structure and basic grammar. Scientific computing researchers who are familiar with the C language can quickly transit to GPU programming.

In a CUDA parallel programming model, it is assumed that the parallel program is executed on independent physical facilities, which are called Device. The facilities are treated as coprocessors of the Host, which runs the C code. In CUDA, each block of threads can be scheduled on any of the available multiprocessors within a GPU, in any order, concurrently or sequentially, so that a compiled CUDA program can execute on any number of multiprocessors, as illustrated by Figure 3.1.2.1. Only the runtime system needs to know the physical multiprocessor count [61].

CUDA C extends C by allowing the programmer to define C functions, called kernels. When called, they are executed N times in parallel by N different CUDA threads, as opposed to only once like regular C functions. The declaration specifier of __global__ is used to define a kernel, and the number of CUDA threads that execute the kernel for a given call is specified using a new <<<…>>> execution configuration syntax. Each thread that executes the kernel is given a unique thread ID. That ID is accessible within the kernel through the built-in threadIdx variable.

The following code shows how it works:

```
#include <cuda_runtime.h>

// Kernel definition

__global__ void addMatrix(float* A, float* B, float* C){
     int i = threadIdx.x;
     C[i] = A[i] + B[i];
}

int main() {

     ...
```

```
    // Kernel invocation with THREAD_NUM threads
    addMatrix<<<1, THREAD_NUM>>>(A, B, C);
    ...
}
```

Listing 4.1.2.1: Sample code adding two matrices A and B within CUDA.

The CUDA programming hierarchy can be seen in Figure 4.1.2.1. Multiple threads will appear as one-, two-, or three-dimensional groups to make up a block. Respectively, a one-, two-, or three-dimensional thread block will be formed. This provides a natural way to invoke computation across the elements in a domain such as a vector, matrix, or volume.

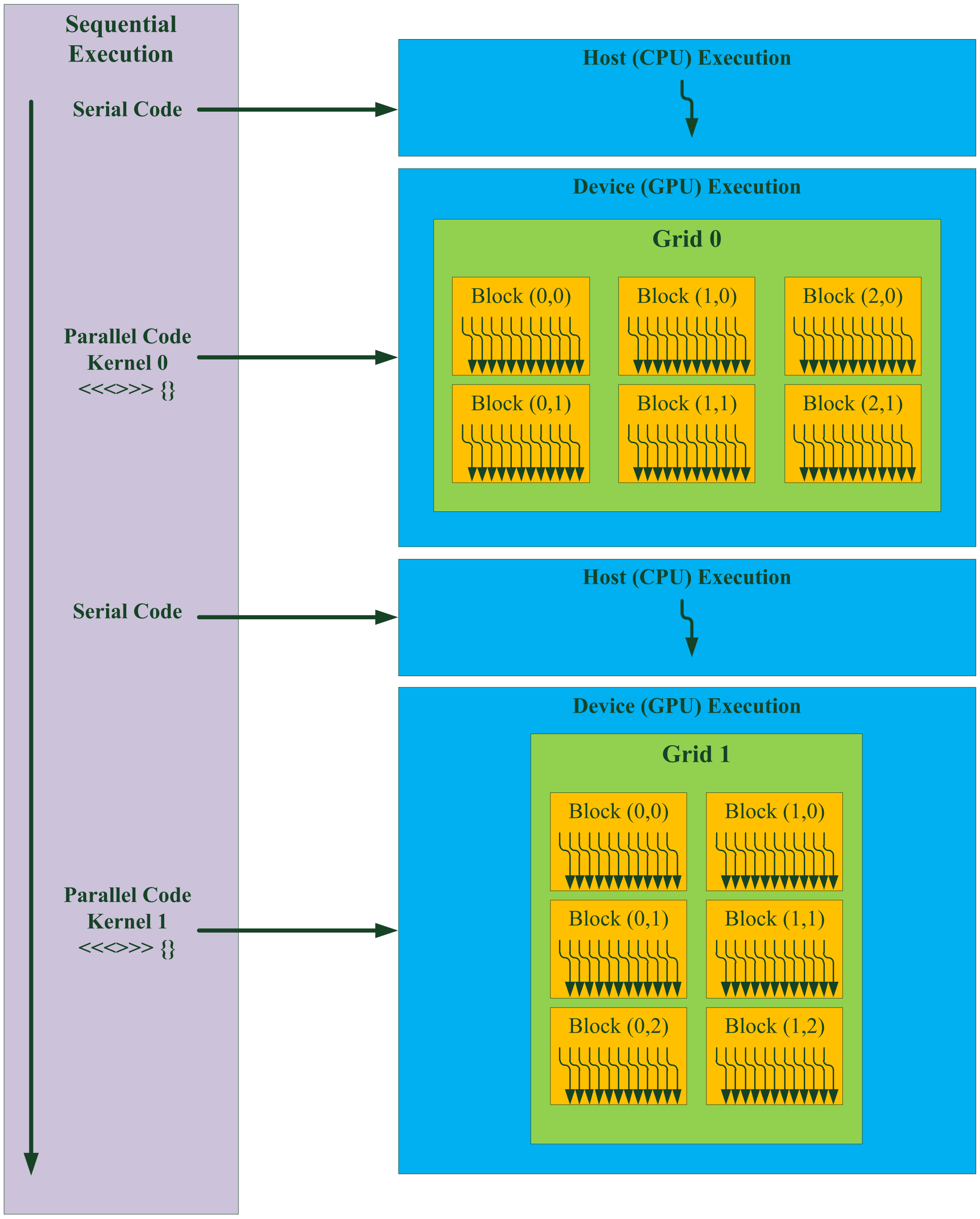


Figure 4.1.2.1: CUDA programming hierarchy.

(Source: NVIDIA CUDA C programming guide version 4.2, 2012.)

The index of a thread and its thread ID relate to each other in a straightforward way: For a one-dimensional block, they are the same; for a two-dimensional block of size (Dx, Dy), the thread ID of a thread of index (x, y) is (x + y Dx); for a three-dimensional block of size (Dx, Dy, Dz), the thread ID of a thread of index (x, y, z) is (x + y Dx + z Dx Dy). We can see an example of this from Listing 4.1.2.2.

```
__global__ void calculate2D(double**A, double**B,
double**C){

     int j=blockIdx.x*blockDim.x+threadIdx.x;
     int k=blockIdx.y*blockDim.y+threadIdx.y;

     if (j<MAX_VALUE && k<MAX_VALUE){
          A[j][k]= B[j][k] + c0*dt/(2*dy)*(C[j][k]-C[j][k-
1]);
     }

     ...

}

int main(){

     // Kernel invocation with one block of THREAD_NUM *
THREAD_NUM * BLOCK_NUM threads

     ...

     calculate2D<<<BLOCK_NUM, THREAD_NUM, THREAD_NUM *
sizeof(double)>>>(A,B,C);
}
```

Listing 4.1.2.2: Sample code: 2D matrix calculation within CUDA.

A kernel can be executed by multiple, equally-shaped thread blocks, so that the total number of threads is equal to the number of threads per block times the number of blocks. The number of thread blocks in a grid is usually dictated by the size of the data being processed or the number of processors in the system, which it can greatly exceed. Blocks can be organized into a one-dimensional, two-dimensional, or three-dimensional grid of thread blocks, as illustrated by Figure 4.1.2.2.

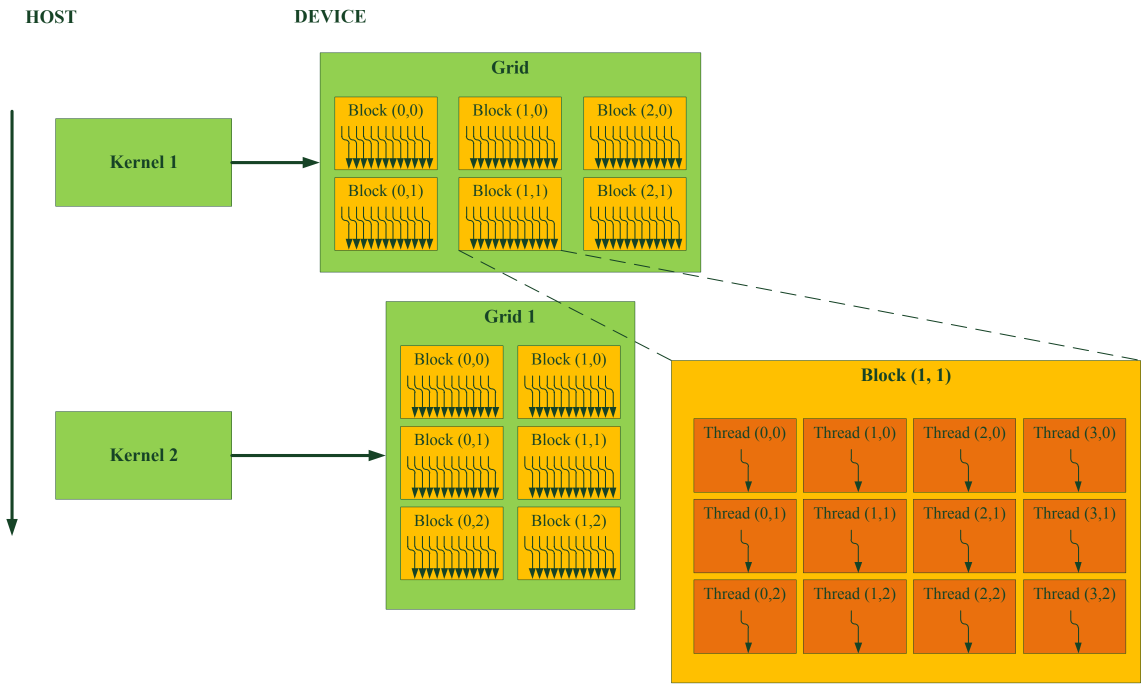


Figure 4.1.2.2: CUDA grid of thread blocks.

(Source: NVIDIA CUDA C programming guide version 4.2, 2012.)

Thread blocks need to execute independently, and it must be possible to execute them in any order, in parallel or in series. The requirement of independence allows thread blocks to be scheduled in any order, across any number of cores, as shown in Figure 4.1.2.1. This enables programmers to write code that scales with the number of cores. Threads within a block can share data through some shared memory and synchronize their execution to coordinate memory accesses. In this way, they cooperate well. More precisely, we can specify synchronization points in the kernel by calling the __syncthreads() intrinsic function, which acts as a barrier at which all threads in the block must wait before any is allowed to proceed. Researchers simply need to decompose the task to the abstract threads, so the program has very good scalability.

CUDA threads may access data from multiple memory spaces during their execution, as illustrated by Figure 3.3.1. Each thread has private local memory and each block has shared memory visible to all threads of the block and with the same lifetime as the block.

All threads have access to the same global memory. There are also two additional read-only memory spaces accessible by all threads, known as constant and texture memory spaces. Global, constant, and texture memory spaces are optimized for different memory usages. Texture memory also offers different addressing modes, as well as data filtering for some specific data formats.

4.1.3 Software Architecture

The general software hierarchy of CUDA is shown in Figure 4.1.3.1. Developers can program based on three CUDA APIs. The highest level is two mathematical function libraries encapsulated in a CUDA toolkit: CUFFT (CU Fast Fourier Transform) and CUBLAS (CU Basic Linear Algebra Subroutines). Developers can use these two math libraries to quickly implement many common programming algorithms. In consideration of the versatility, however, they may not be the optimal choice in terms of execution speed.

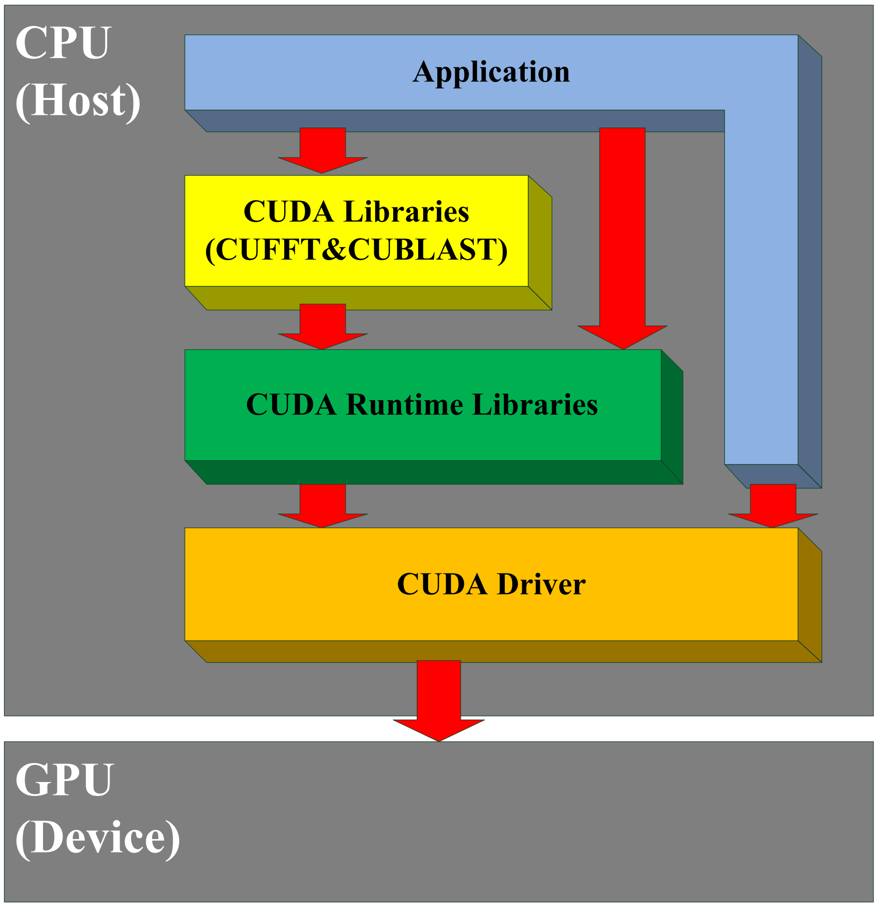


Figure 4.1.3.1: CUDA software stack.
(Source: http://www.viznet.ac.uk/reports/gpu/7)

Lower than that, there is CUDA Runtime library, which is provided by CUDART dynamic library. All of its APIs have a "cuda" prefix. Runtime offers device management, memory management, workflow management, execution control, texture reference management, and application interface for interoperability with OpenGL and Direct3D, which is the encapsulation based on CUDA Driver library with no special initialization functions. It is done automatically at the first call of Runtime functions.

CUDA Driver API is a handle-based underlying interface provided by NVCUDA dynamic library. All of its APIs have a "cu" prefix. In addition to containing all of the features of Runtime API, Driver API also offers Context management and module management functions. Before calling other Driver API functions, cuInit (unsigned int

Flags) has to be called to initialize everything, create a CUDA context, and associate with the specified devices in the system. Using Driver API to program requires a large amount of code and complex debugging, but in this way some more complicated functions can be implemented through hardware operation. CUDA stipulates that Runtime API and Driver API are mutually exclusive, which means that in any one application, there can only be one of them. Mixed utilization is not allowed.

CUDA source code is a file with the extension .cu. When using NVCC (NVIDIA CUDA Compiler) to compile, it first separates the code within the source file from the host and device, i.e., the serial and parallel codes demonstrated in Figure 4.1.2.3, and then calls different compilers to compile separately. Device-side code is compiled into PTX (Parallel Thread eXecution) by NVCC, from which a binary cubin file will be generated. Next, they are integrated into the generated code on the host side and an executable file is built. More specific compiling process can be found in [72].

CUDA has some expansion for C language. Along with a few built-in variables mentioned previously, there is warpSize (current value 32), which represents the size of a warp block, and function-type qualifiers. Specifying the function is executed on the host or device and can be invoked through the host or device. Variable type qualifiers specify the location of a variable in the device memory. A new directive specifies how to execute core instructions on the device through the host.

Function type qualifiers include __device__, __global__ and __host__. The former two perform on the device, and __host__ is akin to normal C functions. __global__ can only be invoked by the host, while __device__ can only be called via the device. So, if we want to conduct a computation using a device in a CUDA program, there has to be a function of __global__.

Variable type qualifiers include __device__, __constant__, and __shared__, to declare variables located in the device, in the constant memory space, and in the shared memory space, respectively.

These extensions have some restrictions. For example, __device__ and __global__ functions do not support recursion; static variables cannot be declared within the function body; the number of function parameters cannot be changed; the return type of a __global__ function must be void; there is a 256-byte limitation when passing __global__ function parameters to the device through the shared memory; and the declaration of __shared__ variables cannot contain any initialization. These restrictions are from lower versions of CUDA. With the evolving of CUDA and its hardware, higher versions can already implement certain features of C++ on HPC devices.

Apart from the extensions of C language, there are also some software components. These include some packaged vector types such as int2, int3, int4, float2, and float3. They all inherit from the basic integer or floating point types and are structures whose components can be accessed via x, y, z, and w. Additionally, they are mathematical functions and versions with "__" prefix, and they have texture reference declaration and its fetch function. The __syncthreads() function synchronizes all threads within a block.

Even though CUDA-based software development has been outside the range of graphic API, an in-depth understanding of hardware and software architectures of CUDA is necessary for improving the efficiency of parallel programs. It is also conducive for efficient parallel algorithm design of CUDA models.

Compared with conventional development of GPU computing approaches, CUDA programming is relatively simple, and there are even more diverse functions that can be realized. This provides an easier way to achieve high-efficiency GPU parallel computation. In addition, performance of hardware that supports CUDA is also being enhanced, and the cost is lower than other devices in terms of general-purpose computing. CUDA has been used since its release in many fields of scientific computing, and many commonly used algorithms have achieved the goal of GPU parallelization. These include: the finite element method [73], the Monte Carlo [74] [75] algorithm, the sequence comparison of the Smith Waterman [76] algorithm, the molecular dynamics [77] simulation, method of moments [78], etc.

4.2 FDTD Parallel Computing Method Based on CUDA

In a traditional serial FDTD algorithm, we need to perform by-point iteration at each step in the time-domain promoting process. Programmers must use multiple FDTD loops to deal with the entire region. CUDA-based FDTD parallel algorithm is a method to parallelize the computation of such order. CUDA abstract threads operate in parallel iteration of one or a few grid points, making full use of hundreds of GPU computing units. The basic process is shown in Figure 4.2.1.

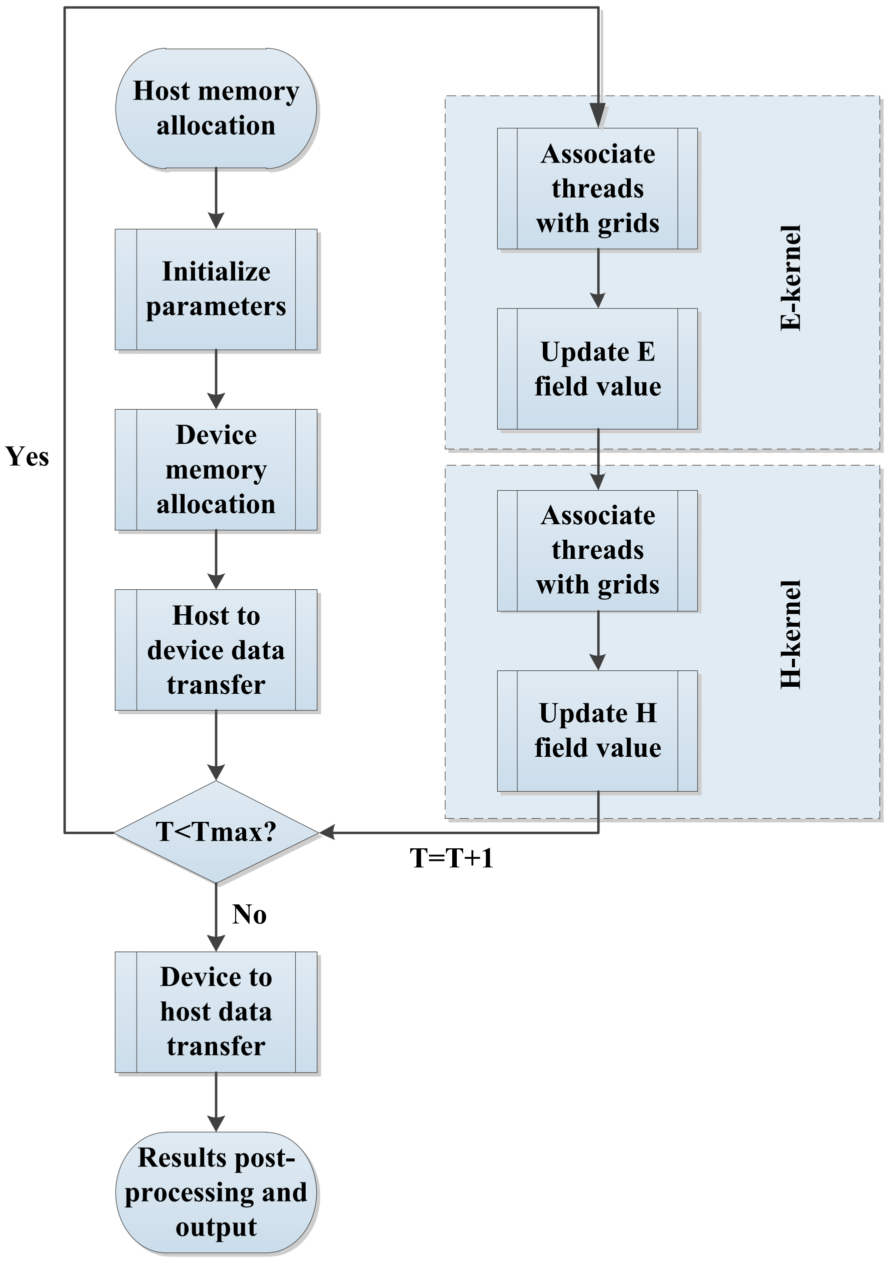

Host memory allocation
Initialize parameters
Device memory allocation
Host to device data transfer
T<Tmax?
Yes
No
Device to host data transfer
Results post-processing and output
Associate threads with grids
Update E field value
E-kernel
Associate threads with grids
Update H field value
H-kernel
T=T+1

Figure 4.2.1: CUDA parallel algorithm flowchart of FDTD.

In the figure above, memory allocation, initialization of field values and parameters, and results post-processing are all one-time operations on Host (which is the CPU side). Graphics memory allocation and memory copying steps are operations of Device (which is the GPU side). Only those in the dashed box, E-kernel and H-kernel, which are the most time-consuming part of FDTD calculation, are performed by the GPU computing units. If there is a long-time occupation of GPU computing units by CUDA, the Windows operating system will determine that the display device does not respond. For Windows, running time of a single GPU program (i.e., a kernel) is about five seconds, so we will use CPU to take care of time-step iteration and cyclically call GPU computing cores to achieve the goal of time-domain iteration. Because each block in CUDA cannot perform synchronous operation, and in FDTD computation we need the adjacent grid point half time-step **H** (**E**) field values to calculate the next during **E** (**H**) recursion, the parallel computation is divided into two kernels for global synchronization to ensure that the required field values within the iteration have all been updated.

As can be seen, compared with the serial algorithm, even though GPU parallel computing brings in device initialization and data transmission, these steps require only one-time operation. FDTD iteration of time domain typically takes thousands of times, so these "extra" processes will not affect the efficiency of GPU parallel computing.

The most important part of GPU parallel computing is to associate the calculation with the abstract threads. In other words, we should properly and reasonably arrange the calculation upon current threads, and make full use of advantages of every storage unit to realize the optimization.

4.3 TWO-DIMENSIONAL CUDA-ENABLED FDTD

4.3.1 Threads Arrangement

In this thesis, we will use one thread to deal with one Yee cell in order to implement parallel computing, as shown in Figure 4.3.1.1. In the figure, Block (x, y) is the block index and t (x, y) is the thread index. [i] [j] within the Ex zone is the coordinate index of the Yee grid, and the arrow indicates a point taken by a thread.

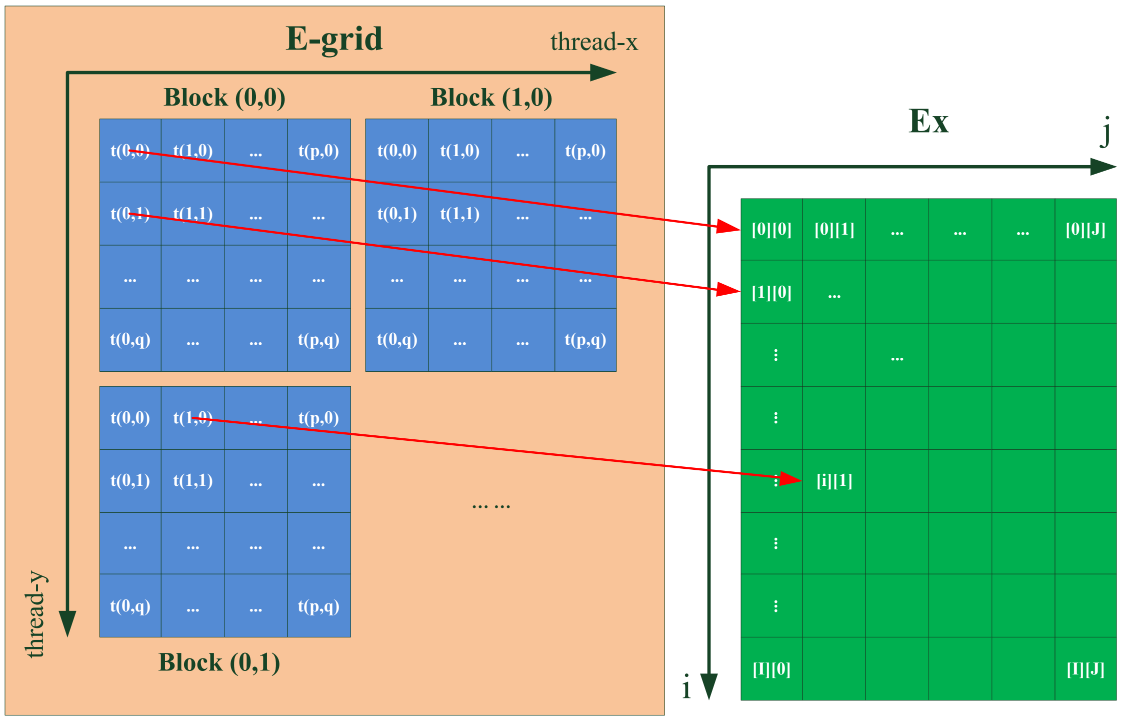


Figure 4.3.1.1: 2D FDTD parallel algorithm thread arrangement.

As can be seen from the figure, both the grid and the block we use are two-dimensional structures, with each block consisting of (p + 1) × (q + 1) threads. Due to the manner of one thread corresponding to one grid cell, to solve a two-dimensional (i + 1) × (j + 1) size problem, the required block number is ((i + 1) / (m + 1)) × ((n + 1) / (j + 1)). When the block number is not an integer, it should be rounded up for its ceiling.

Figure 4.3.1.1 shows that the allocated memory in the proposed algorithm is continuous in the j-direction. In CUDA, priority is given to threadIdx.x. Therefore, we can map the x-direction of threadIdx and blockIdx (i.e., thread.x in Figure 4.3.1.1) to the j-direction of Ex. This thread scheduling can meet the requirement of global memory coalesced conditions, and improve the effective bandwidth of memory access.

In two-dimensional problems, we use the manner of one-to-one to organize parallel threads. If the position of a grid point processed by a current thread is determined, we can simply use recursion formulas to achieve an FDTD algorithm. Given, as described previously, that field values are all assigned within the global memory, one global memory access often requires a lot of clock cycles. Although more active threads in arithmetic can alleviate some memory access latency, to achieve higher parallel efficiency, full use of other types of memory is necessary.

4.3.2 Implementation Algorithms and Approaches

The finite-difference time-domain implementation running on CUDA was tested on a computer whose specifications are shown in Table 4.3.2.1.

Operating System	64-bit Windows 7 Home Premium
Memory (RAM)	10 GB
CPU	Dual Intel i7-2600 Frequency: 3.40 GHz Number of Cores: 4
GPU	NVIDIA GeForce GT 520 Graphics Clock: 810 MHz Processor Clock: 1620 MHz CUDA Cores: 48

Table 4.3.2.1: Specifications of the PC testing platform.

The NVIDIA GT520 has a CUDA Compute Capability of 2.1. Detailed specifications of the GT520 are listed in Appendix A. As shown in Table 4.3.2.1, there are 4 processors in it. However, for the purpose of this thesis, only one of the processors is utilized.

Some example code for the two-dimensional FDTD method's main field update equations is shown in Listing 4.3.2.1.

```
QHz[iJ][iK]= Z0*Hz[iJ][iK] - c0*dt/(2*dx)*(QEy[iJ+1][iK]-
QEy[iJ][iK]);
```

```
HzN[iJ][iK]= 1/Z0*QHz[iJ][iK] +
dt/(2*mu0*dy)*(QEx[iJ][iK+1]-QEx[iJ][iK]);

…

ExN[iJ][iK]= QEx[iJ][iK] + c0*dt/(2*dy)*(QHz[iJ][iK]-
QHz[iJ][iK-1]);
```

Listing 4.3.2.1: Main update equations example for 2D FDTD.

After initialization, all necessary data are transferred from the CPU to the GPU's global memory. Then, the CUDA kernel that contains the update equations of Listing 4.3.2.1 is executed. To compare execution time, CUDA's timer functions are utilized. Only the time taken to run the main loop of the FDTD update equations is recorded.

The CUDA code has to be executed twice for the sake of analyzing both the accuracy and the execution time between CPU and GPU. This is because GPU works with the data stored in its global memory and the data has to be transferred to the CPU before it can be analyzed. Then the CUDA code is executed again, this time without memory transfers, which will provide a more accurate and fair comparison against the CPU's execution time.

We can see the differences between traditional C code working on CPU and CUDA C working on GPU from Listing 4.3.2.2 and Listing 4.3.2.3 below.

```
for(i = 0; i < iSXmax - 1; i++)
{
     for(j = 1; j < iSYmax - 1; j++)
     {
     h_pdExN[i * iSYmax + j] = h_pdQEx[i * iSYmax + j] +
dc0 * dt / (2.0 * dy) * (h_pdQHz[i * (iSYmax - 1) + j] -
h_pdQHz[i * (iSYmax - 1) + j - 1]);

     }
}
```

Listing 4.3.2.2: Main update loop within CPU.

```
cudaMemcpy2D(d_pdQEy, szPitchQEy, h_pdQEy2, (iSYmax - 1) *
sizeof(double), (iSYmax - 1) * sizeof(double), iSXmax,
cudaMemcpyHostToDevice);

GPU_FDTD_Kerenl1<<<grids, threads>>>(d_pdHz,
               szPitchHz / sizeof(double),
               d_pdQEx,
               szPitchQEx / sizeof(double),
               d_pdQEy,
               szPitchQEy / sizeof(double),
               d_pdQHz,
               szPitchQHz / sizeof(double),
               d_pdHzN,
               szPitchHzN / sizeof(double),
               iSYmax,
               iSXmax,
               dZ0,
               dc0,
               dt,
               dx,
               dy,
               dmu0,
               dSource,
               iis,
               ijs
               );
```

Listing 4.3.2.3: Main update loop within GPU.

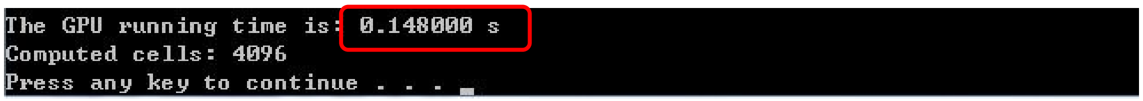
The listing below illustrates code for CUDA kernel:

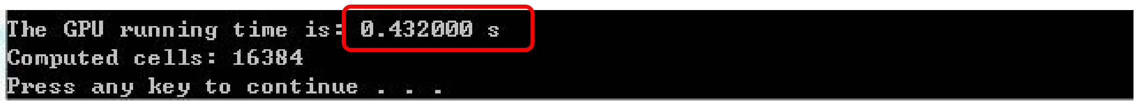
```
__global__ void GPU_FDTD_Kerenl1(double const *d_pdHz,
size_t const szPitchHz, double const *d_pdQEx, size_t const
szPitchQEx, double const *d_pdQEy,
     size_t   const   szPitchQEy,
     double          *d_pdQHz,
     size_t   const   szPitchQHz,
     double          *d_pdHzN,
     size_t   const   szPitchHzN,
     unsigned int const iSYmax,
     unsigned int const iSXmax,
     double   const   dZ0,
     double   const   dc0,
     double   const   dt,
     double   const   dx,
     double   const   dy,
     double   const   dmu0,
     double   const   dSource,
     int      const   iis,
     int      const   ijs
     )
{
     unsigned int uiX = blockIdx.x * blockDim.x +
threadIdx.x;
     unsigned int uiY = blockIdx.y * blockDim.y +
threadIdx.y;

     unsigned int uiHzId  = uiY * szPitchHz + uiX;
     unsigned int uiQExId = uiY * szPitchQEx + uiX;
     unsigned int uiQEyId = uiY * szPitchQEy + uiX;
     unsigned int uiQHzId = uiY * szPitchQHz + uiX;
     unsigned int uiHzNId = uiY * szPitchHzN + uiX;

     if(uiX < iSYmax - 1 && uiY < iSXmax - 1)
     {
          d_pdQHz[uiQHzId] = dZ0 * d_pdHz[uiHzId] - dc0 *
dt / (2.0 * dx) * (d_pdQEy[(uiY + 1) * szPitchQEy + uiX] -
d_pdQEy[uiQEyId]);

          d_pdHzN[uiHzNId] = 1.0 / dZ0 * d_pdQHz[uiQHzId] +
dt / (2.0 * dmu0 * dy) * (d_pdQEx[uiQExId + 1] -
d_pdQEx[uiQExId]);
          if(uiX + 1 == iis && uiY + 1 == ijs)
          {
               d_pdHzN[uiHzNId] = d_pdHzN[uiHzNId] -  dt /
dmu0 * dSource;
          }
     }
}
```

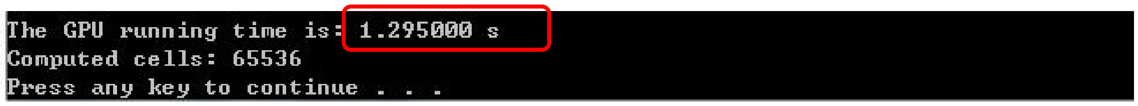
Listing 4.3.2.4: Example: CUDA kernel code for FDTD.

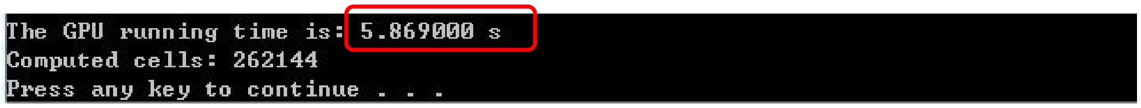
Undoubtedly, acceleration can be achieved and will be discussed in the next chapter. Here, we focus only on the initial implementation of CUDA-enable FDTD. There are a few interesting observations that can be made from the results. First, as expected, higher speed-ups are obtained when the simulation size is increased. This speed-up will be more appreciable as simulation size is increased and FDTD is done in three dimensions instead of only two. A simulation that will take hours to run on CPU could potentially take only minutes to run on a GPU. Also, the results show that for small simulation sizes such as 100 cells, CPU runs much faster than GPU. This is a result of the latency of memory transfers between host (CPU) and device (GPU), as discussed in Chapter 3. Because both host and device have separate memory spaces, data has to be transferred between host and device in order for the GPU to perform the calculations. However, when the simulation size is small, the fast speed of the GPU in arithmetic operations is affected by the time taken to transfer the data. When the size is smaller than the total number of available devices, the computing capability of the unused devices are wasted during a particular calculation cycle. We will keep on discussing these in Chapter 5.

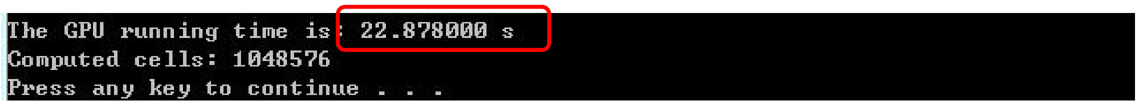
Some of the initial run time results can be seen from Figure 4.3.2.1 to Figure 4.3.2.6 with 100 FDTD time steps:


```
The GPU running time is: 0.148000 s
Computed cells: 4096
Press any key to continue . . .
```
Figure 4.3.2.1: GPU initial run result: simulation size of 64×64 with 100 time steps.


```
The GPU running time is: 0.432000 s
Computed cells: 16384
Press any key to continue . . .
```
Figure 4.3.2.2: GPU initial run result: simulation size of 128×128 with 100 time steps.


```
The GPU running time is: 1.295000 s
Computed cells: 65536
Press any key to continue . . .
```
Figure 4.3.2.3: GPU initial run result: simulation size of 256×256 with 100 time steps.


```
The GPU running time is: 5.869000 s
Computed cells: 262144
Press any key to continue . . .
```
Figure 4.3.2.4: GPU initial run result: simulation size of 512×512 with 100 time steps.



```
The GPU running time is: 22.878000 s
Computed cells: 1048576
Press any key to continue . . .
```

Figure 4.3.2.5: GPU initial run result: simulation size of 1024×1024 with 100 time steps.

More detailed speeding-up strategies and results will be thoroughly discussed in Chapter 5, where comparisons will be given generally and separately.

4.4 THREE-DIMENSIONAL CUDA-ENABLED FDTD

4.4.1 Threads Arrangement

CUDA stipulates that grid structure can only be one-dimensional or two-dimensional, so dealing with three-dimensional FDTD problems cannot be done as simply as two-dimensional correspondence. We can use two approaches for three-dimensional FDTD parallel algorithm threads organization.

One approach is using a thread to correspond to the whole line in the direction of i, called TFL (Thread For Line) mode. Each thread sequentially calculates every point along i-direction on the line. In this thesis, field values are indexed following the manner of [i][j][k]. In k-direction memory is continuous, and the greatest skip of memory change happens in i-direction. TFL total number of threads is equal to the number of three-dimensional Yee grids on the jk-plane. This approach is relatively simple, but the thread computing task is more complex.

The other approach, however, is using a thread to correspond to a single point, called TFP (Thread For Point) mode. Each recursion of jk-plane composed by Yee grids is just like two-dimensional FDTD algorithm, taken by a CUDA thread of a two-dimensional region. Each thread corresponds to one of the points, and the threads pairing manner of each plane is shown in Figure 4.3.1.1. The total thread number of TFP mode is the same as the number of three-dimensional Yee grids. The calculation process can be conveniently implemented after the pairing relationship is established.

4.4.2 Implemented Algorithms and Approaches

The three-dimensional FDTD method has six main equations, whose conversion into computer programming is shown in Listing 4.4.2.1 below.

```
ex[i][j][k] = ca[id] * ex[i][j][k] + cby[id] * (hz[i][j][k]
- hz[i][j-1][k]) - cbz[id] * (hy[i][j][k] - hy[i][j][k-1]);

ey[i][j][k] = ca[id] * ey[i][j][k] + cbz[id] * (hx[i][j][k]
- hx[i][j][k-1]) - cbx[id] * (hz[i][j][k] - hz[i-1][j][k]);

ez[i][j][k] = ca[id] * ez[i][j][k] + cbx[id] * (hy[i][j][k]
- hy[i-1][j][k]) - cby[id] * (hx[i][j][k] - hx[i][j-1][k]);

hx[i][j][k] = da[id] * hx[i][j][k] + dbz[id] *
(ey[i][j][k+1] - ey[i][j][k]) - dby[id] * (ez[i][j+1][k] -
ez[i][j][k]);

hy[i][j][k] = da[id] * hy[i][j][k] + dbx[id] *
(ez[i+1][j][k] - ez[i][j][k]) - dbz[id] * (ex[i][j][k+1] -
ex[i][j][k]);

hz[i][j][k] = da[id] * hz[i][j][k] + dbx[id] *
(ex[i][j+1][k] - ex[i][j][k]) - dbz[id] * (ey[i+1][j][k] -
ey[i][j][k]);
```

Listing 4.4.2.1: Main update equations example for 3D FDTD.
(Source: http://www.eecs.wsu.edu/~schneidj/ufdtd/chap9.pdf)

The three-dimension FDTD method is significantly more challenging to work with on CUDA than the two-dimension FDTD method. As mentioned above, CUDA architecture does not support running a large number of threads in three-dimensional, and thus alternative approaches have to be used to avoid this. In the two-dimensional FDTD method, the CUDA kernel was configured for two-dimensional blocks and grids because it simplifies programming. However, this could not be extended to the three-dimensional FDTD method. At this point, it is worth noting the limitations in CUDA’s block-grid configuration. Even though CUDA supports three-dimensional blocks, only 64 threads are allowed in the z-dimension of a block, compared to 512 for x- and y- dimensions. Additionally, CUDA does not support three-dimensional grids at all. To overcome this, we can use a two-dimensional configuration of blocks and threads. A for-loop can then be used to cycle through the z-dimension within the CUDA kernel, as shown in Listing 4.4.2.2:

```
// Kernel declaration
__global__ void kernel(float *d, int size_x, int size_y,
int size_z) {
    unsigned int xId=blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int yId=blockIdx.y * blockDim.y + threadIdx.y;
    if ( xId < size_x && yId < size_y ) {
          for ( int zId = 0; zId < size_z; zId ++ ) {
               ...
               d[index] = ...;
          }
     }
}

int main() {
     ...
     dim3 grid (ceil ((float)size_x/16), ceil
((float)size_y/16), 1);
     dim3 threads(16, 16, 1);

     //Kernel invoking
     kernel<<<grid, threads>>>(d, size_x, size_y, size_z);
}
```

Listing 4.4.2.2: 3D converted loop using 2D blocks and grids in CUDA.

CHAPTER 5 COMPUTATION ACCELERATION

5.1 PERFORMANCE OPTIMIZATION STRATEGIES

Performance optimization revolves around three basic strategies:

- Maximize parallel execution to achieve maximum utilization;
- Optimize memory usage to achieve maximum memory throughput;
- Optimize instruction usage to achieve maximum instruction throughput.

To achieve parallel execution on GPUs, the original code should be converted to a CUDA code. A general sequence of a CUDA code follows these 6 steps:

- Initialize GPU.
- Allocate variables.
- Transfer data form CPU to GPU.
- GPU computation.
- Transfer data from GPU to CPU.
- Finalize GPU.

In a traditional for-loop, we have the initialization part, the condition part, the afterthought part, and the loop body, whereas in CUDA, this is converted into a pair of CUDA kernel code and kernel call. A kernel in CUDA is like a function in C/C++, which is a method in Java/python. CUDA generates thousands of GPU threads that simultaneously execute kernel programs on stream processors. Figure 5.1.1 is a CUDA kernel pseudo call and code.

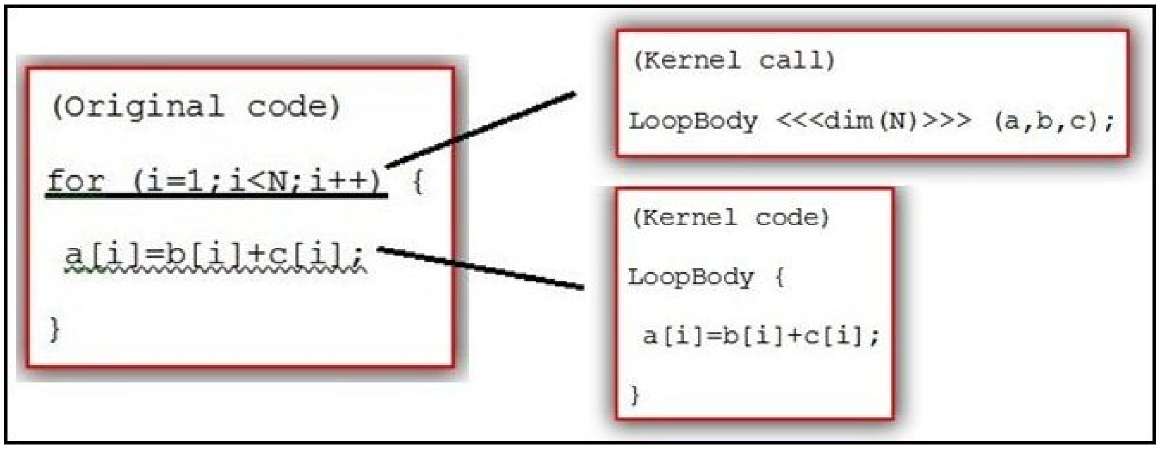


Figure 5.1.1: CUDA kernel pseudo call and code.

Listing 5.1.1 is another pair of real CUDA kernel call and kernel program.

```
// kernel call
exKernel<<<grids, threads>>>(dEx, dHy, AHz);

// CUDA kernel program
__global__ void exKernel(float*dEx, float*dHy, float*dHz){
     int i = blockDim.x * blockIdx.x + threadIdx.x;
     int j = blockDim.y * blockIdx.y + threadIdx.y;
     int k = blockDim.z * blockIdx.z + threadIdx.z;
     int n = Nx * Ny* k +Nx * j + i;
     dEx[n] = c0 * dEx[n] + c1 * (dHz[n] - dHz[n-Nx]) - c2
* (dHy[n] - dHy[n-Nx*Ny]);
}
```

Listing 5.1.1: A pair of CUDA kernel call (upper) and kernel program (lower).

Within CUDA, hardware is free to schedule thread blocks on any processor, which could be 1D/2D/3D. This simplifies memory addressing when processing multi-dimensional data. A kernel is executed by a grid of blocks. A block is a batch of threads that can cooperate with each other by sharing data through shared memory and synchronizing execution. In this section, we will start with a test of accelerated computation to sum squares in arrays. We did this first in a traditional way and compared the results on CPU and GPU, making sure there are no calculation errors. Most importantly, we tracked the relative running time.

In CUDA, __global__ is added in front of a function to indicate that this function is about to be executed on GPU:

```
__global__ static void sumOfSquares(int *num, int* result)
{
     int sum = 0;
     int i;
     for(i = 0; i < DATA_SIZE; i++) {
          sum += num * num;
     }
     *result = sum;
}
```

Listing 5.1.2: __global__ function calculating sum of squares.

Some limitations exist for GPU executing programs. For example, there cannot be any return values. Our next step is to have CUDA run this code, using the pattern:

```
functionName<<<BLOCK_NUMBER,THREAD_NUMBER,SHARED_MEMORY_SIZ
E>>> (Parameters);
```

After calling it, we need to transfer the results back to CPU memory:

```
sumOfSquares<<<1, 1, 0>>>(gpudata, result);
int sum;
cudaMemcpy(&sum, result, sizeof(int),
cudaMemcpyDeviceToHost);
cudaFree(gpudata);
cudaFree(result);
printf("sum: %d\n", sum);
```

Listing 5.1.3: __global__ function invoking.

The numbers of blocks and threads are both 1, since this code used only one thread running. We did not use any shared memory, so it was set to 0. To ensure there is no error in the calculation, we can add CPU computing code for verification:

```
sum = 0;
for(int i = 0; i < DATA_SIZE; i++) {
     sum += data * data;
}
printf("sum (CPU): %d\n", sum);
```

Listing: 5.1.4: CPU computing code for verification.

CUDA provides us with a clock function that can obtain the current timestamp. This is useful for determining the time spent on program execution, with the frequency of GPU execution units used as a very small metric cell for the running time. To do this, the sumOfSquares function code should be changed into:

```
__global__ static void sumOfSquares(int *num, int* result,
     clock_t* time) {

     int sum = 0;
     int i;
     clock_t start = clock();
     for(i = 0; i < DATA_SIZE; i++) {
          sum += num * num;
     }

     *result = sum;
     *time = clock() - start;
}
```

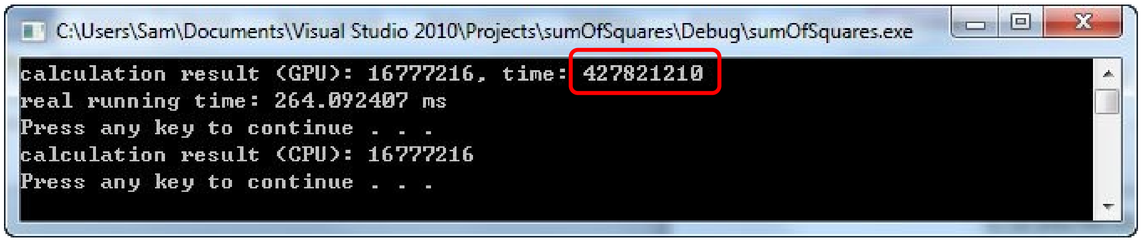
Listing: 5.1.5: Code for time recording.

And part of the main function should be rewritten as:

```
int* gpudata, *result;
clock_t* time;
cudaMalloc((void**) &gpudata, sizeof(int) * DATA_SIZE);
cudaMalloc((void**) &result, sizeof(int));
cudaMalloc((void**) &time, sizeof(clock_t));
cudaMemcpy(gpudata, data, sizeof(int) * DATA_SIZE,
     cudaMemcpyHostToDevice);
sumOfSquares<<<1, 1, 0>>>(gpudata, result, time);

int sum;
clock_t time_used;
cudaMemcpy(&sum, result, sizeof(int),
cudaMemcpyDeviceToHost);
cudaMemcpy(&time_used, time, sizeof(clock_t),
cudaMemcpyDeviceToHost);
cudaFree(gpudata);
cudaFree(result);
printf("sum: %d time: %d\n", sum, time_used);
```

Listing 5.1.6: Part of main function with data transfer between host and device.

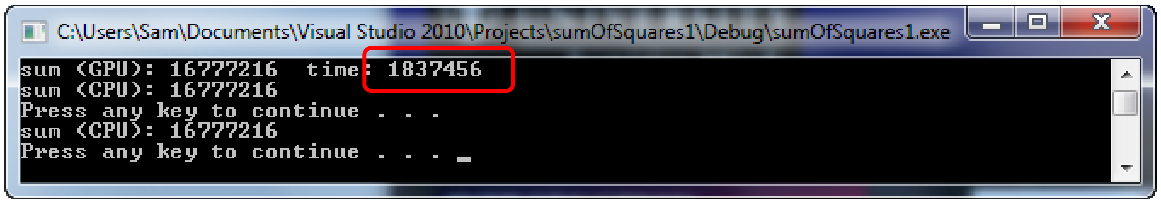
The result is close to 428 million time units (running time 264.09 ms), which, as mentioned above, is the frequency of GPU execution units used as the very small metric cell for the running time. This is shown in Figure 5.1.2. Keeping this in mind, let us see what happens next.


C:\Users\Sam\Documents\Visual Studio 2010\Projects\sumOfSquares\Debug\sumOfSquares.exe

```
calculation result (GPU): 16777216, time: 427821210
real running time: 264.092407 ms
Press any key to continue . . .
calculation result (CPU): 16777216
Press any key to continue . . .
```

Figure 5.1.2: sumOfSquares: initial run result.

Additional threads can be brought into the program. We can achieve this by adding #define THREAD_NUM after #define DATA_SIZE. We then start trying to parallelize it, adding 256 threads and a variable named threadIdx, thereby changing the entire code. The result for time is now 1.8 million time units, which is 233 times faster than before. That is the result of eliminating latency with a large amount of threads and the power of parallelization computing. Of course, this example is just about a simple matrix computation; in real, larger, and more sophisticated example problems, the time will be slower. This example is very typical and representative of the best performance of GPU parallel computing.


C:\Users\Sam\Documents\Visual Studio 2010\Projects\sumOfSquares1\Debug\sumOfSquares1.exe

```
sum (GPU): 16777216  time: 1837456
sum (CPU): 16777216
Press any key to continue . . .
sum (CPU): 16777216
Press any key to continue . . .
```

Figure 5.1.3: sumOfSquares test result with 256 threads defined.

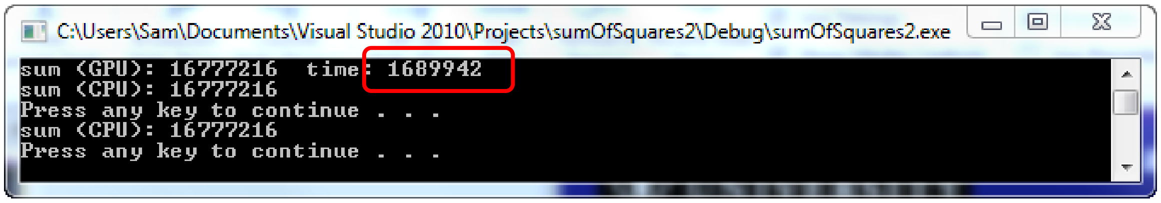
We know that the memory on GPUs is Dynamic Random-Access Memory (DRAM), so the most efficient way to access data would be doing it continuously. DRAM is a type of random-access memory that stores each bit of data in a separate capacitor within an integrated circuit. When a thread is waiting for data from memory, GPU will switch to the next thread, which changes the code into something like this:

```
__global__ static void sumOfSquares(int* num, int* result,
clock_t* time){
     const int tid = threadIdx.x;
     int sum = 0;
     int i;
     clock_t start;
     if (tid == 0)
          start = clock();
     for (i = tid; i < DATA_SIZE; i += THREAD_NUM){
          sum += num * num;
     }
     result[tid] = sum;
     if (tid == 0)
          *time = clock() - start;
}
```

Listing 5.1.6: Efficient data access for GPU.

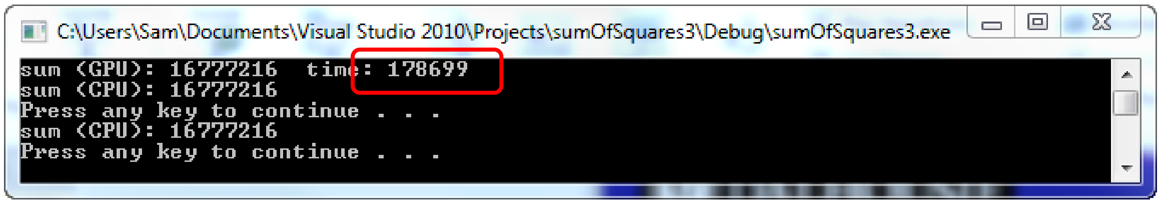
When one thread is waiting, GPU will switch to the next thread, so the actual order of execution is: thread 0 -> thread 1 -> thread 2 ->....

As we see in Figure 5.1.5, this gives us a result of 1.68 million time units, which is 1.1 times faster than last time. We can keep changing the definition THREAD_NUM into 512, and even 1024, which makes the result 0.8 million and 0.68 million, respectively. On the other hand, multiple blocks can be used to further increase the number of threads. For example, when BLOCK_NUM 32 is added to our code, the result is 0.18 million, which is around 4 more times faster. This is shown in Figure 5.1.6.


C:\Users\Sam\Documents\Visual Studio 2010\Projects\sumOfSquares2\Debug\sumOfSquares2.exe

```
sum (GPU): 16777216  time: 1689942
sum (CPU): 16777216
Press any key to continue . . .
sum (CPU): 16777216
Press any key to continue . . .
```

Figure 5.1.4: sumOfSquares test result: fixed way for data access.


C:\Users\Sam\Documents\Visual Studio 2010\Projects\sumOfSquares3\Debug\sumOfSquares3.exe

```
sum (GPU): 16777216  time: 178699
sum (CPU): 16777216
Press any key to continue . . .
sum (CPU): 16777216
Press any key to continue . . .
```

Figure 5.1.5: sumOfSquares test result: with BLOCK_NUM of 32 defined.

Here, four representative cases are selected to make the general speed-up comparison:

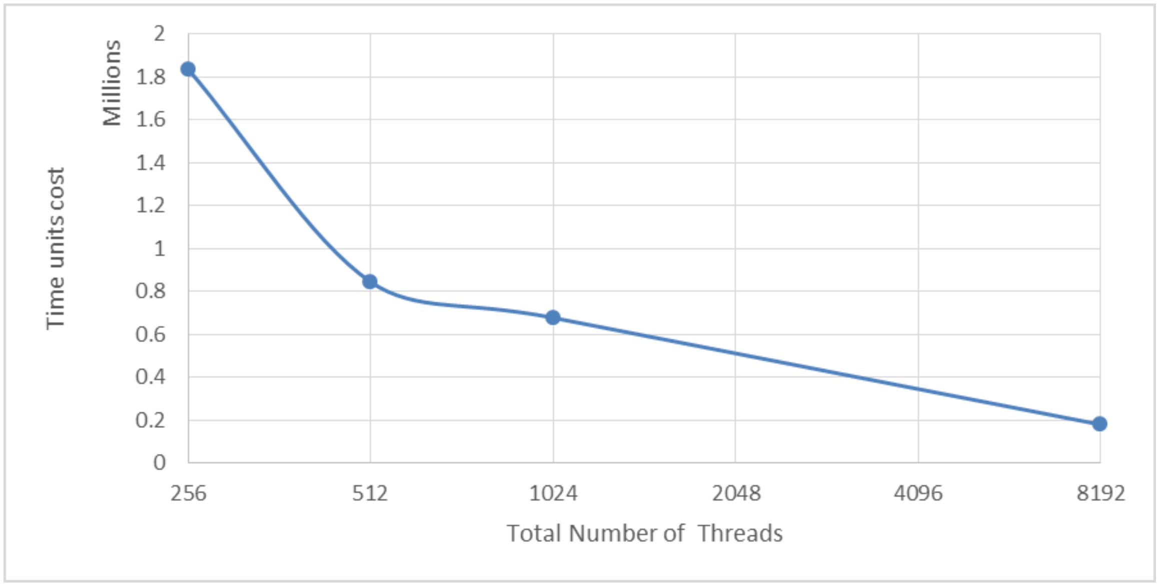


Figure 5.1.6: Four representative cases for general speed-up comparison.

5.2 MEMORY ACCESSES

5.2.1 Global Memory

There are different kinds of memory on a CUDA device, each with different scope, lifetime, and caching behavior. Global memory, which resides in device DRAM, can be used for transfers between the host and device as well as for the data input to and output from kernels. The name global here refers to scope, as it can be accessed and modified from both the host and the device. Global memory can be declared in global (variable) scope using the __device__ declaration specifier as in the first line of the following code snippet, or dynamically allocated using cudaMalloc() and assigned to a regular C pointer variable. Global memory allocations can be alive for the lifetime of the application. Depending on the compute capability of the device, global memory may or may not be cached on the chip.

The __global__ qualifier declares a function as being a kernel. Such a function is executed on the device; the callable form is on the host only. The __global__ functions must have a void return type, such that any call to a __global__ function must specify its execution configuration. A call to a __global__ function is asynchronous, meaning it returns before the device has completed its execution.

We have discussed how threads are grouped into thread blocks, which are assigned to multiprocessors on the device. There is a finer grouping of threads into warps during execution. Multiprocessors on the GPU execute instructions for each warp in SIMD fashion. The warp size (effectively, the SIMD width) of all current CUDA-capable GPUs is 32 threads.

5.2.2 Local Memory

Local memory space resides in device memory, so local memory accesses have the same high latency and low bandwidth as global memory accesses and are subject to the same requirements for memory coalescing. Local memory, however, is organized such that consecutive 32-bit words are accessed by consecutive thread IDs. Accesses are therefore fully coalesced as long as all threads in a warp access the same relative address (e.g., same index in an array variable, same member in a structure variable). Local memory is just thread local global memory. It is slower (both in terms of bandwidth and latency) than either registers or shared memory. It also consumes memory controller bandwidth that would otherwise be available for global memory transactions. The performance impact of spilling or deliberately using local memory can be anything from minor to severe, depending on the hardware being utilized and how local memory is used.

Local memory accesses only occur for some automatic variables that the compiler is likely to place in local memory. These include arrays that may or may not be indexed with constant quantities, large structures or arrays that would consume too much register space, and any variable if the kernel uses more registers than available (also known as register spilling).

5.2.3 Shared Memory

Shared memory latency is a lot lower than uncached global memory latency (provided that there are no bank conflicts between the threads, which we will examine later in this section). Shared memory is allocated per thread block. Hence, all threads in the block have access to the same shared memory. Threads can access data in shared memory loaded from global memory by other threads within the same thread block. This capability (combined with thread synchronization) has a number of uses, such as user-managed data caches, high-performance cooperative parallel algorithms (e.g., parallel reductions), and global memory coalescing facilitation in cases where it would otherwise not be possible.

While threads in a block run logically in parallel, not all threads can execute physically at the same time. Therefore, we need to be careful to avoid race conditions when sharing data between threads. For example, thread A and thread B load a data element from global memory and store it to shared memory. Then, A wants to read B's element from shared memory, and vice versa. Let's assume that A and B are threads in two different warps. If B has not finished writing its element before A tries to read it, we have a race condition, which can lead to undefined behavior and wrong results. To avoid this, we should synchronize threads, using __syncthreads(). A thread execution can only proceed past a __syncthreads() after all threads in its block have executed the __syncthreads(). So we can avoid the race condition described above by calling __syncthreads() after the store to shared memory and before any threads load from shared memory. It is important to be aware that calling __syncthreads() in divergent code is undefined and can lead to a deadlock situation where all threads within a thread block must call __syncthreads() at the same point.

__shared__ variable declaration specifier is used to declare shared memory. Type qualifiers have the following characteristics: Variables in thread registers only stay in the kernel; variables in global memory threads only stay in the kernel; __device__ __shared__ type variables in shared memory in a block only stay in the kernel; __device__ type variables in global memory in a grid stay until the application exits; and __device__ __constant__ type variables in a grid stay until the application exits.

```
// matrix kernel using global and shared memory
__global__ void matmulShared(float* c, float* a, float* b,
int N ){
     int col = blockIdx .x * blockDim .x + threadIdx .x;
     int row = blockIdx .y * blockDim .y + threadIdx .y;
     int M = ( N + BlockSize - 1 ) / BlockSize ;
     float sum = 0.0;
     for ( int m = 0; m < M; m++ ){
          // all threads in block copy their element from
          // matrix a and matrix b to shared memory
          __shared__ float a_s[ BlockSize ][ BlockSize ];
          __shared__ float b_s[ BlockSize ][ BlockSize ];
          int c = m * BlockSize + threadIdx .x;
          int r = m * BlockSize + threadIdx .y;
          a_s[threadIdx.y][threadIdx.x] = a[IDX(row,c,N)];
          b_s[threadIdx.y][threadIdx.x] = b[IDX(r,col,N)];
          __syncthreads ();
          int K = (m == M-1 ? (N-m*BlockSize) : BlockSize);
          for ( int k = 0; k < K; k++ ){
               sum+=a_s[threadIdx.y][k]*b_s[k][threadIdx.x];
          }
          __syncthreads ();
     }
     if ( col < N && row < N )
          c[ IDX(row ,col ,N)] = sum;
}
```

Listing 5.2.3.1: Sample code of __global__ and __shared__ memory combination.

(Source: www.math-cs.gordon.edu/courses/cps343/presentations/CUDA_Memory.pdf)

5.3 ACENET

ACEnet (Atlantic Computational Excellence Network) is a consortium of Atlantic Canadian Universities providing researchers with high performance computing (HPC) resources, collaboration and visualization tools, software, training, and support [79]. ACEnet is a partner consortium of Compute Canada, the organization responsible for research High Performance Computing in Canada.

ACEnet is Atlantic Canada's entry into this national fabric of HPC facilities. It is a partnership of seven institutions, including Memorial University of Newfoundland, University of New Brunswick, Saint Mary's University, St. Francis Xavier University, Dalhousie University, Mount Allison University, and the University of Prince Edward Island.

ACEnet creates and operates HPC facilities that interconnected by high-speed networks, allowing them to behave as a single, regionally distributed "computational power grid" of enormous capacity. ACEnet also creates and operates sophisticated video-teleconferencing facilities to bind together geographically dispersed research communities [80].

Each cluster in ACEnet consists of a number of computers or "nodes", acting like "managers"; and each node has several CPUs with multiple cores that can be treated as single-core processors, acting like "workers". These machines are running Red Hat Enterprise Linux (RHEL) 4.

The principal benefit of high-performance computing arises from the ability to apply many CPUs to a single problem – parallel computing. Three models of parallel computing are supported at ACEnet. "Embarrassingly" or "perfectly" parallel problems can be treated with a collection of independent serial jobs. Such usage is supported via the job scheduler, particularly task arrays. Some sources refer to this as High-Throughput Computing (as distinct from High-Performance Computing), but regardless of what it is called, it is supported at ACEnet. One of the ACEnet clusters called Brasdor is the preferred platform for such work. Message-passing and shared-memory parallel computing are supported with the MPI and OpenMP application programming interface (API), respectively. If we intend to either develop or run code that uses MPI or OpenMP, they would be very good choices.

The ACEnet hardware resources are located at several universities and include the following clusters [81]:

- Brasdor (brasdor.ace-net.ca) at St. Francis Xavier University
- Fundy (fundy.ace-net.ca) at University of New Brunswick
- Mahone (mahone.ace-net.ca) at Saint Mary's University
- Placentia (placentia.ace-net.ca) at Memorial University
- Glooscap (glooscap.ace-net.ca) at Dalhousie University
- Courtenay (courtenay.ace-net.ca) at University of New Brunswick (Saint John)

For CUDA compiling and testing purposes we can request an interactive job, as in this example:

For compiling and testing purposes, we can request an interactive job, as shown in Listing 5.3.1 below:

```
$ qrsh -l h_rt=3:0:0 -pe openmp 4 -q graphics.q –cwd bash
$ module load gcc cuda
$ nvcc source.cu -o executable
$ ./executable
```

Listing 5.3.1: ACEnet compiling and testing request using PuTTY.

For a batch job, we can use:

```
#$ -cwd
#$ -l h_rt=1:0:0
#$ -pe openmp 4
#$ -q graphics.q
```

```
module load gcc cuda
./executable
```

Listing 5.3.2: ACEnet commands for batch jobs.

A more detailed discussion on implementation and performance is provided in the following sections.

5.4 LOCAL PLATFORM SPEED-UP PERFORMANCE

As stated in Chapter 4, the first GPU testing environment we used is GeForce GT 520 with 48 CUDA cores. The detailed specifications of the GPU and CPU in the test can be found in Table 5.4.1.1 and Table 5.4.1.2, respectively.

GPU Version	GeForce GT 520
Core Speed	810 MHz
Architecture	Fermi GF119
Notebook GPU	No
Release Date	14 May 2011
Memory	1024 MB
Memory Speed	900 MHz
Memory Bus	64 Bit
Memory Type	DDR3
Memory Bandwidth	14.4 GB/sec
Shader Processing Units	48
Actual Shader Performance	96
Technology	40 nm
Texture Mapping Units	8
Texture Rate	6.5 GTexel/s
Render Output Units	4

Pixel Rate	3.2 GPixel/s
DirectX	11
Shader Model	5.0
Open GL	4.1
Max Power	29 Watts

Table 5.4.1.1: Specifications summary of the testing GPU: GeForce GT 520.

CPU Version	Intel i7-2600
Microarchitecture	Sandy Bridge
Release Date	January 9, 2011
Technology (micron)	0.032
Frequency	3.4 GHz
Turbo Frequency	3.8 / 3.7 / 3.6 / 3.5 GHz
Data Width	64 bits
Clock Multiplier	34
Cores	4
Threads	8
Turbo Core / Turbo Boost	Yes
Unlocked multiplier	No
Number of Controllers	1
Supported Memory	DDR3-1066, DDR3-1333

Table 5.4.1.2: Detailed information of CPU in the testing computer.

We simulated an electromagnetic cavity with metal walls and a point source inside. The 2D code was written to execute the main update equation using CUDA. Specifically, Ex, Ey and Hz update equations are executed. The Ex and Ey fields had to be transferred from the host to CUDA before executing a CUDA kernel, and then transferred back from CUDA to the host after the kernel had finished executing. As expected from the previous discussion, the memory operations were costly and resulted in a slower execution.

To validate our method, we simulated the cavity of 1000mm×1000mm with perfect electrical conductor walls. A point source of a modulated Gausses function with the carrier frequency of 0.5GHz and bandwidth of 0.5GHz was located at the location of (260mm, 250mm) in the cavity. The size of the cavity is 100 by 100 cells and each cell size is 10 mm×10mm. The time-domain E-field at location (760mm, 250mm) was recorded and shown in Figure 5.4.1a. The Fourier transform of the E-field is shown in Figure 5.4.1b, and the spikes indicate the resonant frequencies. Table 5.4.1.3 is the comparison between the results obtained from our FDTD method and the theoretical resonant frequencies. It is easy to find that the maximum difference is 5.96%. Thus, the results obtained from our method agree well with the theoretical results.

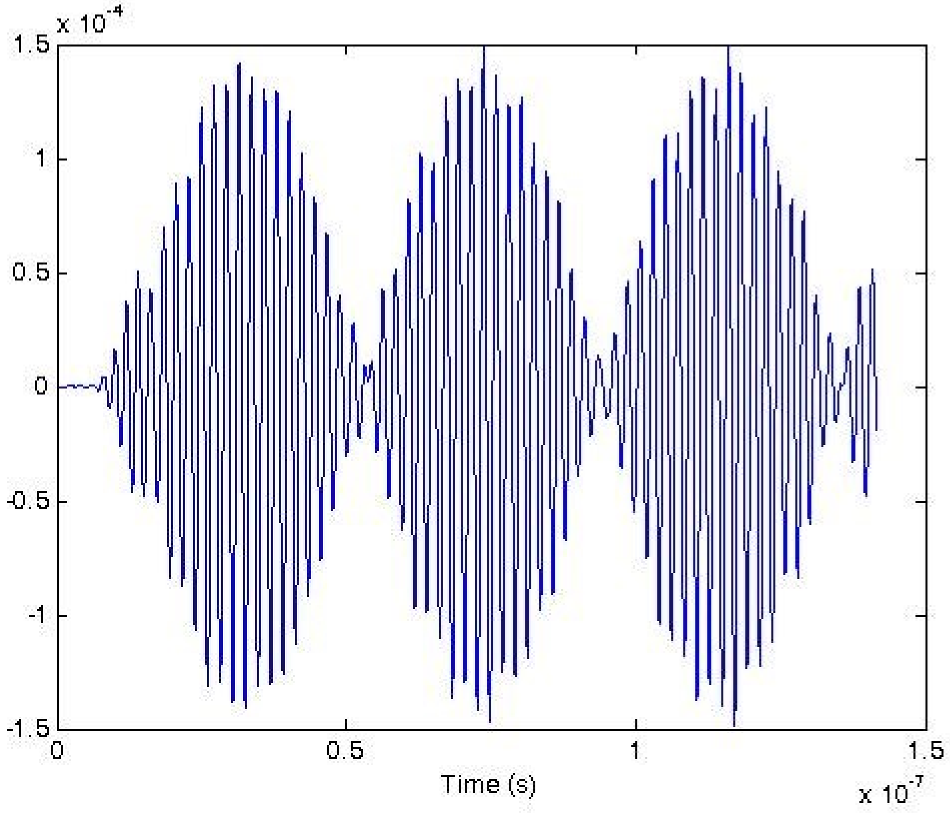


Figure 5.4.1a: The time-domain E-field recorded.

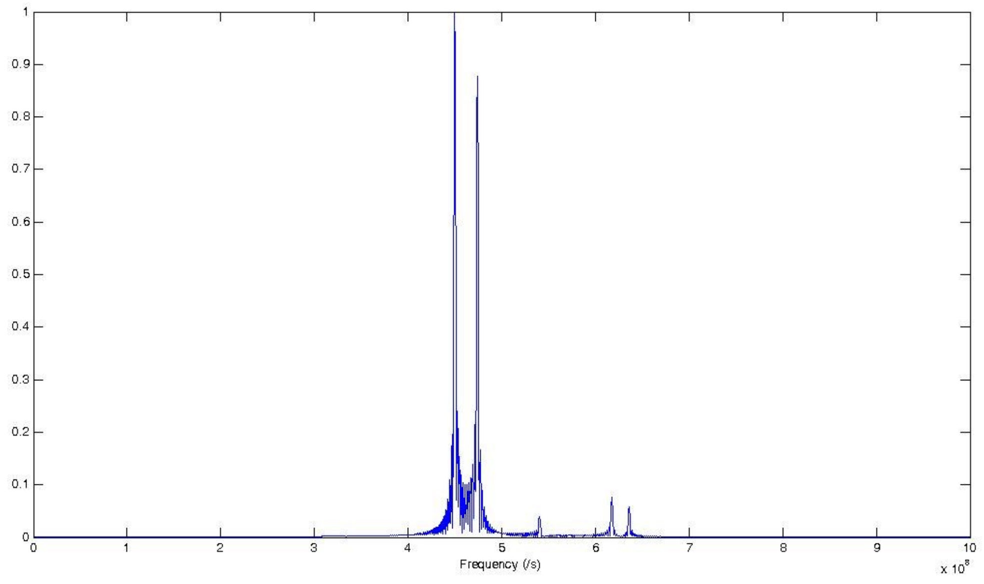


Figure 5.4.1b: The E-field in the frequency domain (100 by 100 cells with 21198 steps).

Mode	TE_{22}	TE_{13}	TE_{23}	TE_{14}	TE_{33}
Fre_theory (GHz)	0.4243	0.4743	0.5408	0.6185	0.6364
Fre_computed (GHz)	0.4496	0.4733	0.5399	0.6160	0.6356
Deviation	5.96%	0.21%	0.17%	0.40%	0.13%

Table 5.4.1.3: Comparison between the results obtained from our method and the theoretical resonant frequencies.

The original CPU execution time results with 1000 FDTD time steps are shown from Figure 5.4.1.1 to Figure 5.4.1.5, and CPU execution time results can be seen in Figure 5.4.1.6 (with 1000 time steps).

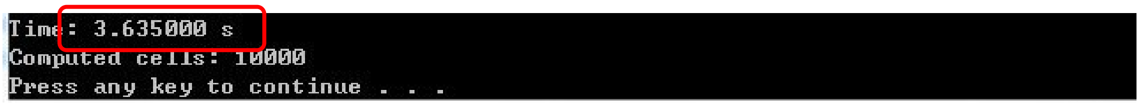

```
Time: 3.635000 s
Computed cells: 10000
Press any key to continue . . .
```

Figure 5.4.1.1: FDTD simulation size of 100×100 with 1000 time steps (CPU).

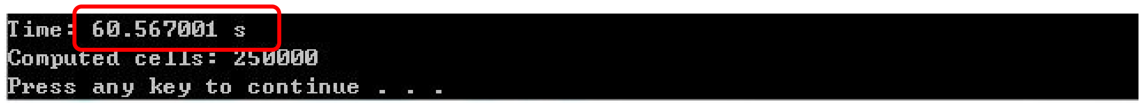
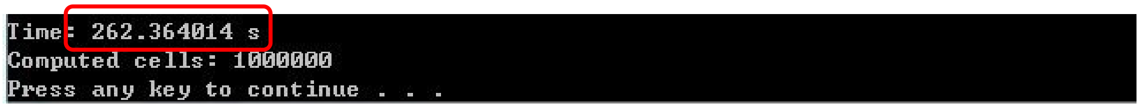
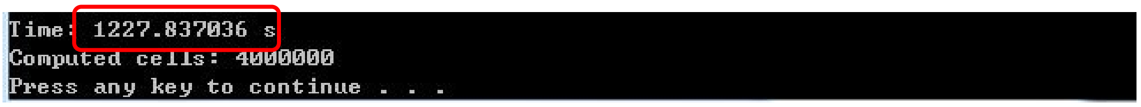

```
Time: 60.567001 s
Computed cells: 250000
Press any key to continue . . .
```

Figure 5.4.1.2: FDTD simulation size of 500×500 with 1000 time steps (CPU).



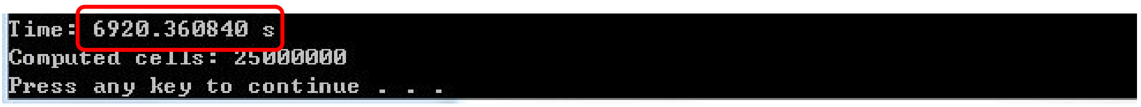
```
Time: 262.364014 s
Computed cells: 1000000
Press any key to continue . . .
```

Figure 5.4.1.3: FDTD simulation size of 1000×1000 with 1000 time steps (CPU).



```
Time: 1227.837036 s
Computed cells: 4000000
Press any key to continue . . .
```

Figure 5.4.1.4: FDTD simulation size of 2000×2000 with 1000 time steps (CPU).



```
Time: 6920.360840 s
Computed cells: 25000000
Press any key to continue . . .
```

Figure 5.4.1.5: FDTD simulation size of 5000×5000 with 1000 time steps (CPU).

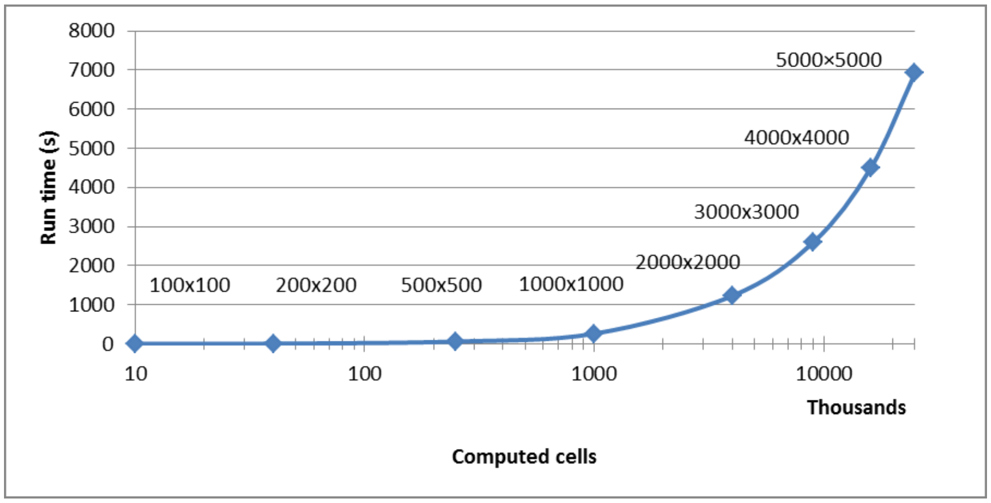


Figure 5.4.1.6: Simulation results on CPU.

The simulation of field distribution can be seen in Figure 5.4.1.7.

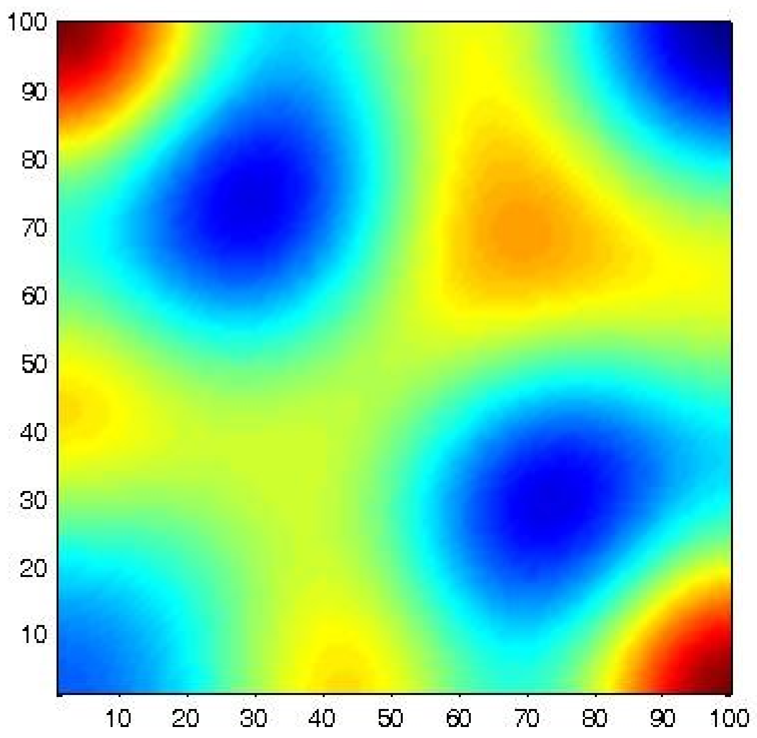
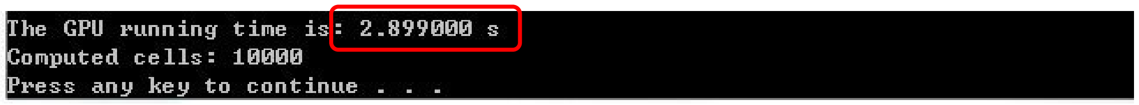


Figure 5.4.1.7: FDTD simulation result of Hz field distribution (at 1000 time step)

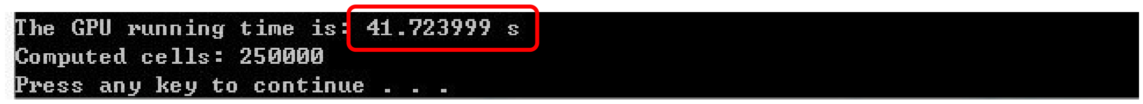
In a CUDA parallel programming model, it is assumed that the parallel program is executed on independent physical facilities, which is called Device. Facilities are treated as coprocessors of the Host, which runs the C code. In CUDA, each block of threads can be scheduled on any of the available multiprocessors within a GPU, in any order, concurrently or sequentially, so that a compiled CUDA program can execute on any number of multiprocessors.

Using various combinations of threads and blocks, we got the corresponding simulation results, some of which have been illustrated in Section 4.3.2. Here we select some representative results with 1000 FDTD time steps to make the specific comparisons, as shown from Figure 5.4.1.8 to Figure 5.4.1.12:



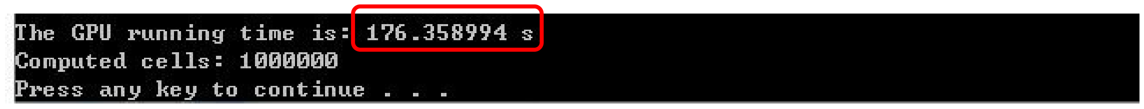
```
The GPU running time is: 2.899000 s
Computed cells: 10000
Press any key to continue . . .
```

Figure 5.4.1.8: FDTD simulation size of 100×100 with 1000 time steps (GPU).



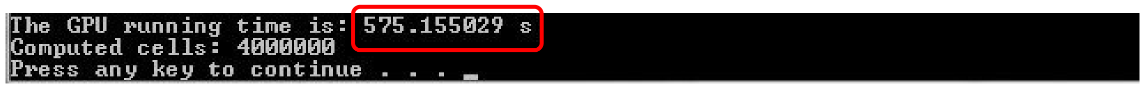
```
The GPU running time is: 41.723999 s
Computed cells: 250000
Press any key to continue . . .
```

Figure 5.4.1.9: FDTD simulation size of 500×500 with 1000 time steps (GPU).



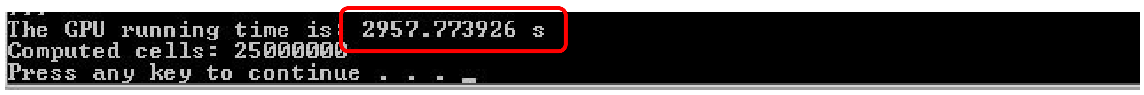
```
The GPU running time is: 176.358994 s
Computed cells: 1000000
Press any key to continue . . .
```

Figure 5.4.1.10: FDTD simulation size of 1000×1000 with 1000 time steps (GPU).



```
The GPU running time is: 575.155029 s
Computed cells: 4000000
Press any key to continue . . .
```

Figure 5.4.1.11: FDTD simulation size of 2000×2000 with 1000 time steps (GPU).



```
The GPU running time is: 2957.773926 s
Computed cells: 25000000
Press any key to continue . . .
```

Figure 5.4.1.12: FDTD simulation size of 5000×5000 with 1000 time steps (GPU).

We can then show a general speed comparison between CPU and GPU, as shown in Figure 5.4.1.13.

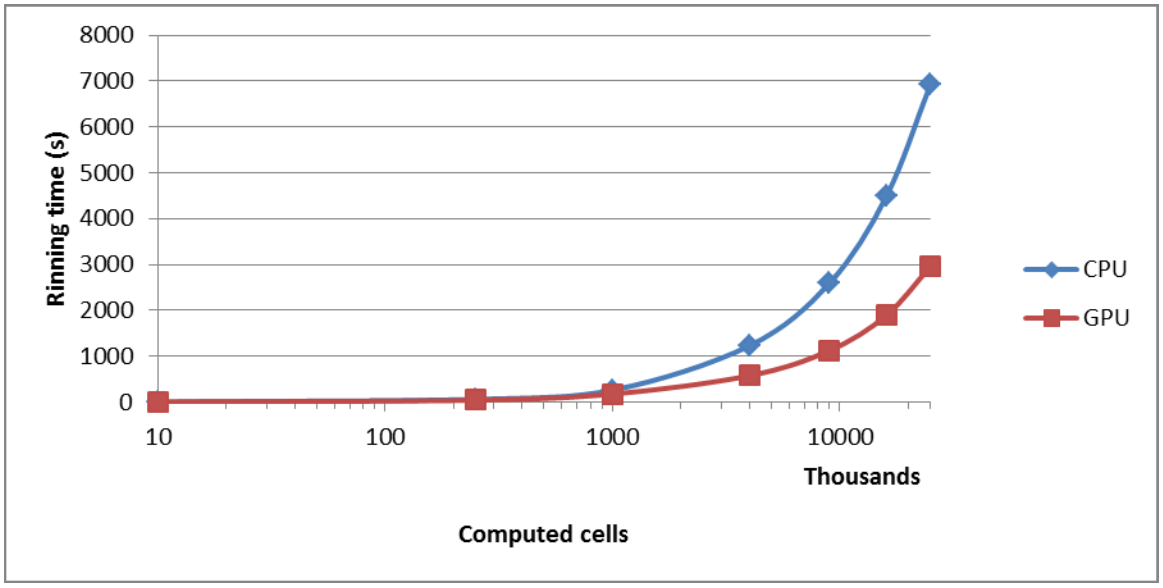


Figure 5.4.1.13: Comparison between CPU and CUDA GPU simulation speeds.

From the results, we realize that when the computing scale is not big enough, there are no significant differences between CPU and GPU. That is because the total number of threads used is relatively small, and thus cannot properly avoid memory access latency issues. As a result, the advantages of GPU parallel computing have not been demonstrated. With increasing cell numbers, more obvious calculation speed boost can be achieved and seen. There is one thing we have to clarify that, not all the FDTD algotithms are suitable for GPU parallel computing. There are a lot of factors that can limit the maximum speed, such as whether the number of the computed cells is too small, what kind of environment the code is executed in, or what structures are there in the FDTD algorithm. For our case, part of the algorithm depends a lot on the CPU. Specifically, when the code is processed at the "tridiagonal" part, there must be data transferring from CPU to GPU, as stated in Chapter 4, and then from GPU back to CPU afterwards. This part would be relatively costly, so that the there is not such an amazing speed-up for dozens or even hundreds of times as shown in our test within Section 5.1. If we can make the sub-function parallel as well, or do something on optimizing the algorithm, there would be a tremendous speed-up achievement. We will further discuss this in the next section.

For computing precision, plots of FDTD simulation results of accuracy between CPU and GPU computing have been given below in Figure 5.4.1.14. The plots are generated from the results of simulation size 100 × 100. The location of observation is at Ex cell (50, 50).

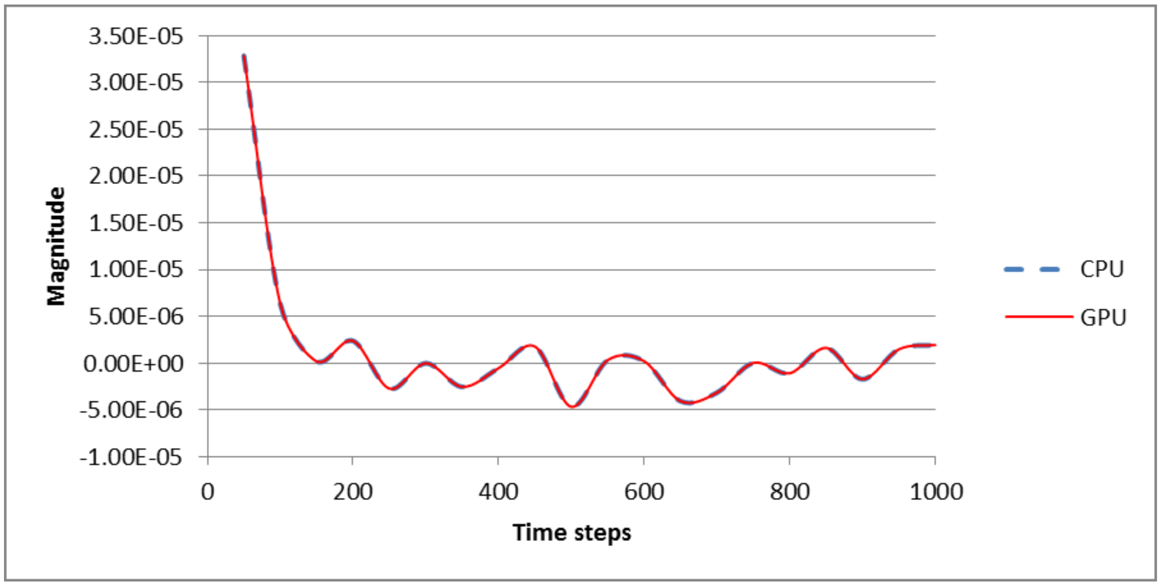


Figure 5.4.1.14: Plot of simulation results to compare accuracy between CPU and GPU.

The next GPU testing environment we used is GeForce GTX 650, whose detailed specifications in the test can be found in Table 5.4.1.4.

GPU Version	GeForce GTX 650
Core Speed	1058 MHz
Architecture	Kepler GK107
Notebook GPU	No
Release Date	01 Sep 2012
Memory	2048 MB
Memory Speed	1250 MHz
Memory Bus	128 Bit
Memory Type	GDDR5
Memory Bandwidth	80 GB/sec
Shader Processing Units	384
Actual Shader Performance	384
Technology	28 nm
Texture Mapping Units	32
Texture Rate	33.9 GTexel/s

Render Output Units	16
Pixel Rate	16.9 GPixel/s
DirectX	11.1
Shader Model	5.0
Open GL	4.3
Max Power	64 Watts

Table 5.4.1.4 Specifications summary of the testing GPU: GeForce GTX 650.

Similarly, we obtained the simulation results of the same problem and completed the comparison of the three test speeds as can be seen from Figure 5.4.1.15 below:

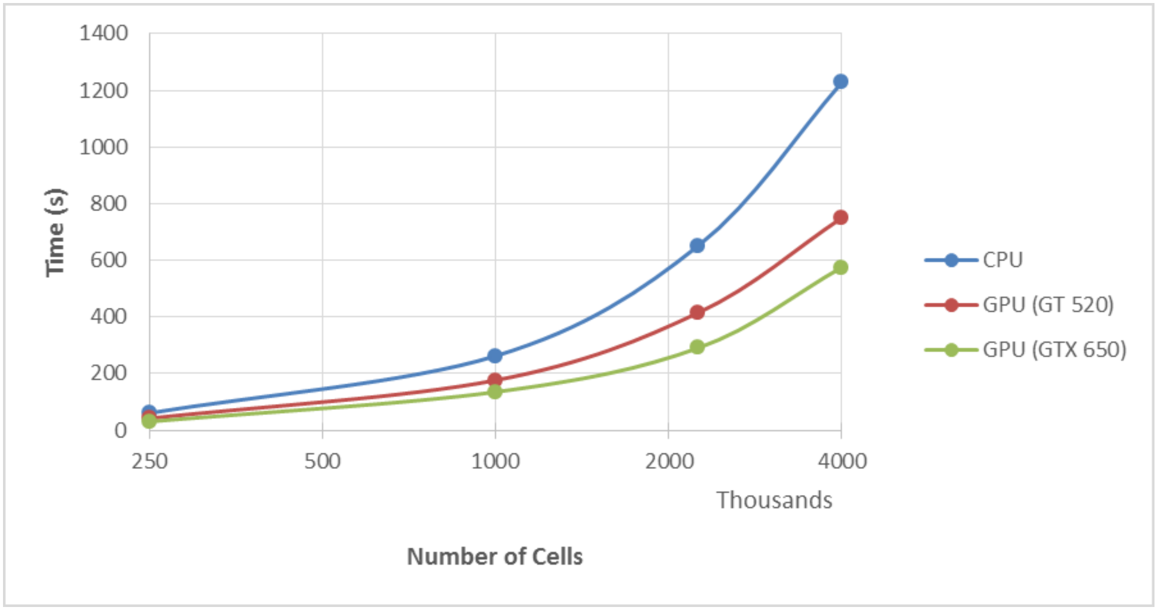


Figure 5.4.1.15 General local speed-up comparison

From the figure we noted that the running speed did not change a lot while the computing size was not large enough. However, when the cells kept increasing and exceeded a certain value, the speed-up performance became extremely notable.

5.5 MATLAB GPU PERFORMANCE AND ACENET COMPARISON

5.5.1 Implementation on ACEnet and Initial Run

Traditional C is not the only programming language that can be implemented with GPU computing and supported by ACEnet HPC platform. Matlab can also be used to demonstrate this part. As a high-level language and interactive environment for numerical computation, Matlab is able to work efficiently with the ACEnet Distributed Computing Server at the cluster of placentia.ace-net.ca.

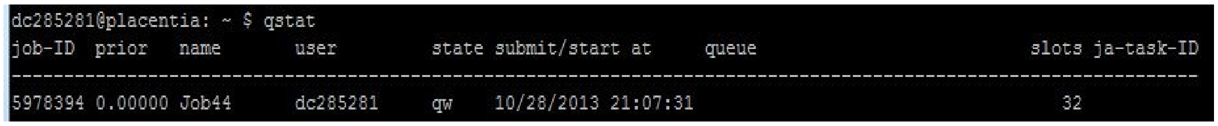
First, we need to create a Matlab workspace on Placentia, log in to Placentia, and make a "MATLAB" working directory (e.g., in PuTTY, this is done with mkdir ~/MATLAB). Next, we create communicating jobs and submit them to the cluster, each one a separate Grid Engine job. The commands are:

```
myCluster=parcluster('Placentia');
j=createCommunicatingJob(myCluster,'Type','pool');
createTask(j,@dvadi3d_div_3,0,{});
j.NumWorkersRange=[1 4];
submit(j);
```

Listing 5.5.1.1: ACEnet: submit parallel jobs.

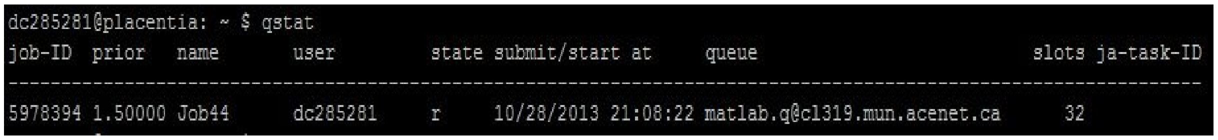
If we open a separate terminal session on placentia.ace-net.ca with PuTTY and use qstat, we can see jobs with names like "Job1.1", "Job1.2", "Job1.3", and "Job1.4", along with their working status. It takes some time for these to be scheduled and run. From within Matlab, we can also check "j.State" to see if the job is pending, running, or finished. When it is finished, we retrieve the results from the cluster and save them [79].

Currently, ACEnet supports multi-core parallel computing with a maximum of 32 CPU cores as "workers". With the increase of multi-cores supported on the Placentia cluster in the near future, larger scale computing jobs will be performed and more improvements will be made.



```
dc285281@placentia: ~ $ qstat
job-ID  prior   name       user         state submit/start at     queue                                  slots ja-task-ID
-----------------------------------------------------------------------------------------------------------------------
5978394 0.00000 Job44      dc285281     qw    10/28/2013 21:07:31                                           32
```

Figure 5.5.1.1: ACEnet test operation: check job status – waiting.



```
dc285281@placentia: ~ $ qstat
job-ID  prior   name       user         state submit/start at     queue                                  slots ja-task-ID
-----------------------------------------------------------------------------------------------------------------------
5978394 1.50000 Job44      dc285281     r     10/28/2013 21:08:22 matlab.q@cl319.mun.acenet.ca              32
```

Figure 5.5.1.2: ACEnet test operation: check job status – running.

In our initial run, we tested FDTD simulation at a scale of 1000×1000 cells and 2000×2000 cells, from 4-core to 32-core (max) parallelization. The results are shown in Figure 5.5.1.3 and Figure 5.5.1.4.

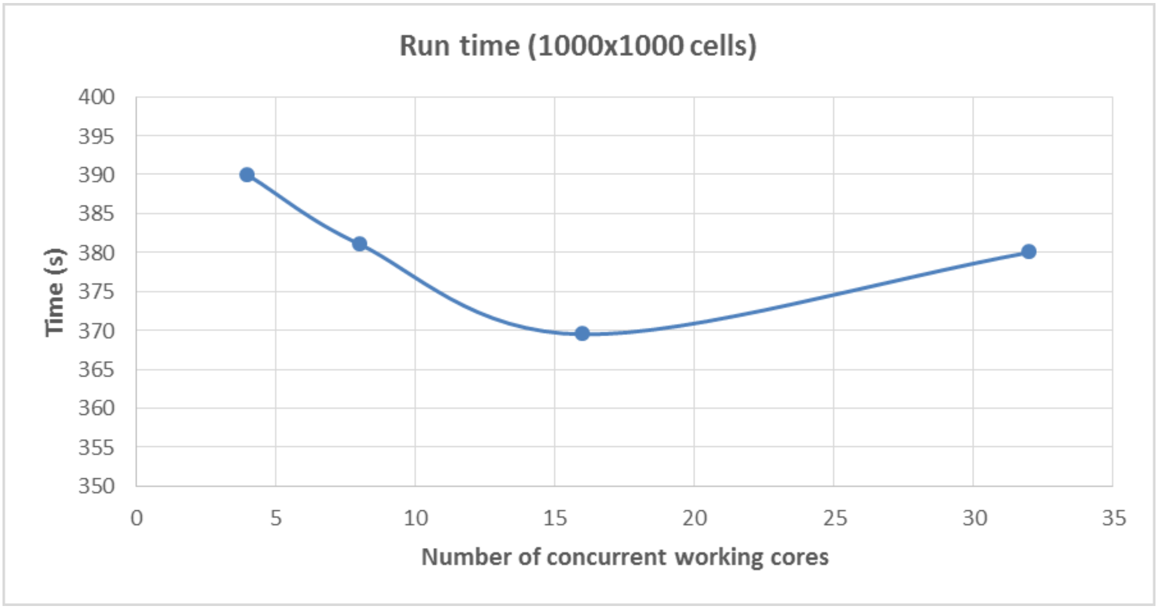


Figure 5.5.1.3: ACEnet running times with different number of cores (1000×1000 cells).

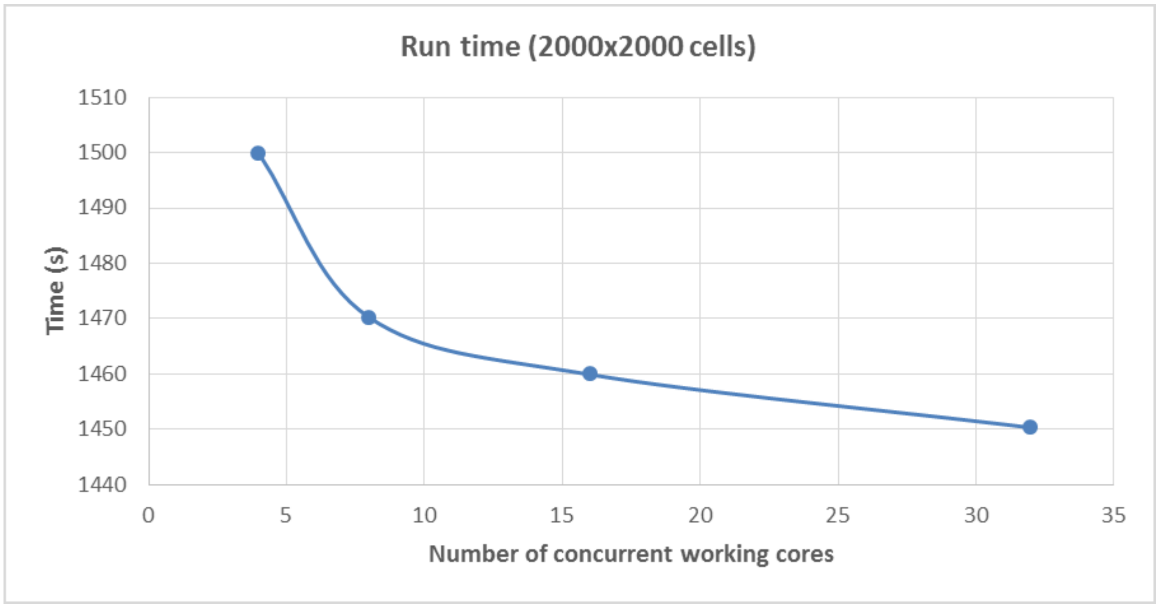


Figure 5.5.1.4: ACEnet running times with different number of cores (2000×2000 cells).

If we closely observe the first figure above (computing scale of 1000×1000 cells), we can see that there is a time rise from 16-core to 32-core parallel implementation. This is an obvious and common situation in parallel computing jobs, including GPU multi-thread calculation, as serial and parallel computations have significantly different implementations that make them better suited to different tasks. Parallel computing can handle large amounts of data in many streams, performing relatively simple operations on them, but is ill-suited to complex processing on a single or very few streams of data. As a result, they are not appropriate for handling small-scale tasks they do not greatly benefit from, due to numerous time-consuming factors such as allocating sub-jobs to multiply "workers" (e.g., threads in GPU, concurrent cores in this case on ACEnet) and gathering data back together from them.

When we enlarge the computing scale up to 2000×2000 cells, this kind of "running time rise" is currently eliminated, as can be seen from Figure 5.5.1.4. With the increase of the multi-core number, the running time decreases and the speed increases. Of course, this is the observation under the current conditions of maximum provided "workers" of ACEnet.

We will keep tracking this down as a part of our future work with the increasing of system development and openness.

5.5.2 MATLAB GPU Performance

Using a Matlab parallel method can sometimes reduce the computing time of FDTD, as will using a multi-thread method like CUDA C programming, described earlier in this chapter and in Chapter 4. Using MATLAB for GPU computing can accelerate programs with GPUs more easily than by using C or FORTRAN. With the familiar MATLAB language we can take advantage of the CUDA GPU computing technology without having to learn the intricacies of GPU architectures or low-level GPU computing libraries.

For comparison purposes, we used the FDTD simulation test at a scale from 100×100 cells up to 2000×2000 cells, from local CPU, GPU to 32-core ACEnet parallelization. The test contains two parts: first, the simulation of original algorithm by local CPU and ACEnet; and second, the optimizated algorithm by CPU and GPU parallel computing simulation.

First, we did this simulation on a local CPU – just like what we did before using C, only that the efficiency is lower than C. Then we submitted our job upon ACEnet. The running time data is saved within a .mat file and then copied to our local computer for further processing. On ACEnent we can even define how many "workers" we want as long as they are available and the total number is not exceeding the limit. Here we used 32.

We can check out the running time differences from the table below (both with 100 time steps):

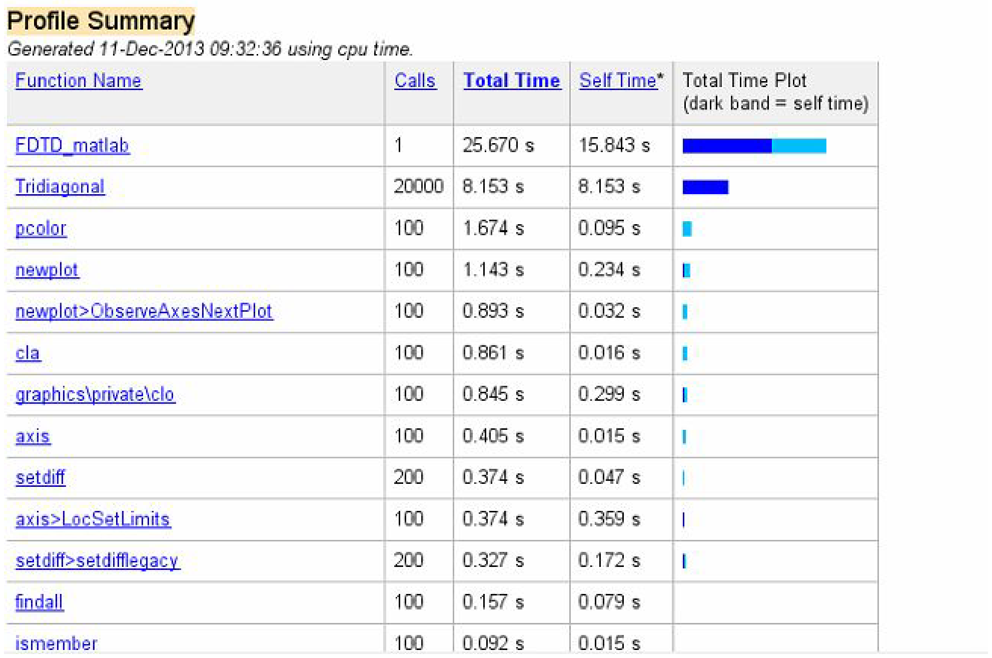
Computed cells	Local CPU running time (s)	ACEnet running time (s)
100 × 100	4.654956	2.168121
200 × 200	18.130818	7.739342

500 × 500	113.887529	52.161244
1000 × 1000	467.16541	210.004858
1500 × 1500	1003.703	449.130
2000 × 2000	1880.266284	849.136342

Table 5.5.2.1: Efficiency comparison between local CPU and ACEnet.

We notice that the 32-core ACEnet running performance is better than local computer, but without exciting surprises. Next, we tried to use GPU with Matlab. As said at the beginning of this section, using MATLAB for GPU computing can accelerate programs with GPUs more easily than some traditional languages. The “gpuArray” and “gather()” are two of the most important weapons for the implementation, for transferring data to and from GPU. But when we are done rewriting the code, we found that the speed is not accelerated at a very high level, sometimes even slower. Again, this is partly caused by the frequent data transferring between GPU and CPU. To overcome this, with no space for the deeper parallelization like the situation we mentioned at last section, we should come up with another algorithm which is more suitable for our parallel computing.

First, we need to identify the bottlenecks in the original program, with the help of profiler.



Profile Summary

Generated 11-Dec-2013 09:32:36 using cpu time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
FDTD_matlab	1	25.670 s	15.843 s	
Tridiagonal	20000	8.153 s	8.153 s	
pcolor	100	1.674 s	0.095 s	
newplot	100	1.143 s	0.234 s	
newplot>ObserveAxesNextPlot	100	0.893 s	0.032 s	
cla	100	0.861 s	0.016 s	
graphics\private\clo	100	0.845 s	0.299 s	
axis	100	0.405 s	0.015 s	
setdiff	200	0.374 s	0.047 s	
axis>LocSetLimits	100	0.374 s	0.359 s	
setdiff>setdifflegacy	200	0.327 s	0.172 s	
findall	100	0.157 s	0.079 s	
ismember	100	0.092 s	0.015 s	

Figure 5.5.2.1: Matlab profiler to check the bottleneck.

From here we can see that optimizing the sub-function of tridiagonal() is the key. But we also need to know that to get the best performance of GPU, FDTD for-loops should be rewritten and optimized. Porting all codes to GPU is the most desirable way, but not all the codes are suitable for the porting; moreover, there are some algorithms or a certain part of the algorithms not possible for GPU parallel computing. We have no other choices in computing them in unparallel ways. Fortunately, FDTD for-loops are possible for the parallel computing. First, data are transferred to GPU by the function of gpuArray.zeros (). And next, rewrite the for-loop

```
for iJ=1:iSXmax-1
    for iK=2:iSYmax-1
        QEx(iJ,iK) = Ex(iJ,iK);
    end
end
```

as

```
QEx(1:iSXmax-1,2:iSYmax-1)=Ex(1:iSXmax-1,2:iSYmax-1);
```

Because tridiagonal() itself is treated as a usage of catch-up method for solving equations, it would be difficult to simply modify the code as it involves iteration issues. We considered to use the existing diag() function to generate tridiagonal equations, and call the internal right-side-divide such that we could obtain a one-time solution of N equations, which will greatly improve the speed. In this way, we can also connect the tridiagonal() function to GPU since gpuArray supports right-side-divide and diag().

Therefore, we chose to use gallery() to generate the “tridiag” matrix, which greatly improved the efficiency. But the drawback is that we need to transfer the data in FDTD back to CPU, and then X would return to GPU for the follow-up when the periodical calculation is over. Fortunately, this does not affect the general efficiency.

We tested the GPU and CPU performances using the new algorithm. The result can be found in the Table below. The GPU runs much faster than CPU.

Computed cells	CPU running time (s)	GPU running time (s)
100 × 100	10.238405	0.671983
200 × 200	24.107771	1.561842
500 × 500	93.121168	4.91796
1000 × 1000	294.953172	17.452599
1500 × 1500	601.894	38.798
2000 × 2000	1036.129096	86.8301

Table 5.5.2.2: Performance comparison between CPU and GPU with the new algorithm.

Simultion accuracy comparison of the four situations is given below in Figure 5.5.2.2. The location of the field observation is at the center of the cavity.

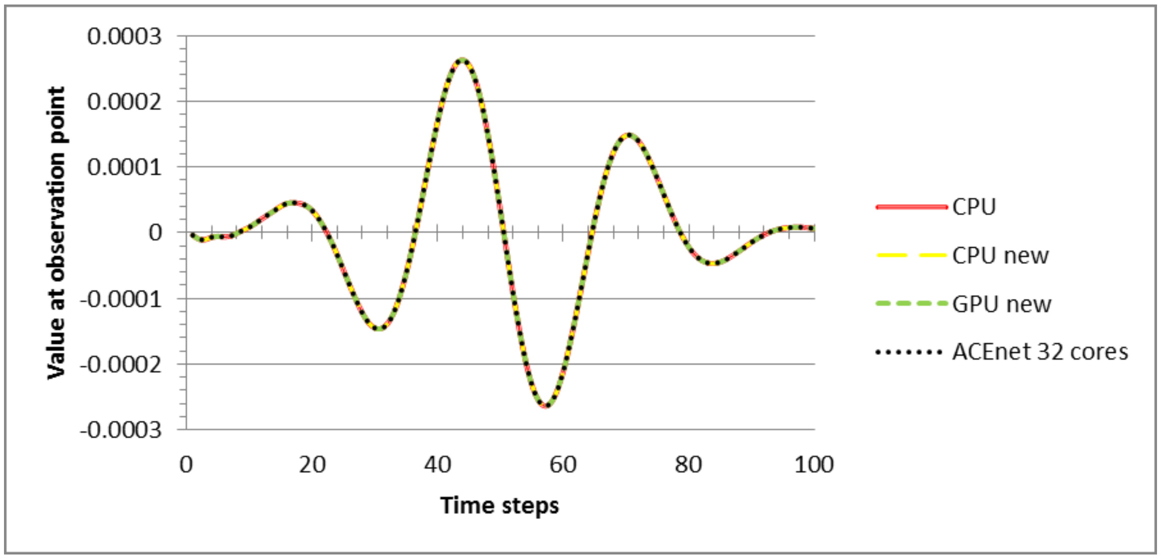


Figure 5.5.2.2: Plot of simulation accuracy comparison.

Summarizing all the run time results, we obtain Figure 5.5.2.3 (with 100 time steps):

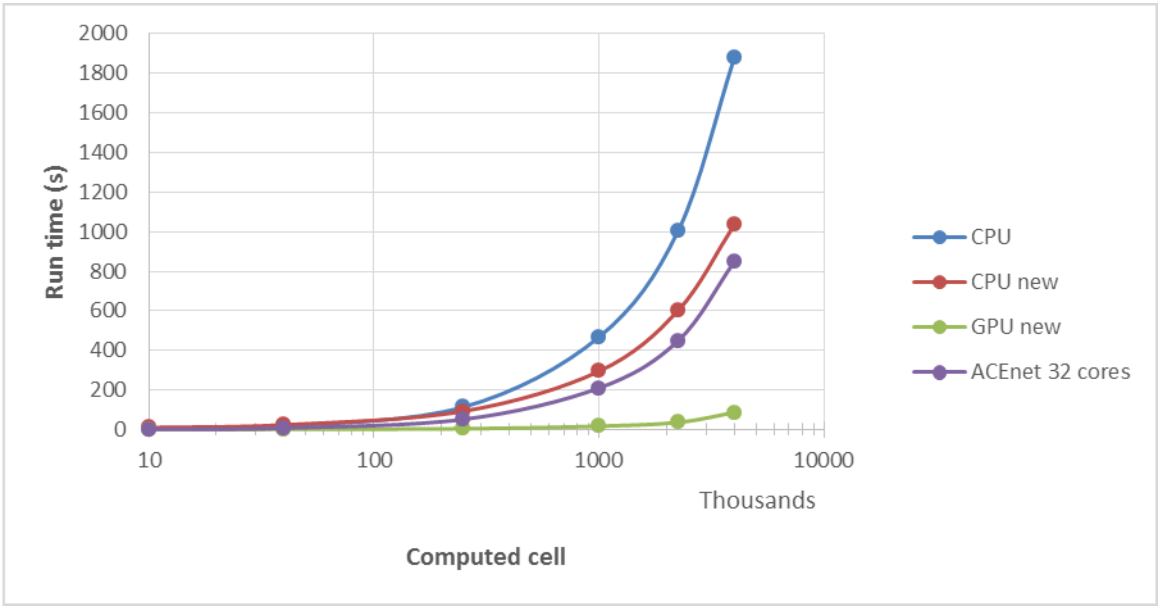


Figure 5.5.2.3: General speed-up comparison.

The above figure shows the substantial difference that parallel computing has made on FDTD modeling. If the problem size is smaller than 40,000, the elapsed time we measured is mainly for networking and bookkeeping. When a problem size becomes large, they become negligible; most of the time is used for the FDTD calculation. The speed-up efficiency can be seen in Figure 5.5.2.3:

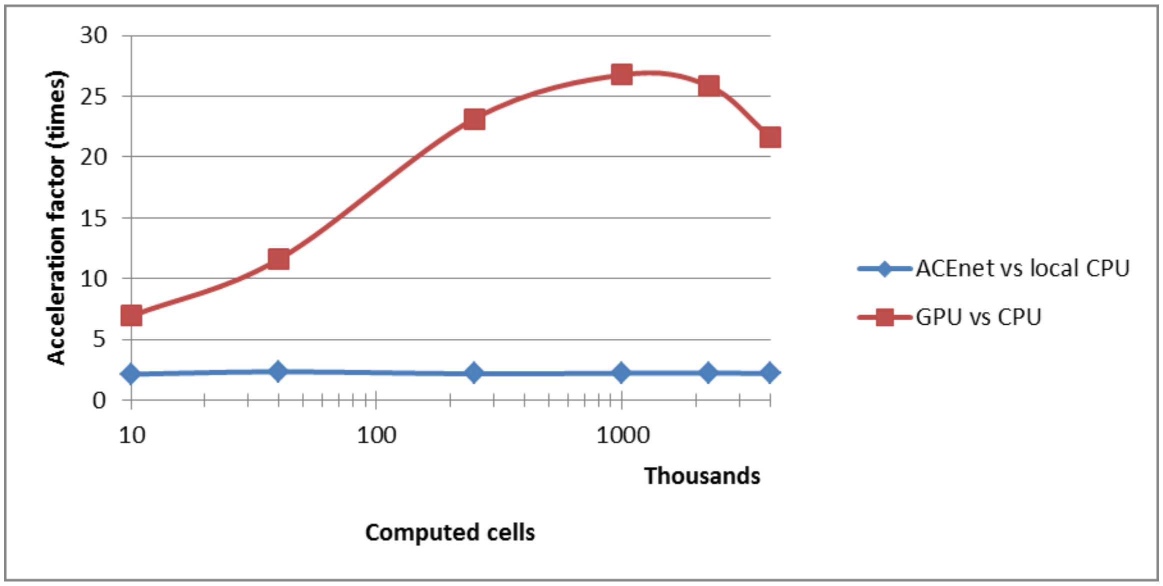


Figure 5.5.2.4: Speed-up tendency with increasing computing scales.

Below are the computing hardwre specifications:

Local computer:
Memory (RAM): 10.0 GB
CPU: Dual Intel Core i7-2600 @3.40 GHz
GPU: GeForce GT520

ACEnet basic information:
Maximum cores used: 32
CPU: Dual-Core AMD Opteron 2.6 GHz (cl001-cl012), Dual-Core AMD Opteron 2.8 GHz (cl021-cl030), Dual-Core AMD Opteron 3.0 GHz (cl056-cl107), Quad-Core AMD Opteron 2.4 GHz (cl108), Quad-Core AMD Opteron 2.7 GHz (cl135-cl266)

For the GPU acceleration, as the computing scale rises, our acceleration rate is also boosted, especially when the number of cells is greater than 40,000, while the ACEnet acceleration keeps still. This shows that the GPU parallel computing has a good performance for our method with suitable algorithms: it is 26.77 times faster than the CPU computing.

CHAPTER 6 CONCLUSIONS

6.1 CONCLUSIONS

The thesis finds that the FDTD method is well suitable to run on parallel architectures such as CUDA GPU. A wide range of methods have been implemented such as C, CUDA C, Matlab GPU, and ACEnet. On the other hand, there are certain situations that GPU cannot handle; they include problems that are generally small or unparallelizable. Very small problems lack the parallelism needed to use all the threads on the GPU. Unparallelizable problems have too many dependent branches, which can prevent data from efficiently streaming from GPU memory to the cores or reduce parallelism. Examples of these kinds of problems include: most graph algorithms, small signal processing problems, searching, sorting, and so on. In our case, the FDTD is usually not a small problem and is suitable to run on the parallel architecture that uses GPU. Generally speaking, GPU can handle a lot of intensive computing, and the effect could be very significant.

If a problem requires frequent transferring intermediate results between GPU and CPU, it could lead to a limited speed-up performance. In our work, a new computation algorithm has been developed to improve the situation. The new algorithm uses function invoking and matrix right-side-divide, optimizing the solution of tridiagonal equations and operations of for-loops. This removes the for-loop operation in the program. As a result, a speed-up of 26.77 times has been obtained.

The new algorithm was also implemented on ACEnet which is a multi-CPU system. The result shows that it is much slower than GPU. With more than 25 times faster with GPU, it means that instead of waiting an hour for the CPU to complete a simulation, it can now be done in about 2.5 minutes. Furthermore, this may be further improved, since the GPU cards used in this work are not best-configured for scientific computing. Off-the-shelf versions like GT 520 are more affordable and cost-saving. It should also be noted that computational capabilities of GPUs still improve at a fast pace, and CUDA framework is also consistently being developed. With these improvements, it is almost certain that programming on GPU will become easier and higher speed-ups will be achieved in the near future.

More importantly and significantly, multi-platform of the parallel implementation has been and will be tried. The acceleration in runtime has made many investigations possible and will also pave the way for other studies of large-scale computational electromagnetic problems involving big data that were previously impractical. This is a field that will definitely attract future in-depth research.

6.2 FUTURE WORK

This thesis explored the capabilities and prospects of using GPU as an alternative to CPU. The implementation of the GPU program was relatively straightforward, and there is still a lot of potential work that can be done, such as new algorithms with CUDA C to make it possible for further parallelization.

Moreover, with more powerful GPU cards for scientific computing like NVIDIA Tesla or Quadro commercially available, a better optimized GPU program can be developed. In addition, although ACEnet now only provides 40 workers that can work for Matlab, it may come with thousands of cores in the future. It can also be used for future large-scale computation.

BIBLIOGRAPHY

[1] NVIDIA Zone CUDA in Action, http://www.nvidia.com/object/cuda_in_action.html, and http://www.nvidia.com/object/cudaappsflash-ew.html#.

[2] CUDA FAQ, https://developer.nvidia.com/cuda-faq.

[3] S. E. Krakiwsky, L. E. Turner, and M. M. Okoniewski, “Graphics Processor Unit (GPU) Acceleration of Finite-Difference Time-Domain (FDTD) Algorithm,” Proc. 2004 International Symposium on Circuits and Systems, vol. 5, pp. V-265–V-268, May 2004.

[4] S. E. Krakiwsky, L. E. Turner, and M. M. Okoniewski, “Acceleration of Finite-Difference Time-Domain (FDTD) Using Graphics Processor Units (GPU),” 2004 IEEE MTT-S International Microwave Symposium Digest, vol. 2, pp. 1033-1036, June 2004.

[5] S. Adams, J. Payne, and R. Boppana, “Finite Difference Time Domain (FDTD) Simulations Using Graphics Processors,” Proceedings of the 2007 DoD High Performance Computing Modernization Program Users Group (HPCMP) Conference, pp. 334-338, 2007.

[6] V. Demir and A. Elsherbeni, “Compute unified device architecture (CUDA) based finite-difference time-domain (FDTD) implementation,” Journal of the Applied Computational Electromagnetics Society (ACES) 25.4 (2010).

[7] CUDA C Programming Guide, http://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[8] C. Y. Ong, M. Weldon, S. Quiring, L. Maxwell, M. Hughs, C. Whelan, and M. Okoniewski, “Speed it up,” IEEE Microwave Magazine, vol. 11, no. 2, pp. 70-78, April 2010.

[9] D. Donno, A. Esposito, L. Tarricone, and L. Catarinucci, “Introduction to GPU computing and CUDA programming: a case study on FDTD,” IEEE Antennas Propag. Magazine, vol. 52, no. 3, pp. 116-122, June 2010.

[10] T. Nagaoka, and S. Watanabe, “A GPU-based calculationusing the three-dimensional FDTD method for electromagnetic field analysis,” Engineering in Medicine and Biology, 2010 Annual Int. Conf. of the IEEE, pp. 327-330, August-September 2010.

[11] J. Chi, F. Liu, E. Weber, Y. Li, and S. Crozier, “GPU accelerated FDTD modeling of radio -frequency field-tissue interactions in high-field MRI,” IEEE Trans. Biomedical Eng., vol. 58, no. 6, pp. 1789-1796, June 2011.

[12] M. Livesey, J. F. Stack, F. Costen, T. Nanri, N. Nakashima, and S. Fujino, "Development of a CUDA implementation of the 3D FDTD method," IEEE Antennas Propag. Magazine, vol. 54, no. 5, pp. 186-195, October 2012.

[13] NVIDIA Tesla Personal Supercomputing GPUs, http://www.nvidia.ca/object/personal-supercomputing.html.

[14] A. Taflove and S.C. Hagness, Computational Electrodynamics: The Finite-Difference Time-domain method 3rd edition Boston: Artech House Norwood, MA, 2005.

[15] K. Yee, "Numerical solution of initial boundary value problems involving Maxwell's equations in isotropic media." Antennas and Propagation, IEEE Transactions on 14.3 (1966): 302-307.

[16] P. B. Johns, "Application of the transmission-line-matrix method to homogeneous waveguides of arbitrary cross-section." Electrical Engineers, Proceedings of the Institution of 119.8 (1972): 1086-1091.

[17] E. Purcell and C. Pennypacker, "Scattering and absorption of light by nonspherical dielectric grains." The Astrophysical Journal 186 (1973): 705-714.

[18] L. Greengard and V. Rokhlin. "A new version of the fast multipole method for the Laplace equation in three dimensions." Acta numerica 6.1 (1997): 229-269.

[19] M. Jaswon, "Integral equation methods in potential theory. I." Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences 275.1360 (1963): 23-32.

[20] A. Hrerinikoff, Solution of Problems of elasticity by the framework method [J], Journal of Applied Mechanics, 1941, 8 (4): 169-175.

[21] D. McHenry, "A lattice analogy for the solution of plane stress problems", Journal of Institute of Civil Engineers, 1943, 21 (2): 59-82.

[22] R. Courant, "Variational Methods for the Solution of Problems of Equilibrium and vibrations", Bulletin of the American Mathematical Society, 1943, 49 (1): 1-23.

[23] Finite Element Method, http://en.wikipedia.org/wiki/Finite_element_method

[24] Richard H. Gallagher and his work, http://www.nndb.com/people/811/000169304/

[25] G. Strang and G. Fix, "An Analysis of The Finite Element Method". Prentice Hall. ISBN 0-13-032946-0.

[26] O. Zienkiewicz, R. Taylor; J. Zhu, The Finite Element Method: Its Basis and Fundamentals (Sixth ed.). Butterworth-Heinemann. ISBN 0750663200.

[27] J. Reddy, An Introduction to the Finite Element Method (Third ed.). McGraw-Hill. ISBN 9780071267618.

[28] Computational Electromagnetics, Method of moments (MoM) or boundary element method (BEM), http://en.wikipedia.org/wiki/Computational_electromagnetics

[29] K. Mei and J. Van Bladel, "Scattering by perfectly-conducting rectangular cylinders." Antennas and Propagation, IEEE Transactions on 11.2 (1963): 185-192.

[30] R. Harrington, Field computation by moment methods. Wiley-IEEE Press, 1993.

[31] R.Harrington and J. Harrington, Field Computation by Moment Methods. Oxford University Press, 1996.

[32] N. Engheta, W. Murphy and V. Rokhlin, "The fast multipole method (FMM) for electromagnetic scattering problems." Antennas and Propagation, IEEE Transactions on 40.6 (1992): 634-641.

[33] J. Song and W. C. Chew, "Fast multipole method solution of three dimensional integral equation." Antennas and Propagation Society International Symposium, 1995. AP-S. Digest. Vol. 3. IEEE, 1995.

[34] Boundary Element Method, http://en.wikipedia.org/wiki/Boundary_element_method

[35] K. L. Shlager and J. B. Schneider, "A selective survey of the finite-difference time-domain literature", Antennas and Propagation Magazine, IEEE, 37(4):39-57, 1995.

[36] David B. Davidson, Computational Electromagnetics for RF and Microwave Engineering. Second Edition, Cambridge University Press, 2010.

[37] L. M. Angelone, S. Tulloch, G. Wiggins, S. Iwaki, N. Makris, and G. Bonmassar. "New high resolution head model for accurate electromagnetic field computation." In ISMRM Thirteenth Scientific Meeting, page 881, Miami, FL, USA, 2005.

[38] Strengths of FDTD Modeling, http://en.wikipedia.org/wiki/Finite-difference_time-domain_method#Strengths_of_FDTD_modeling

[39] Gauss's flux theorem, http://en.wikipedia.org/wiki/Gauss's_law

[40] D. Sullivan, Electromagnetic simulation using the FDTD method. Wiley-IEEE Press, 2013.

[41] R. Calalo, J. Lyons, and W. Imbriale, "Finite difference time domain solution of electromagnetic scattering on the hypercube." Proceedings of the third conference on Hypercube concurrent computers and applications-Volume 2. ACM, 1989.

[42] K. Tatalias, and J. Bornholdt, "Mapping electromagnetic field computations to parallel processors." Magnetics, IEEE Transactions on 25.4 (1989): 2901-2906.

[43] W. Buchanan, William J., and N. Gupta, “Parallel processing techniques in EMP propagation using 3D Finite-Difference Time-Domain (FDTD) method.” Advances in Engineering Software 18.3 (1993): 149-159.

[44] N. Oguni and H. Aasai, “Estimation of parallel FDTD-based electromagnetic field solver on PC cluster with multi-core CPUs.” Advanced Packaging and Systems Symposium, 2008. EDAPS 2008. Electrical Design of. IEEE, 2008.

[45] C. Guiffaut and K. Mahdjoubi. “A parallel FDTD algorithm using the MPI library.” Antennas and Propagation Magazine, IEEE 43.2 (2001): 94-103.

[46] V. Varadarajan and R. Mittra. “Finite-difference time-domain (FDTD) analysis using distributed computing.” Microwave and Guided Wave Letters, IEEE 4.5 (1994): 144-145.

[47] W. Yu, Y. Liu, T. Su and N. Hunag. “A robust parallel conformal finite-difference time-domain processing package using the MPI library.” Antennas and Propagation Magazine, IEEE 47.3 (2005): 39-59.

[48] M. Su, I. El-Kady and D. Bader. “A novel FDTD application featuring OpenMP-MPI hybrid parallelization.” Parallel Processing, 2004. ICPP 2004. International Conference on. IEEE, 2004.

[49] Y. Liu, Z. Liang and Z. Yang. “A novel FDTD approach featuring two-level parallelization on PC cluster.” Progress in Electromagnetics Research 80 (2008): 393-408.

[50] K. Sano, Y. Hatsuda, L. WANG, and S. Yamamoto. “Performance Evaluation of Finite-Difference Time-Domain (FDTD) Computation Accelerated by FPGA-based Custom Computing Machine.” Interdisciplinary Information Sciences 15.1 (2009): 67-78.

[51] M. DeLorimier and A. DeHon. “Floating-point sparse matrix-vector multiply for FPGAs.” Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays. ACM, 2005.

[52] L. Zhuo and V. Prasanna. “Sparse matrix-vector multiplication on FPGAs.” Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays. ACM, 2005.

[53] J. Durbano and F. Ortiz. “FPGA-based acceleration of the 3D finite-difference time-domain method.” Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on. IEEE, 2004.

[54] H. Kawaguchi, Y. Fujita, and Y. Fujishima. “Improved architecture of FDTD/FIT dedicated computer for higher performance computation.” Magnetics, IEEE Transactions on 44.6 (2008): 1226-1229.

[55] A. Valcarc, G. Roche, A. Juttner, D. Lopez-Perez, and Z. Jie. “Applying FDTD to the coverage prediction of WiMAX femtocells.” EURASIP Journal on Wireless Communications and Networking 2009 (2009): 3.

[56] P. Sypek, A. Dziekonski, and M. Mrozowski. “How to render FDTD computations more effective using a graphics accelerator.” Magnetics, IEEE Transactions on 45.3 (2009): 1324-1327.

[57] S. Peng and Z. Nie. “Acceleration of the method of moments calculations by using graphics processing units.” Antennas and Propagation, IEEE Transactions on 56.7 (2008): 2130-2133.

[58] P. Sypek and M. Mrozowski. “Optimization of a FDTD code for graphical processing units.” Microwaves, Radar and Wireless Communications, 2008. MIKON 2008. 17th International Conference on. IEEE, 2008.

[59] Flynn's taxonomy, http://en.wikipedia.org/wiki/Flynn%27s_taxonomy

[60] Parallel computing, http://en.wikipedia.org/wiki/Parallel_computing

[61] NVIDIA CUDA C programming guide version 4.2, 2012.

[62] NVIDIA Fermi compute architecture white paper, 2009.

[63] O. Yen, M. Weldon, S. Quiring, L. Maxwell, M. Hughes, C. Whelan, and M. Okoniewski. Speed it up. Microwave Magazine, IEEE, 11(2):70-78, 2010.

[64] NVIDIA Tesla c2050/c2070 GPU computing processor at 1/10th the cost, 2010.

[65] V. Kumar, A. Grama, and N. Vempaty. “Scalable load balancing techniques for parallel computers.” Journal of Parallel and Distributed Computing 22.1 (1994): 60-79.

[66] N. Govindaraju, B. Lloyd, W. Wang, and M. Lin. “Fast computation of database operations using graphics processors.” Proceedings of the 2004 ACM SIGMOD international conference on Management of data. ACM, 2004.

[67] J. Dongarra, I. Foster, G. Fox, W. Gropp, and K. Kennedy. Sourcebook of parallel computing. Vol. 3003. San Francisco: Morgan Kaufmann Publishers, 2003.

[68] K. Hillesland and A. Lastra. GPG Floating-Point Paranoia, 2004.

[69] United States General Accounting Office. Patriot missile defense: Software problem led to system failure at Dhahran, Saudi Arabia, February 4, 1992.

[70] R. Sedgewick and K. Wayne. Floating Point, 2010.

[71] NVIDIA CUDA C best practices guide version 4.1, 2012.

[72] NVIDIA CUDA Compiler Driver NVCC, 2013.

[73] D. Komatitsch, D. Michéa, and G. Erlebacher. “Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA.” Journal of Parallel and Distributed Computing 69.5 (2009): 451-460.

[74] P. Martinsen, J. Blaschke, and R. Künnemeyer. “Accelerating Monte Carlo simulations with an NVIDIA graphics processor.” Computer Physics Communications 180.10 (2009): 1983-1989.

[75] T. Preis, P. Virnau, W. Paul, and J. Schneider. “GPU accelerated Monte Carlo simulation of the 2D and 3D Ising model.” Journal of Computational Physics 228.12 (2009): 4468-4477.

[76] A. Khajeh-Saeed, S. Poole, and J. Perot. “Acceleration of the Smith–Waterman algorithm using single and multiple graphics processors.” Journal of Computational Physics 229.11 (2010): 4247-4258.

[77] W. Liu, B. Schmidt, G. Voss, and W. Müller-Wittig, “Accelerating molecular dynamics simulations using Graphics Processing Units with CUDA.” Computer Physics Communications 179.9 (2008): 634-641.

[78] E. Lezar, and D. Davidson. “GPU-accelerated method of moments by example: monostatic scattering.” Antennas and Propagation Magazine, IEEE 52.6 (2010): 120-135.

[79] ACEnet: Accelerating Discovery, http://www.ace-net.ca/wiki/ACEnet

[80] What Will ACEnet Do, http://www.atlanticuniversities.ca/research-initiatives/acenet

[81] ACEnet Cluster Status, http://www.ace-net.ca/wiki/Cluster_Status

APPENDIX A Specifications of GeForce GT 520 Graphics Card

GPU Engine Specs	
CUDA Cores	48
Graphics Clock (MHz)	810
Processor Clock (MHz)	1620
Texture Fill Rate (billion/sec)	6.5
Memory Specs	
Memory Clock	900 MHz (DDR3)
Standard Memory Configuration	1024 MB (DDR3)
Memory Interface	DDR3
Memory Interface Width	64-bit
Memory Bandwidth (GB/sec)	14.4
Feature Support	
OpenGL	4.2
Bus Support	PCI-E 2.0 x16
Supported Technologies	DirectX 11, CUDA, PhysX
Display Support	
Multi Monitor	Yes
Maximum Digital Resolution	2560×1600
Maximum VGA Resolution	2048×1536
Standard Display Connectors	Dual Link DVI-I HDMI VGA (optional)
HDMI	Yes
Standard Graphics Card Dimensions	
Length	5.7 inches
Height	2.7 inches
Width	Single slot
Thermal and Power Specs	
Maximum GPU Temperature (C)	102
Maximum Graphics Card Power (W)	29
Minimum System Power Requirement (W)	300
Legacy Specs	
Audio Input for HDMI	Internal

Source: http://www.geforce.com/hardware/desktop-gpus/geforce-gt-520/specifications

APPENDIX B Specifications of GeForce GTX 650 Graphics Card

GPU Engine Specs	
CUDA Cores	384
Base Clock (MHz)	1058
Texture Fill Rate (billion/sec)	33.9
Memory Specs	
Memory Clock	5.0 Gbps
Standard Memory Configuration	1024 MB (DDR3)
Memory Interface	GDDR5
Memory Interface Width	128-bit
Memory Bandwidth (GB/sec)	80.0
Feature Support	
OpenGL	4.3
Bus Support	PCI Express 3.0
Supported Technologies	3D Vision, CUDA, DirectX 11, PhysX, TXAA, Adaptive VSync, FXAA, NVIDIA Surround
Display Support	
Multi Monitor	4 displays
Maximum Digital Resolution	2560×1600 4096×2160
Maximum VGA Resolution	2048×1536
Standard Display Connectors	One Dual Link DVI-I, One Dual Link DVI-D, One Mini HDMI
HDMI	Yes
Audio Input for HDMI	Internal
Standard Graphics Card Dimensions	
Length	5.70 inches
Height	4.38 inches
Width	Dual-width
Thermal and Power Specs	
Maximum GPU Temperature (C)	98
Maximum Graphics Card Power (W)	64
Minimum System Power Requirement (W)	400
Supplementary Power Connectors	One 6-pin
3D Vision Ready	
3D Blu-Ray	Yes
3D Gaming	Yes
3D Photos	Yes

Source: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-650/specifications