ACCELERATING THE FDTD METHOD USING SSE AND GRAPHICS PROCESSING UNITS

A DISSERTATION SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF MASTER OF SCIENCE
IN THE FACULTY OF ENGINEERING AND PHYSICAL SCIENCES

2011

By
Matthew J Livesey
School of Computer Science

Contents

The total word count of this document is 43,200

List of Tables

List of Figures

Abstract

The Finite-Difference Time-Domain (FDTD) method is a computational technique for modelling the behaviour of electromagnetic waves in three-dimensional space. When executed to solve real-world problems the FDTD method is characterised by long execution times involving a large amount of data organised into matrices. The FDTD method exhibits ample data parallelism, and parallel computing techniques are frequently used to reduce the execution time of FDTD computations. This project explores the opportunities for accelerating the computation of the FDTD method using two types of commonly available parallel hardware.

First, the Streaming SIMD Extensions (SSE) capability of the x86 processor architecture is considered. The SSE capability of processors based on this architecture allow a single instruction to be applied to multiple sets of data simultaneously. This project investigates the possibility of using SSE to accelerate the FDTD method by performing calculations for multiple matrix elements with a single instruction.

Second, general-purpose computing on graphics processing units (GPUs) is considered. As the hardware used in GPUs has become less specialised, they have come to resemble general-purpose, massively parallel processors. Typical modern GPUs have hundreds of processor cores which operate concurrently, offering the potential to greatly accelerate computations with sufficient parallelism. This project investigates the possibility of accelerating the FDTD method using GPU computing.

Both hardware technologies are evaluated for the speedup they offer to the FDTD method, as well as the complexity of their implementation.

Declaration

No portion of the work referred to in this dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

I would primarily like to thank my project supervisor, Dr Fumie Costen, for her excellent support during this project. Her expertise and enthusiasm was indispensable. I could not have done it without her and it was a real pleasure to work with her.

I would like to thank Professor Seiji Fujino, Dr Takeshi Nauri and Dr Norimasa Nakajima at Kyushu University for providing access to their machine, which contained the hardware necessary to perform the GPU experiments undertaken in this project. I would like to thank Xiaoling Yang at 2Comu for access to his machine, which allowed the SSE experiments to be executed on an AMD Opteron processor.

I also want to thank Dr Len Freeman, Graham Riley, Dr Mikel Lujan and Professor Ian Watson for their teaching of the Multi-Core Computing pathway during my Masters study. Without their tuition I would not have been in a position to understand the concepts required for this project.

On a personal note, I would like to thank my family and Emma for their encouragement and patience. Their willingness to listen to me talk about esoteric elements of computing knows no bounds.

Chapter 1

Introduction

1.1 Project description

The Finite-Difference Time-Domain (FDTD) method, developed by Kane Yee in 1966, has become a popular method for solving electro-dynamic problems. The FDTD method is characterised by long computations over large amounts of data organised into matrices, where a small number of calculations are performed repeatedly for each element in each matrix. Due to significant data independence between the calculations for each point in each particular matrix, the FDTD method gives considerable opportunities for parallelism. The algorithms of the FDTD method form the basis of this project.

Single Instruction Multiple Data (SIMD) is a parallel programming paradigm where multiple data items are subject to the same machine instruction simultaneously. This is in contrast to the common Single Program Multiple Data (SPMD) paradigm. In SPMD, multiple instances of the program run in different threads or processes with each instance operating on a different part of the data. The program instances run independently from each other except where explicit synchronisation exists in the program. In a SIMD program, individual instructions are applied to multiple data items during a single instruction issue on the hardware.

Starting with the introduction of MMX technology in 1996, x86 architecture processors have included SIMD capabilities in hardware (MMX is an initialism with no official expansion). Current generations of x86 processors have an extended version of these capabilities known as Streaming SIMD Extensions (SSE). There are multiple generations of SSE, with each generation adding additional functionality, but in general SSE allows x86 processors to perform arithmetic operations on multiple sets of operands simultaneously in a SIMD manner. For example, two double-precision floating-point additions may be performed concurrently with a single instruction. This effectively doubles the throughput of the processor compared to issuing one instruction for each addition operation.

Recently, Graphics Processing Unit (GPU) hardware has become capable of running general purpose computations. As GPU hardware has evolved from fixed functionality pipelines to programmable massively parallel processing units, they have become an attractive target for scientific calculations. The term for this is General Purpose computing on Graphics Processing Units (GPGPU). GPUs have very high theoretical performance bounds for single-precision floating point arithmetic. More recent GPU hardware also has very high theoretical double-precision floating point performance. The actual performance achieved depends on the suitability of the algorithm for GPGPU, and the ability of the programmer to optimise the algorithm for the given hardware.

This project assesses both the performance improvements and the implementation complexity of applying both SSE hardware and GPU hardware to the FDTD method. There is benefit in reducing the execution time of applications of the FDTD method, both to increase the speed with which results are obtained for existing models and to allow larger, more complex situations to be modelled which would previously have been considered impractical due to the length of time required for execution.

1.2 Project scope

The starting point for this project is a simple Fortran based implementation of the FDTD method based on an existing implementation provided by the project supervisor. There are many aspects to the FDTD method which can differ based on the exact real-world scenario being modelled, and these aspects can increase or decrease the complexity of the implementation. Examples of variable aspects of the FDTD method include the way in which the source of the electromagnetic signal is modelled, the boundary conditions applied at the edges of each matrix, and the material being modelled. In this project, a basic form of the FDTD method is used including a hard-source signal at a single point, a Perfect Electric Conductor boundary condition, and the representation of the entire problem space as a vacuum. The algorithm being used in the initial Fortran implementation is described further in Section 3.1.

SSE instructions are used to optimise the calculations which implement the FDTD method equations and are repeated iteratively over each matrix. This is done by integrating routines written in C into the existing Fortran implementation, with the aim of minimising the amount of alteration required. The SSE implementation is executed on three different x86 architecture processors. The results show that the performance differs greatly on each processor, and the reasons for these differences are investigated.

The GPGPU based implementation is written from scratch using NVIDIA's CUDA technology. It is designed to implement exactly the same FDTD model as the Fortran and SSE based implementations. The CUDA implementation is developed through gradual improvement from a sequential approach to a multi-threaded approach using the two-level hierarchy of thread organisation used to express parallelism in CUDA. The GPGPU implementation is executed on both Tesla hardware (the original NVIDIA architecture for GPGPU) and Fermi hardware (a more recent architecture with specific improvements aimed at GPGPU). The GPGPU results are compared to both the original Fortran implementation and the SSE optimized implementation in order to assess the performance advantage offered by each.

1.3 Dissertation overview

This document is structured to broadly reflect the order in which the project progressed:

- Chapter 2 provides in depth background information on the FDTD method, SSE instructions, and the evolution of GPGPU. In each case, the key information from the literature is presented and the advantages and disadvantages are discussed.

- Chapter 3 describes the specifics of each implementation, including the reasons for choosing particular approaches. First, the details of the FDTD implementation used in this project are presented. Following this, the changes required to implement the same FDTD approach using SSE instructions and using CUDA technology for GPGPU computing are presented.

- Chapter 4 presents the performance results of the various implementations. Where interesting or unexpected results are observed, these results are explored in further detail. A number of different configurations including different hardware and different versions of each implementation are presented and compared.

- Chapter 5 summarises the project, in terms of the work performed and the findings observed from the experiments. Further developments which could build on the work in this project are discussed.

Chapter 2

Background and Literature

2.1 The Finite-Difference Time-Domain method

The ability to accurately model electromagnetic radiation has a wide range of technological applications. Examples include modelling the Earth's electromagnetic field in the domain of geophysics, detection of breast cancer using ultrawideband radar, and military defence [1].

Modelling of electromagnetic fields is done through calculating solutions to Maxwell's equations [1] [2] [3]. Maxwell's equations in an isotropic medium are [2] [3] [4]:

$$\nabla \times \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t} \tag{2.1}$$

$$\nabla \times \mathbf{H} = \frac{\partial \mathbf{D}}{\partial t} + \mathbf{J} \tag{2.2}$$

Where $\mathbf{E}$ is the electric field and $\mathbf{H}$ is the magnetic field. $\mathbf{D}$ is electric flux density, $\mathbf{B}$ is magnetic flux density, and $\mathbf{J}$ is the conduction current density [4]. Given the relationships $\mathbf{D} = \epsilon\mathbf{E}$ and $\mathbf{B} = \mu\mathbf{H}$ (where ϵ is permittivity and μ is permeability) and taking $\mathbf{J} = 0$ (as is the case in a source free medium) these equations can be rewritten as six partial differential equations, shown in equations 2.3 to 2.8 [4]:

$$\frac{\partial H_x}{\partial t} = \frac{1}{\mu}\left(\frac{\partial E_y}{\partial z} - \frac{\partial E_z}{\partial y}\right) \tag{2.3}$$

$$\frac{\partial H_y}{\partial t} = \frac{1}{\mu}\left(\frac{\partial E_z}{\partial x} - \frac{\partial E_x}{\partial z}\right) \tag{2.4}$$

$$\frac{\partial H_z}{\partial t} = \frac{1}{\mu}\left(\frac{\partial E_x}{\partial y} - \frac{\partial E_y}{\partial x}\right) \tag{2.5}$$

$$\frac{\partial E_x}{\partial t} = \frac{1}{\epsilon}\left(\frac{\partial H_z}{\partial y} - \frac{\partial H_y}{\partial z}\right) \tag{2.6}$$

$$\frac{\partial E_y}{\partial t} = \frac{1}{\epsilon}\left(\frac{\partial H_x}{\partial z} - \frac{\partial H_z}{\partial x}\right) \tag{2.7}$$

$$\frac{\partial E_z}{\partial t} = \frac{1}{\epsilon}\left(\frac{\partial H_y}{\partial x} - \frac{\partial H_x}{\partial y}\right) \tag{2.8}$$

Yee in 1966 introduced the Finite-Difference Time-Domain (FDTD) method for solving Maxwell's equations [2]. The FDTD method has become a popular approach to solving Maxwell's equations through computation [1]. Taflove 1998 and 2005 provide very comprehensive coverage of developments to the FDTD method since its original introduction [1] [5]. The fundamental basis of Yee's approach is to use finite-difference approximations of equations 2.3 to 2.8 [2] [1]. The space being modelled is discretized into a three-dimensional grid of points [2]. In Figure 2.1, the cube represents a single point in space following the discretization. The electric field ($\mathbf{E}$) components run parallel with edges of the cube, while the magnetic field ($\mathbf{H}$) components run perpendicular to the faces of the cube. This means that the $\mathbf{E}$ and $\mathbf{H}$ components of each point in the grid are interleaved such that each $\mathbf{E}$ value is surrounded by 4 $\mathbf{H}$ values and vice versa [2] [1]. For example, in Figure 2.1 H_z runs perpendicular to the top face of the cube, and is surrounded by 2 edges representing E_x and two edges representing E_y. Since the dimensions of the cube are one step in space in each axis, the distance between H_z and the edges representing E_x and E_y is one half of a step in space.

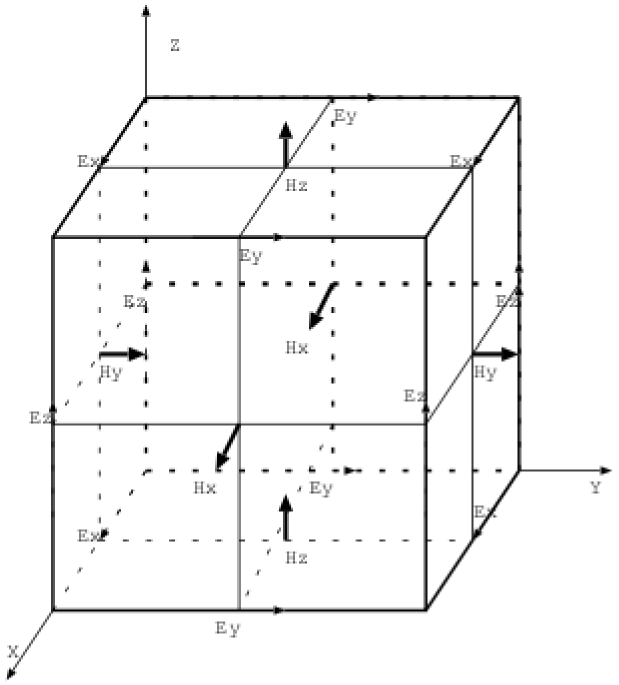


Figure 2.1: Yee's positioning of field components [2]

The values of $\mathbf{E}$ and $\mathbf{H}$ are calculated at half-time intervals, such that at each half-step in time either $\mathbf{E}$ is calculated from the previous values of $\mathbf{H}$ or vice versa [4] [1]. Discretization is performed with respect to time (Δt) and space (Δx, Δy and Δz) to yield equations 2.9 to 2.14 [4] [1].

$$\begin{aligned} H_x^{n+\frac{1}{2}}{}_{(i,j+\frac{1}{2},k+\frac{1}{2})} &= H_x^{n-\frac{1}{2}}{}_{(i,j+\frac{1}{2},k+\frac{1}{2})} \\ +\frac{\Delta t}{\mu(i,j+\frac{1}{2},k+\frac{1}{2})\Delta z} & \left[E_y^n{}_{(i,j+\frac{1}{2},k+1)} - E_y^n{}_{(i,j+\frac{1}{2},k)}\right] \\ +\frac{\Delta t}{\mu(i,j+\frac{1}{2},k+\frac{1}{2})\Delta y} & \left[E_z^n{}_{(i,j,k+\frac{1}{2})} - E_z^n{}_{(i,j+1,k+\frac{1}{2})}\right] \end{aligned} \tag{2.9}$$

$$\begin{aligned} H_y^{n+\frac{1}{2}}{}_{(i+\frac{1}{2},j,k+\frac{1}{2})} &= H_y^{n-\frac{1}{2}}{}_{(i+\frac{1}{2},j,k+\frac{1}{2})} \\ +\frac{\Delta t}{\mu(i+\frac{1}{2},j,k+\frac{1}{2})\Delta x} & \left[E_z^n{}_{(i+1,j,k+\frac{1}{2})} - E_z^n{}_{(i,j,k+\frac{1}{2})}\right] \end{aligned} \tag{2.10}$$

$$+\frac{\Delta t}{\mu(i+\frac{1}{2},j,k+\frac{1}{2})\Delta z}\left[E_x^n{}_{(i+\frac{1}{2},j,k)} - E_x^n{}_{(i+\frac{1}{2},j,k+1)}\right]$$

$$\begin{aligned} H_z^{n+\frac{1}{2}}{}_{(i+\frac{1}{2},j+\frac{1}{2},k)} &= H_z^{n-\frac{1}{2}}{}_{(i+\frac{1}{2},j+\frac{1}{2},k)} \\ +\frac{\Delta t}{\mu(i+\frac{1}{2},j+\frac{1}{2},k)\Delta y}&\left[E_x^n{}_{(i+\frac{1}{2},j+1,k)} - E_x^n{}_{(i+\frac{1}{2},j,k)}\right] \\ +\frac{\Delta t}{\mu(i+\frac{1}{2},j+\frac{1}{2},k)\Delta x}&\left[E_y^n{}_{(i,j+\frac{1}{2},k)} - E_y^n{}_{(i+1,j+\frac{1}{2},k)}\right] \end{aligned} \tag{2.11}$$

$$\begin{aligned} E_x^{n+1}{}_{(i+\frac{1}{2},j,k)} &= E_x^n{}_{(i+\frac{1}{2},j,k)} \\ +\frac{\Delta t}{\epsilon(i+\frac{1}{2},j,k)\Delta y}&\left[H_z^{n+\frac{1}{2}}{}_{(i+\frac{1}{2},j+\frac{1}{2},k)} - H_z^{n+\frac{1}{2}}{}_{(i+\frac{1}{2},j-\frac{1}{2},k)}\right] \\ +\frac{\Delta t}{\epsilon(i+\frac{1}{2},j,k)\Delta z}&\left[H_y^{n+\frac{1}{2}}{}_{(i+\frac{1}{2},j,k-\frac{1}{2})} - H_y^{n+\frac{1}{2}}{}_{(i+\frac{1}{2},j,k+\frac{1}{2})}\right] \end{aligned} \tag{2.12}$$

$$\begin{aligned} E_y^{n+1}{}_{(i,j+\frac{1}{2},k)} &= E_y^n{}_{(i,j+\frac{1}{2},k)} \\ +\frac{\Delta t}{\epsilon(i,j+\frac{1}{2},k)\Delta z}&\left[H_x^{n+\frac{1}{2}}{}_{(i,j+\frac{1}{2},k+\frac{1}{2})} - H_x^{n+\frac{1}{2}}{}_{(i,j+\frac{1}{2},k-\frac{1}{2})}\right] \\ +\frac{\Delta t}{\epsilon(i,j+\frac{1}{2},k)\Delta x}&\left[H_z^{n+\frac{1}{2}}{}_{(i-\frac{1}{2},j+\frac{1}{2},k)} - H_z^{n+\frac{1}{2}}{}_{(i+\frac{1}{2},j+\frac{1}{2},k)}\right] \end{aligned} \tag{2.13}$$

$$\begin{aligned} E_z^{n+1}{}_{(i,j,k+\frac{1}{2})} &= E_z^n{}_{(i,j,k+\frac{1}{2})} \\ +\frac{\Delta t}{\epsilon(i,j,k+\frac{1}{2})\Delta x}&\left[H_y^{n+\frac{1}{2}}{}_{(i+\frac{1}{2},j,k+\frac{1}{2})} - H_y^{n+\frac{1}{2}}{}_{(i-\frac{1}{2},j,k+\frac{1}{2})}\right] \\ +\frac{\Delta t}{\epsilon(i,j,k+\frac{1}{2})\Delta y}&\left[H_x^{n+\frac{1}{2}}{}_{(i,j-\frac{1}{2},k+\frac{1}{2})} - H_x^{n+\frac{1}{2}}{}_{(i,j+\frac{1}{2},k+\frac{1}{2})}\right] \end{aligned} \tag{2.14}$$

Note that due to the half stepping in both time and space the coordinates in equations 2.9 to 2.14 lie at non-integer values. In order to easily lay these out in a data structure for a computation, they are required to be integer values. This can be achieved by placing $E_x(0,0,0)$, $E_y(0,0,0)$, $E_z(0,0,0)$, $H_x(0,0,0)$, $H_y(0,0,0)$ and $H_z(0,0,0)$ as in Figure 2.2 [3]. The components of $\mathbf{E}$ and $\mathbf{H}$ are variously offset by half a step in the 3 axes of space in order to remove the fractional indices.

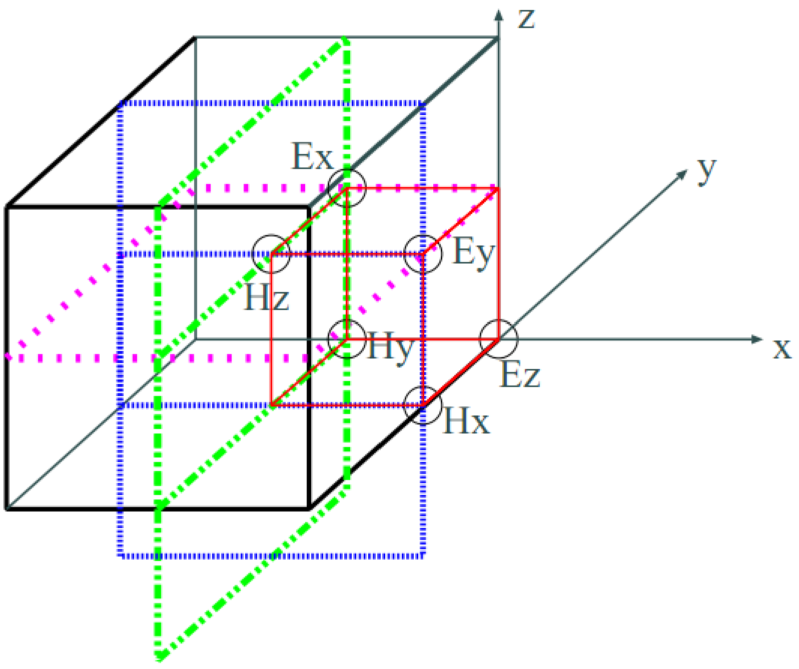


Figure 2.2: Placement of $E_x(0,0,0)$, $E_y(0,0,0)$, $E_z(0,0,0)$, $H_x(0,0,0)$, $H_y(0,0,0)$ and $H_z(0,0,0)$ to allow integer coordinates [3]

This offsetting yields equations 2.15 to 2.20 [4]. The basis of the FDTD method is iteratively solving each equation for each point in the grid, where each iteration over the entire grid represents a single time step.

$$\begin{aligned} H_x^{n+1}(i,j,k) = H_x^n(i,j,k) &+ \frac{\Delta t}{\mu(i,j,k)\Delta z}\left[E_y^n(i,j,k) - E_y^n(i,j,k-1)\right] \\ &- \frac{\Delta t}{\mu(i,j,k)\Delta y}\left[E_z^n(i,j,k) - E_z^n(i,j-1,k)\right] \end{aligned} \tag{2.15}$$

$$H_y^{n+1}(i,j,k) = H_y^n(i,j,k) + \frac{\Delta t}{\mu(i,j,k)\Delta x}\left[E_z^n(i,j,k) - E_z^n(i-1,j,k)\right] \tag{2.16}$$

$$-\frac{\Delta t}{\mu(i,j,k)\Delta z}\left[E^{n}_{x}{}_{(i,j,k)} - E^{n}_{x}{}_{(i,j,k-1)}\right]$$

$$H^{n+1}_{z}{}_{(i,j,k)} = H^{n}_{z}{}_{(i,j,k)} + \frac{\Delta t}{\mu(i,j,k)\Delta y}\left[E^{n}_{x}{}_{(i,j,k)} - E^{n}_{x}{}_{(i,j-1,k)}\right] - \frac{\Delta t}{\mu(i,j,k)\Delta x}\left[E^{n}_{y}{}_{(i,j,k)} - E^{n}_{y}{}_{(i-1,j,k)}\right] \tag{2.17}$$

$$E^{n+1}_{x}{}_{(i,j,k)} = E^{n}_{x}{}_{(i,j,k)} + \frac{\Delta t}{\epsilon(i,j,k)\Delta y}\left[H^{n}_{z}{}_{(i,j+1,k)} - H^{n}_{z}{}_{(i,j,k)}\right] - \frac{\Delta t}{\epsilon(i,j,k)\Delta z}\left[H^{n}_{y}{}_{(i,j,k+1)} - H^{n}_{y}{}_{(i,j,k)}\right] \tag{2.18}$$

$$E^{n+1}_{y}{}_{(i,j,k)} = E^{n}_{y}{}_{(i,j,k)} + \frac{\Delta t}{\epsilon(i,j,k)\Delta z}\left[H^{n}_{x}{}_{(i,j,k+1)} - H^{n}_{x}{}_{(i,j,k)}\right] - \frac{\Delta t}{\epsilon(i,j,k)\Delta x}\left[H^{n}_{z}{}_{(i+1,j,k)} - H^{n}_{z}{}_{(i,j,k)}\right] \tag{2.19}$$

$$E^{n+1}_{z}{}_{(i,j,k)} = E^{n}_{z}{}_{(i,j,k)} + \frac{\Delta t}{\epsilon(i,j,k)\Delta x}\left[H^{n}_{y}{}_{(i+1,j,k)} - H^{n}_{y}{}_{(i,j,k)}\right] - \frac{\Delta t}{\epsilon(i,j,k)\Delta y}\left[H^{n}_{x}{}_{(i,j+1,k)} - H^{n}_{x}{}_{(i,j,k)}\right] \tag{2.20}$$

Many FDTD problems are classified as open regions, meaning they extend indefinitely in the spatial domain [1]. A computer can only store and process a finite amount of data, and therefore the spatial domain needs to be limited [1]. The spatial domain is extended to the size necessary to enclose the object being modelled, and a boundary condition at the edges of this domain is used to simulate infinite size [1]. The purpose of the boundary condition is to suppress reflection of outgoing waves. Where the problem space is represented as matrices, this means manipulating the values at the faces of the matrices to give the results expected from infinite space. The boundary condition must simulate open space to the extent necessary to ensure that the FDTD computation produces results to the required level of accuracy for the given application [1]. The common term for such a boundary condition is an absorbing boundary condition (ABC) [3]. Material ABCs model the existence of a lossy material around the spatial domain to dampen outgoing waves [5]. This means padding the matrices in each direction with additional values, where the values of the padding are set to model the lossy material. A commonly used material ABC is the Perfectly Matched Layer (PML) boundary condition technique which is particularly effective at absorption and therefore at achieving high accuracy [5]. However, the PML technique requires a boundary of 6 - 10 layers of padding to be effective [3]. Adding extra layers increases the memory usage of the computation and this can be is undesirable in some cases [3]. An alternative is non-material ABCs that apply one-way wave equations at the boundary to prevent the reflection of outgoing waves [5] [3]. These do not require the additional boundary layers required by the PML technique but are not as effective at preventing reflection [5] [3]. There is a trade-off between minimising memory usage and achieving high accuracy when choosing a boundary condition approach.

In order for an FDTD computation to be valid, the size of the temporal step Δt and the spatial step Δs must satisfy the Courant Friedrichs Lewy (CFL) Condition [3]. Figure 2.3 shows instances where the condition is and is not met. The small circles represent the spatial grid where the distance between each circle is Δs. The larger circles show the propagation of a wave at time nΔt and (n+1)Δt (that is, a single step in time). In instance (a), the wave moves less that one spatial step for each time step. This results in a stable solution [3]. In instance (b), the wave moves more than one spatial step for each time step [3]. This results in an unstable solution. In general for a stable solution a computation must satisfy the CFL condition shown in equation 2.21 (where c represents the speed of light) [3].

$$\Delta t \leq \frac{1}{c}\left(\frac{1}{\Delta x^2}+\frac{1}{\Delta y^2}+\frac{1}{\Delta z^2}\right)^{-\frac{1}{2}} \tag{2.21}$$

Adding higher resolution to an FDTD computation can give more accurate results. A higher resolution means decreasing the value of Δs and therefore using more points to represent the same amount of space, which increases the memory usage and the amount of computation required. Additionally, due to the CFL condition, decreasing the value of Δs requires the value of Δt to be decreased to a value which satisfies equation 2.21 (where Δx, Δy and Δz are equal to Δs). This means more iterations are required to represent the same length of time in the computation, further increasing the computational cost. As a result, the FDTD method is characterised by large memory requirements and long execution times.

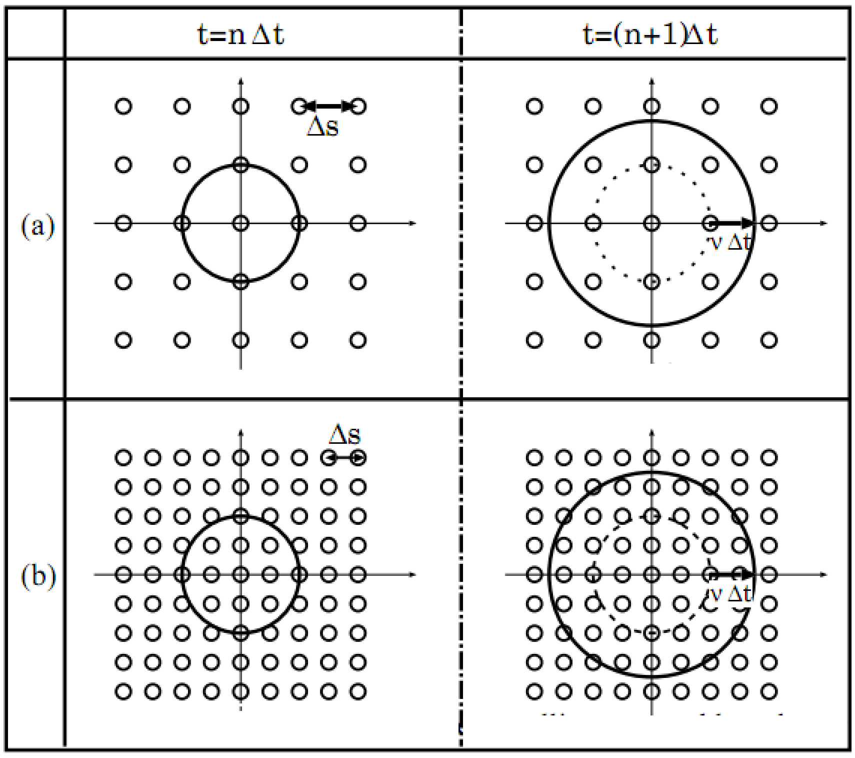


Figure 2.3: Representation of the Courant Friedrichs Lewy Condition [3]

The long execution times required to perform an FDTD computation can be mitigated by exploiting concurrency in the algorithm. Looking at equations 2.15 to 2.20, it can be seen that each component of **E** is dependent on itself and components of **H** but not on other values of **E**. Similarly, each component of **H** is not dependent on any other component of **H**. This means that all the components of **E** (an x, y and z axis value for every point in the spatial domain) can be independently calculated, and similar can be done with the components of **H**. Since the calculations are independent from one another they can be easily parallelised, with synchronisation only required when calculations for all of **E** or all of **H** have been completed. For example, Guiffaut and Mahdjoubi, 2001 present a parallel FDTD implementation using MPI (MPI stands for Message Passing Interface, and is a standard tool for performing parallel computation on distributed memory clusters) [6]. They show that on a spatial grid of 60 $\times$ 60 $\times$ 50, using 200 time steps, their method achieves an efficiency of greater than 80% with up to 20 processors, meaning a maximum speedup close to 18 compared to the execution time on a single processor [6]. This result supports the conclusion that the FDTD method is a good candidate for acceleration through parallel execution.

2.2 Streaming SIMD extensions

2.2.1 Introduction to SIMD

A conventional single processor architecture such as the Intel Pentium 4 can be described as “Single Instruction Single Data” (SISD) [7]. SISD processors have one stream of instructions and one stream of data, and as such execute in a sequential manner [7]. Conventional multiprocessor systems, including modern multi-core architectures, provide a number of processing units where each processing unit is able to execute its own instructions on its own data. A processing unit in this context may be a separate processor or one core in a multi-core processor, and is identified generically as the unit of hardware with its own instruction path and data path. This type of architecture can be described as “Multiple Instruction Multiple Data” (MIMD) [7]. It is possible for the different processing units in a MIMD machine to each execute a different program, as may be the case in a multi-tasking operating system. However in high performance computing it is common to use several processing units in a MIMD machine to execute different parts of the same program, using loops and branches to determine which part of the program is executed by which processing unit. This is described as “Single Program Multiple Data” and is a common parallel programming paradigm used in high performance computing [7]. Shared memory parallel computing with OpenMP and distributed memory parallel computing with MPI are common examples of the SPMD paradigm. Since 2003, conventional CPUs have increasingly been designed and built with multiple cores, to the point where today, the vast majority of conventional desktop and server CPUs are multi-core and capable of SPMD execution [8].

An alternative and less common architecture is to have multiple data streams with a single instruction stream. Multiple execution units within a processor use different data but are controlled by a single instruction issue unit and as such the execution units all execute the same instruction synchronously [7]. This is the ”Single Instruction Multiple Data” (SIMD) paradigm [7]. It allows very fine-grained data parallelism to be exploited, where multiple data items follow the same execution path in unison as a series of SIMD instructions are issued [7].

Figure 2.4 shows how a simple 4-element vector addition would be performed on an SPMD and a SIMD architecture. In the SPMD version two threads are created, each containing different instructions and operating on a different part of the data. The two threads can execute in parallel, but are independent of each other. Note that SPMD

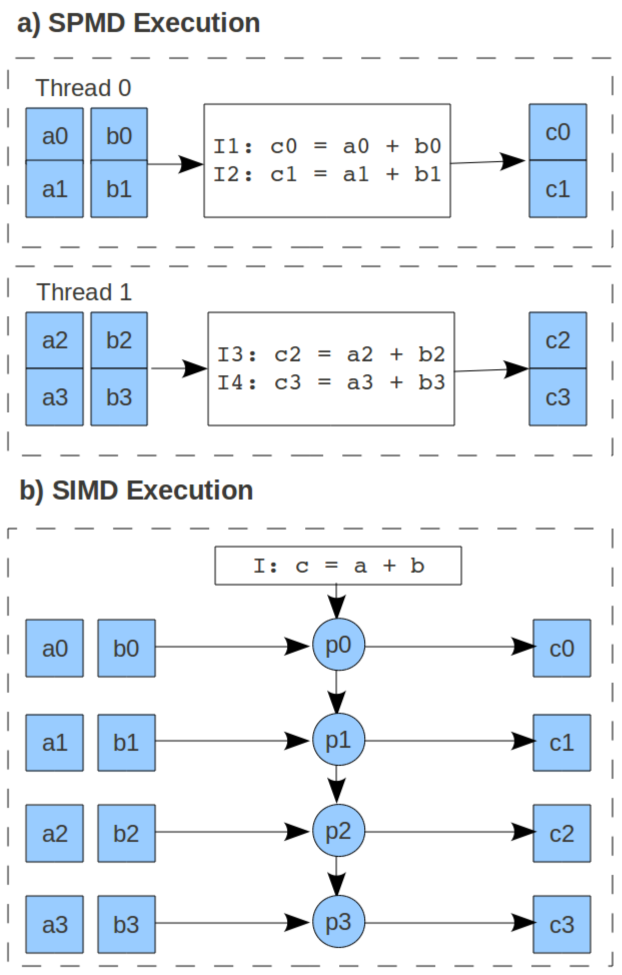
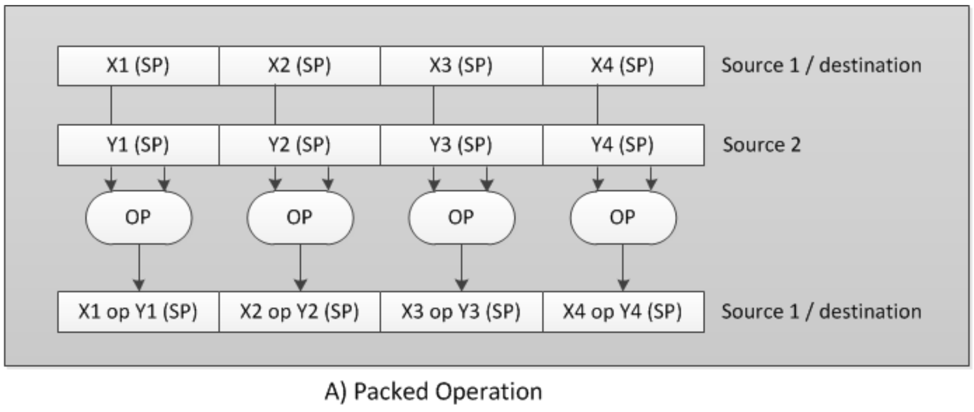


Figure 2.4: Comparison of SPMD and SIMD execution of a vector addition [9]

does not require that hardware is available to run the two threads in parallel. The threads are a software construct, and if only a single processing unit is available they can be executed sequentially. In the SIMD version, a single instruction is passed to 4 execution units, which perform the vector addition in parallel with each execution unit operating on a different part of the array. SIMD requires specific hardware support for issuing a single instruction synchronously to the multiple execution units, and so is less flexible than SPMD. However where hardware is available to support SIMD it may be more efficient than SPMD for the kind of fine-grained parallelism observed in problems such as vector addition. Software support for creating and managing threads can cost many CPU cycles, whereas SIMD is performed using native hardware instructions so software constructs such as threads are not required.

2.2.2 Capabilities of the SSE instruction set

Starting with the Pentium III in 1999, Intel introduced Streaming SIMD Extensions (SSE) to the x86 architecture [9]. Thakkur and Huff, 1999 [9] provide detail on the capabilities and motivations of SSE. SSE adds eight 128-bit wide registers to the architecture [9]. In CPUs supporting the first generation of SSE, each register holds four 32 bit floating-point values (single-precision floating-point numbers) [9]. The x86 instruction set is also extended to provide instructions which operate on the additional registers [9]. The arithmetic instructions provided with the first generation of SSE are addition, subtraction, multiplication, division, maximum, minimum, reciprocal, square-root and reciprocal square-root [9]. These instructions are available in scalar or packed form [9]. Figure 2.5 shows the difference between scalar and packed execution.



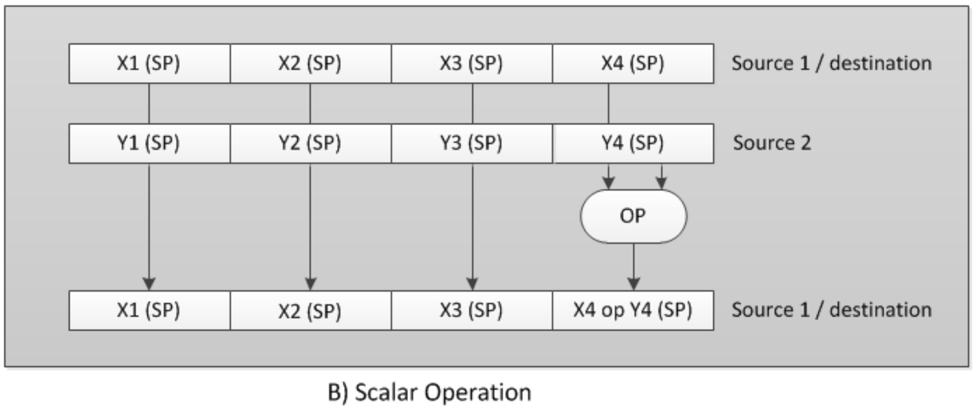


Figure 2.5: Scalar and packed execution of SSE operations [9]

In scalar form, the lowest 32 bits of each input register are used in the operation and the result is stored in the lower 32 bits of the output register. The other 96 bits of the output register are unaffected. In packed form, each 32 bit section of each input register is treated as a separate operand. The instruction is performed on each 32 bit section, meaning that 4 instances of the operation are performed on different data for a single instruction. The packed form of the SSE registers therefore provides a SIMD capability, since the single instruction issued is performed on 4 data items simultaneously and synchronously. Note that, as shown in Figure 2.5, the SSE instructions have a two operand format, meaning that the first input register is also the destination register.

Huynh, 2003 [10] describes how AMD introduced support for SSE starting with the Athlon MP processor architecture (MP is commonly interpreted as meaning 'Multi-Processor' to reflect the dual processor capability of the architecture). Prior to this, AMD had their own SIMD architecture extensions called "3DNow!" which were partially compatible with Intel's SSE architecture [10]. Athlon MP introduced an upgraded version, called "3DNow! Professional" which was fully compatible with the SSE instruction set [10]. From this point, developers were able to produce software optimised with SSE instructions and have these optimisations realised on both Intel and AMD processors. This effectively means that SSE is a ubiquitous technology on x86 architecture processors produced since 2003.

Thakkur and Huff state that "Streaming SIMD Extensions enable new levels of visual realism and real-time video encoding on the PC platform and also enhance the PC's video decode, speech recognition, and image manipulation capabilities" [9]. Similarly, AMD state on their website in relation to "3DNow!" that "This technology is designed to produce more realistic and lifelike 3D images and graphics, enhanced sound and video, and an extreme internet experience" [11]. These statements indicate the motivation for the original introduction of SSE was focused on multimedia applications. In fact, the ability to execute an instruction which operates on 4 sets of data simultaneously is also an attractive proposition for scientific algorithms such as the Finite-Difference Time-Domain method described in Section 2.1. Such applications perform a large number of arithmetic operations and exhibit a significant amount of data parallelism. However since the first generation of SSE provides only single-precision floating-point support, it is of limited use in the field of scientific applications where double-precision is usually required in order to achieve results with the necessary level of accuracy.

Starting with the release of the Pentium 4, Intel introduced a second generation of Streaming SIMD Extensions, commonly known as SSE2 [12]. Intel's technical information on the Pentium 4 [12] shows that SSE2 provides 144 new instructions which operate on the 128 bit registers. These instructions include support for double-precision floating-point values, and integer values of 8, 16, 32, 64 and 128 bits in size [12]. Figure 2.6 shows how the 128-bit registers can be divided differently to support different data types [13]. Floating-point arithmetic can be performed on the types labelled 'floats' and 'doubles', while integer arithmetic can be performed on the other data types [13]. SSE2 instructions therefore allow between 1 and 16 arithmetic operations with a single instruction, depending on the data type in use.

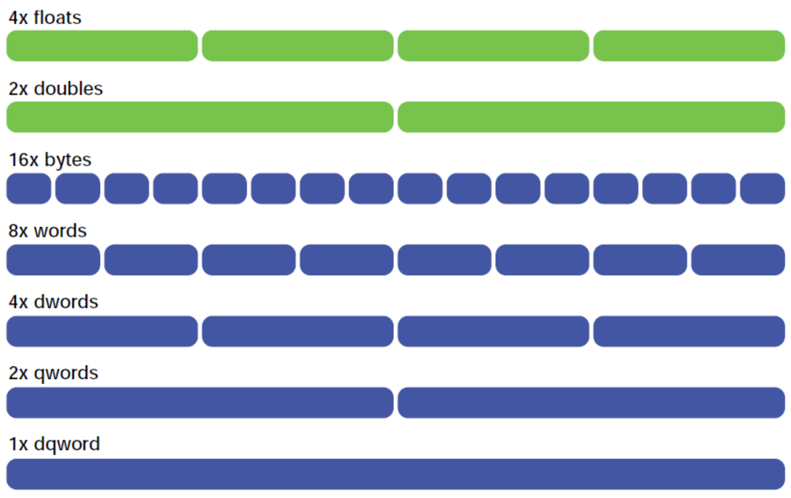


Figure 2.6: Data types supported by SSE2 instructions [13]

Since double-precision floating-point values are 64 bits in length, a single packed instruction can perform two double-precision floating-point operations at once. The same arithmetic instructions are available for packed double-precision floating-point values as with the original set of instructions for single-precision. Executing a packed SSE instruction on double-precision floating-point numbers effectively causes the hardware to act as a SIMD architecture with 2 execution units.

The Intel software developer's manual [14] provides detailed information on the subsequent generations of SSE, which gradually add additional instructions. For example, at the time of writing SSE is at version 4.2 in the newest Intel processors, which adds text-processing instructions [14]. Since the double-precision instructions added in SSE2 provide the required arithmetic operations for calculating the equations of the FDTD method this project focuses on the SSE2 instruction set. As well as identifying SSE2 as sufficient for the FDTD method, there is an additional advantage in limiting the scope of the project to SSE2. SSE2 is commonly available on both Intel and AMD processors, whereas SSE3 and above are less widely implemented. Therefore solutions based on SSE2 are more widely applicable, and allow the implementations presented in Chapter 3 to be executed on a wider range of hardware.

2.2.3 Issues with floating point arithmetic

With the introduction of SSE to the x86 architecture, processors based on this architecture have two execution paths for performing floating-point arithmetic. The original Floating Point Unit (FPU) of the x86 architecture is commonly referred to as the x87 FPU [14]. The x87 FPU and the scalar versions of the SSE floating point instructions perform the same function; the execution of arithmetic operations on two operands to a single result. Due to differences in the internal operation of the two floating-point paths, logically identical arithmetic expressions may not produce the same result.

Monniaux, 2008 provides a detailed study of the issues of floating point arithmetic [15]. The IEEE-754 standard for representing numbers in a floating-point format is universally adopted in modern processors [15]. In short, the bits used to represent a floating point value are split into a sign bit, a mantissa, and an exponent. The mantissa is multiplied by 2 to the power of the exponent to give the represented number. Therefore the size of the exponent determines the range of numbers available while the size of the mantissa determines the accuracy. That floating point representations cannot represent every number with exact precision and should only be treated as approximations is a well known issue [15]. One might expect that given a universal standard is in use, running the same algorithm on different hardware would produce the same approximation as the result. This is not always the case for a variety of reasons [15].

In the case of the x86 architecture, one reason for such discrepancies is that while both the x87 FPU and the SSE registers conform to the IEEE-754 64-bit double precision format when results are written to memory, they do not use the same representations internally [15]. The x87 FPU uses an 80-bit representation internally, with the results being reduced to 64-bit precision only when written back to memory [15]. Conversely, the SSE registers use a 64-bit representation at all times [15]. This means that if several arithmetic operations are combined within the registers of the processor before a final result is written to memory, this result may differ depending on the execution path taken. This is demonstrated with the code sample in Figure 2.7 which performs double-precision floating-point arithmetic on some arbitrary values, chosen to demonstrate the effect.

When compiled and executed on a machine with a 32-bit Intel Core 2 Duo processor running Linux using the Gnu Compiler Collection (GCC) with the command line

```
double x=  (1.23E−08+0.23)/0.45E16+ 1.12345;
double y= (0.34E−08+0.12)/0.12E16 + 1.5645 ;
double z= (x+y) / 7.654E−20;
printf("z:\t%22.19e\n",z);
```

Figure 2.7: Example of a computation which produces different results on different floating point paths

flag "-mfpmath=387" (which forces the use of the x87 FPU) the resulting output of the code in Figure 2.7 is:

3.5118238829370265600e+19.

When compiled again with GCC but with the flag "-mfpmath=SSE" (which forces the use of scalar SSE instructions) the resulting output is:

3.5118238829370261504e+19

The different internal operation of the CPU for the alternative floating-point paths leads to slightly different results. In a more complicated algorithm with thousands or millions of floating-point operations where previous results are reused in further calculations, the effect is likely to be magnified to produce differences of greater significance. This could cause a problem when producing an implementation of the FDTD method using packed instructions in the SSE registers. The correctness of any new implementation will usually be judged by comparing the results to those of an existing implementation. However these results may not be exactly the same if the existing implementation executes on the x87 FPU.

The matter is further complicated by the fact that on 32-bit x86 architectures, the x87 FPU is the default execution path for floating-point arithmetic, while on 64-bit x86 architectures, the SSE scalar instructions form the default execution path [15]. This means that comparing a new implementation to an existing one may show identical results on a 64-bit machine but show differences on a 32-bit machine. These complications were taken into account during the implementations and experiments conducted for this project, and Section 3.5 investigates the accuracy of the various solutions.

2.2.4 Existing experiments applying SSE to the FDTD method

Recent research by Yu et. al [16] shows the results of using SSE packed instructions to accelerate an FDTD application. While not explicitly stated in [16], it is observable from the code samples given and description of the method that single-precision floating point arithmetic is being used, and therefore each packed SSE instruction is performing 4 operations at once. Using a simple ideal test case, they find that the performance is close to 4 times faster when using the packed instructions [16]. However when a more complex, realistic scenario is used, speedup is reduced to around 1.5 [16].

No literature on using the double-precision functionality of SSE2 to accelerate the FDTD method was found. Since Yu et. al were able to speedup a test case by almost four times using single-precision instructions, it may be possible to produce a speedup of close to two using the double-precision instructions on a similar test case. This project will investigate the possibility of accelerating the performance of a simple double-precision FDTD implementation using the SSE registers on x86 architecture processors.

2.2.5 Advanced Vector Extensions (AVX)

Intel extended the SIMD capabilities of their processors with the introduction of Advanced Vector Extensions (AVX) to their architecture code-named Sandy Bridge [17]. The second generation of Intel Core processors were the first to be released based on Sandy Bridge, at the beginning of 2011 [17]. AVX uses 256-bit registers rather than the 128-bit registers used by SSE [17]. These are not new registers, but rather an increase in the size of the registers used for SSE [17]. The lower 128-bits of these registers are able to execute SSE instructions to support existing code [17]. AVX introduces instructions which allow all 256 bits to be used for SIMD operations. AVX therefore allows eight single-precision floating-point calculations or four double-precision floating-point calculations to be performed with a single instruction [17]. This is twice the throughput of SSE packed instructions. In addition, AVX improves on SSE by using a three-operand format, so the results of SIMD calculations do not need to be written over the top of one of the source operands [17]. AVX provides a similar capability to SSE (floating-point SIMD operations supported in hardware) but with increased throughput and additional flexibility.

AMD have included support for AVX in the design of their next generation architecture code-named Bulldozer [18]. At the time of writing there are no processors available based on Bulldozer, however they are expected to become available in September 2011 [19].

Over time, processors with AVX technology are likely to become increasingly common, as happened with SSE. Eventually the majority of x86 architecture processors in use will likely contain either Intel or AMD's implementation of AVX and it will be become as ubiquitous as SSE2 is today. However at the present time there are very few AVX capable processors available and AVX has not been considered for use in the implementations in this project. Due to the similarity between the operation of SSE and AVX, the techniques presented in Chapter 3 for applying SSE to the FDTD method could also be used with AVX. The increased throughput of AVX compared to SSE means it is reasonable to assume that an AVX based implementation of the FDTD method would provide higher performance than the SSE implementations developed in this project.

2.3 Computing with Graphics Processing Units

2.3.1 The evolution of GPGPU

The use of three-dimensional (3D) graphics applications for computing in professional environments, such as government, scientific visualisation, and cinematic effects production began in the 1980s [20]. During this time, the Silicon Graphics company produced specialised 3D graphics hardware aimed at the professional environment [20]. In the mid-1990s, the use of 3D graphics in computer games led to increasingly wide adoption of graphics accelerators in consumer computing [20]. Early consumer graphics processing units (GPUs) such as the GeForce 256 from NVIDIA were constructed from a fixed function pipeline consisting of vertex processors followed by pixel processors, reflecting the common sequence of operations performed by applications when rendering 3D graphics [21]. As GPU technology evolved over time, the vertex and pixel processors became increasingly programmable, to allow more and more powerful graphics applications to be realised [21].

The availability of programmable pipelines led to interest in implementing computations for purposes other than graphics on GPUs [20]. The high arithmetic throughput of GPU hardware meant that general purpose computations performed on GPU hardware had the potential to achieve high performance, making them attractive for computationally expensive tasks such as those often seen in scientific computing [20]. At the stage of programmable pipelines, the only way to program a GPU was to use a graphics API (Application Programming Interface) such as OpenGL in order to produce code which could interface with the graphics hardware [20]. Since such APIs focus specifically on rendering graphics, general purpose applications had to be expressed in terms of graphics constructs such as colours and pixels. In this way the GPU was effectively “tricked” into to executing a general purpose computation as if it was a graphics workload [20]. The expansion in popularity of general purpose computing on graphics processing units (GPGPU) was therefore held back by the requirement to learn a graphics API and by the restrictions on program behaviour enforced by such an API [20] [8].

Gradually, the functionality of vertex processors and pixel processors converged [21]. Due to the convergence, having separate dedicated hardware on a GPU for both pixel and vertex processing became undesirable. The overall design cost was greater than necessary, since the vertex processor and pixel processor were being separately designed while the result of those designs were becoming increasingly similar [21]. Additionally, applications which relied more heavily on one processor type than the other led to inefficiency in terms of hardware usage [21]. For example, in an application mainly using vertex processors, the pixel processors would be mostly idle despite being well-suited to performing the work of the application. Lindholm et. al [21] describe the design of NVIDIA's Tesla architecture, which features unified processors. The key parts of the Tesla design are shown in Figure 2.8. GPUs using the Tesla architecture are made up of an array of streaming multi-processors (marked SM in Figure 2.8), where each multi-processor contains a group of streaming processor (SP) cores. For example, the NVIDIA GeForce 8800 GPU has 16 multi-processors, each containing 8 cores, making 128 processor cores in total. The multi-processors are controlled by the work distribution units. There are separate work distribution units for vertex and pixel work, reflecting the fact that these share the same processing hardware resources in the Tesla architecture.

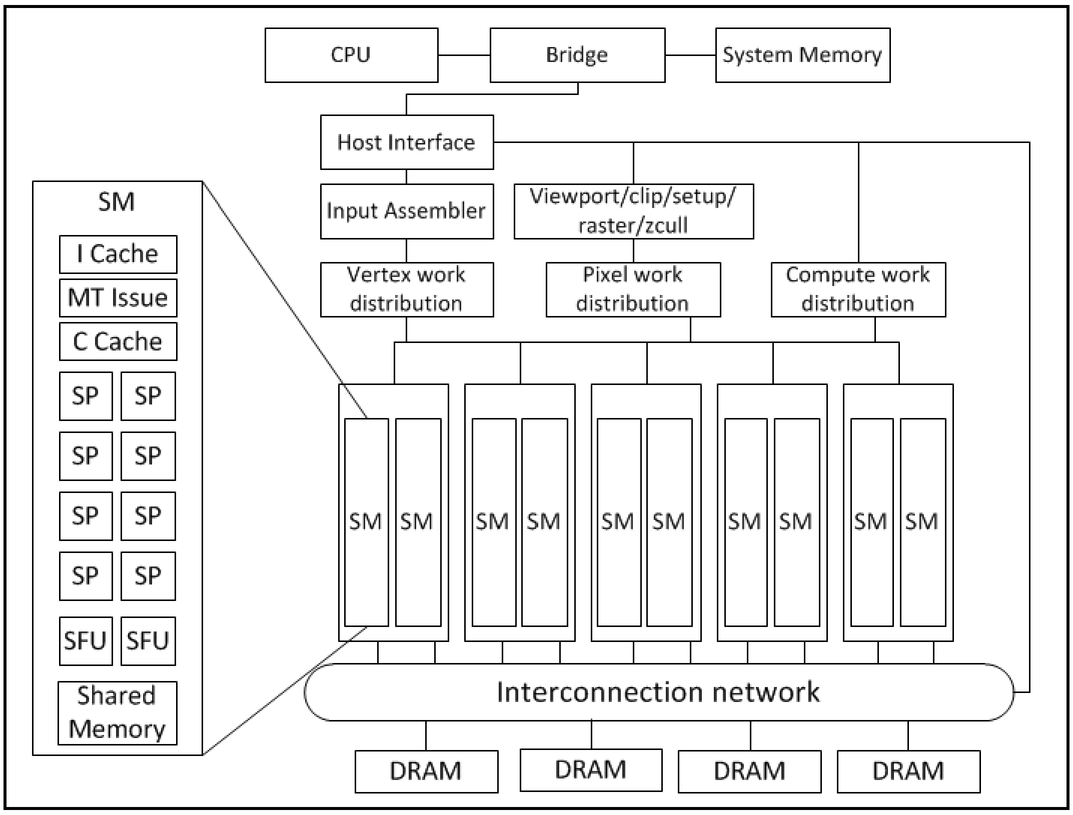


Figure 2.8: The Tesla unified graphics and computing GPU architecture [21]

In addition there is a compute work distribution unit for dispatching general purpose work to the array of streaming multi-processors. The addition of an architectural component dedicated to general purpose computing shows that this type of GPU usage was recognised as valuable by NVIDIA when Tesla was designed. Significantly, in conjunction with the introduction of the Tesla architecture, NVIDIA introduced the Compute Unified Device Architecture programming model (CUDA) [21] [20] [8]. CUDA is an extension to the C and C++ languages [21]. Note that while Lindholm et. al [21] refer to the hardware architecture as Tesla and the programming model as CUDA, the more recent literature from Kirk and Hwu [8] and Sanders and Kandrot [20] use the term CUDA more loosely to describe both the architecture and the programming model. For clarity, this document takes the approach of Lindholm et. al in using the terms Tesla and CUDA to describe the hardware and programming model respectively.

CUDA allows a programmer to specify which parts of a program should execute on the CPU as normal, and which parts can be executed on the GPU [21]. Work to be performed on the GPU is termed a 'kernel' [21]. Computations suitable for execution on a GPU are characterised by massive parallelism and a large number of arithmetic instructions. A CUDA kernel operates over a grid of data, where a grid is divided into blocks which are specified in either one or two dimensions. Each block is further divided into threads in one, two or three dimensions. The maximum dimensions of a grid and a block are hardware specific [20]. A common limit in current CUDA-capable GPUs is that each grid may not exceed 65,535 in each dimension, and each block is limited to 512 threads [20]. This gives a maximum of over 2 trillion threads for a single kernel, allowing for massively parallel execution.

The threads within a single block are grouped together in collections of threads called 'warps' [8]. A warp typically contains 32 threads [8]. All the threads within a warp are executed on a single streaming multiprocessor (SM), with each thread being executed on a different SP core. Since there are 32 threads per warp and 8 cores per SM, it takes 4 cycles for one warp to complete one instruction. Since an SM has one instruction issue unit (marked MT Issue in Figure 2.8) for all the SP cores it contains, the execution of a warp is similar to the single instruction multiple data (SIMD) paradigm described in Section 2.2.1. The threads within a warp proceed synchronously, so that each thread must finish an instruction before all threads are issued the next instruction. However the operation of threads within a warp is more flexible than typical SIMD operation. It is possible for threads to branch and execute different code paths from other threads in their warps [8]. When a thread is issued an instruction which is not part of its code path (for example an instruction within an IF block while the thread has branched to the ELSE block), that thread ignores the instruction. Therefore while thread execution within a warp is functionally able to support diverging code paths which SIMD cannot support, performance will be lost when divergence occurs since it leads to some cores being idle. This method of execution employed by the Tesla architecture is referred to as "Single Instruction Multiple Thread" (SIMT) [8].

In order to maximise performance and make the most of the multi-processing capability of a GPU, a programmer must be aware of the limitations of the architecture and the solutions for dealing with them. Kirk and Hwu [8] and Sanders and Kandrot [20] describe the limitations and CUDA best practices in detail. Perhaps the main factor affecting the performance of a GPU kernel is memory access efficiency [8]. For example, while an NVIDIA G80 GPU has a theoretical performance of 367 gigaflops (billion floating point operations per second), its global memory access bandwidth of 86.7 gigabytes per second limits memory throughput to 21.6 giga single-precision floating-point data per second [8]. Therefore if the number of arithmetic operations is equal to the number of global memory accesses, memory will be the limiting factor. The relationship between arithmetic operations and memory accesses is defined as the compute to global memory access (CGMA) ratio [8]. In general, increasing the CGMA ratio will increase the performance of a CUDA kernel [8]. In order to increase the CGMA ratio, the different memory types of the architecture need to be understood. Figure 2.9 shows the types of memory available in the Tesla architecture [8].

- Registers are allocated per-thread to store private scalar variables, and allow extremely fast access due to being on-chip [8]. Each streaming multi-processor has a limited number of registers to dynamically allocate to threads, so excessive use of registers will reduce the number of threads that can run simultaneously on the cores of a Streaming Multiprocessor which may reduce performance [8].

- Shared memory is also on-chip (as shown in Figure 2.8), allowing fast access, but is shared between all threads in a block [8]. This allows threads within the same block to cooperate and share data [8]. If threads cooperate to load commonly used data into shared memory, the overall number of accesses to global memory can be reduced, which can help increase the CGMA ratio [8].

- Constant memory is memory which is populated before a kernel is executed and cannot be updated during execution of a kernel. Values in constant memory can

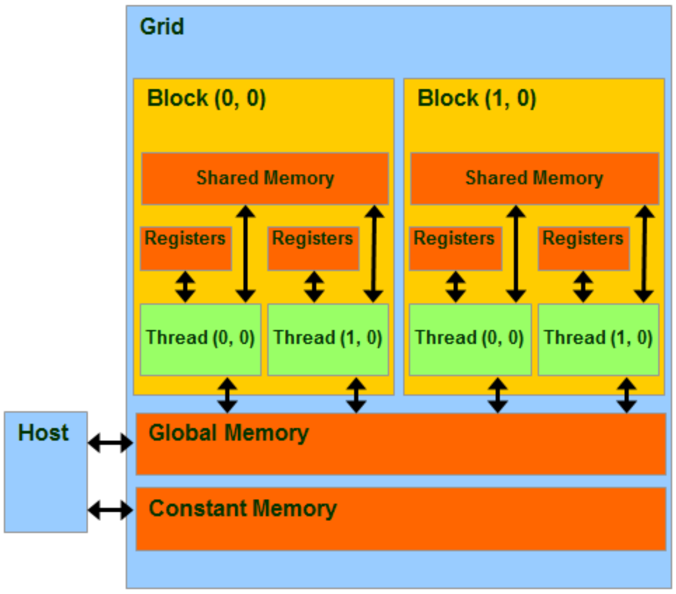


Figure 2.9: The Tesla device memory model [8]

be cached and broadcast to all threads in a block and therefore reduces memory bandwidth for certain patterns of memory access [20]. The constant cache hardware which makes this possible is marked "C Cache" in Figure 2.8. Where constant memory is used appropriately to store data commonly used by many threads, it can increase the CGMA ratio [8].

- Global memory is very large and is accessible to all threads in all blocks in a grid. It is shown in Figure 2.8 as DRAM and is connected to all the SM units. It has limited bandwidth so will limit performance if used too extensively [8]. It is possible to optimise the performance of access to global memory through "memory coalescing" [8].The threads within a warp execute synchronously within a single Streaming Multiprocessor. If they access consecutive memory locations at a particular instruction in the kernel function the individual memory requests are combined into a single larger request. Fetching a large, consecutive block of memory gives more efficient use of the memory bandwidth [8]. Kirk and Hwu [8] present a Molecular Visualisation example where rearranging the kernel function from an uncoalesced to a coalesced pattern of memory access increases the overall performance of the algorithm by about 20%.

- Local memory is private per thread but is stored in the global memory so has slow access times [8]. It is used for private array variables declared by a thread within a kernel function [8]. Kirk and Hwu [8] state that “From our experience, one seldom needs to use automatic array variables in kernel and device functions”. Where it is necessary to use private array variables within a kernel function, access time to the arrays will be comparable to the access time of global memory, but without the ability to optimise with coalescing.

Unlike CPUs, Tesla architecture GPUs do not have any layers of cache between the processors and the global memory. Therefore the programmer cannot rely on a cache to mitigate the affects of memory access time. Instead the programmer must reduce the reliance on main memory and increase the CGMA ratio through careful algorithmic design, maximising the use of registers, shared memory and constant memory.

The SP cores are multiply-add units, meaning they support basic arithmetic operations only. Each SM also includes two special functional units (marked SFU in Figure 2.8) to carry out operations such as trigonometric functions and reciprocals [21]. Since there are fewer SFUs than SP cores on each SM, using many operations requiring the SFU in a GPU kernel will reduce the parallel throughput of the SM and therefore reduce the performance of the kernel. The first generation of Tesla architecture GPUs only supported single-precision floating-point operations [22]. While this was corrected in the second generation of the architecture, double-precision performance was far below that of single-precision [22]. A more recent major revision to the architecture, named Fermi, introduces much more comprehensive double-precision support [22]. Fermi also introduces improvements in the memory system. The shared memory can be configured as partially user programmable and partially a level 1 cache [22]. A level 2 cache to global memory, unified across all SMs, is also introduced [22]. This means that some of the memory optimisation techniques required to achieve high performance with the first generations of CUDA capable GPUs may no longer be required. As a result, a programmer implementing a double-precision computation targeted at a GPU needs to be aware of exactly what level of hardware the computation will be executed on in order to know what sort of double-precision support will be available, and what sort of manual memory optimisation techniques in the code are justified. While CUDA kernels are functionally portable across different CUDA capable GPU hardware, the performance of a kernel designed for a particular GPU may not be matched when run on a different GPU.

While CUDA is a powerful and popular method for developing GPGPU applications, it has a drawback in that it is specific to NVIDIA hardware. As such, it cannot be used to develop applications for other vendors’ GPU hardware, or other types of massively parallel hardware. The Kronos Group, which is the cross-industry group which manages the OpenGL graphics library, has developed a cross-platform parallel computing API called OpenCL [8]. The purpose of OpenCL is to allow the development of parallel programs which run on heterogeneous hardware, including GPUs [8]. As such, OpenCL is an alternative to CUDA for developing GPGPU applications [8]. OpenCL is compatible with NVIDIA hardware, but also with hardware from other vendors such as AMD, who are NVIDIA's main competitors in the consumer graphics industry [8].

OpenCL has a data-parallel model which directly corresponds to the CUDA model [8]. OpenCL uses the term kernel to refer to a program which executes on a GPU, as with CUDA [8]. The concepts of "grid", "block" and "thread" from CUDA are termed "NDRange", "work item" and "work group" in OpenCL [8]. Functionally, the OpenCL model for programming a GPU is equivalent to that of CUDA and incorporates the same memory model [8]. Since OpenCL is designed to target a heterogeneous range of hardware, device management and starting kernels on devices is more complicated than with CUDA [8]. CUDA is able to hide some of the complexities of identifying and initialising devices since it is vendor specific, but with OpenCL the programmer must handle these tasks explicitly [8].

This project uses CUDA as the technology for developing GPGPU implementations, since NVIDIA hardware was available for the task, and CUDA is considered more mature and simpler than OpenCL. However it should be noted that the CUDA implementations presented in Chapter 3 could also be developed in OpenCL due to the correspondence between the programming models. The solutions devised in this project for executing the FDTD method on GPU hardware could be applied to devices from other vendors by re-implementing the solution in the OpenCL language.

2.3.2 Existing research into GPGPU computing

Amorim et al. 2009 [23] look at the difference between using the OpenGL graphics library and the CUDA method for accelerating the performance of computations. Their algorithm is a Jacobi Iteration in the domain of cardiac eletrophysiology, chosen due to its inherent parallelism [23]. The Jacobi Iteration involves several iterations over a matrix, with each element being updated based on its neighbours, so is somewhat similar to the FDTD method described in Section 2.1. [23] implements an OpenGL version using the pixel fragment processor of a programmable GPU to map the computation in terms of graphics constructs such that each element of the matrix is represented as a single pixel. [23] also produces a version of the algorithm in CUDA, and seeks to optimise it using coalesced memory access and shared memory. The results show that on less powerful GPU hardware, the OpenGL and CUDA versions perform roughly the same, but on more powerful hardware, the CUDA version provides better performance [23]. [23] also comments that the OpenGL implementation was more difficult to produce due to the reliance on understanding graphics programming [23]. The results from [23] support NVIDIA's own claims about the relative ease of use and performance benefits of using CUDA for GPGPU computing over the original method of using existing graphics APIs.

Che et al. 2008 use CUDA to develop and evaluate several computationally demanding applications [24]. Their chosen applications are traffic simulation, thermal simulation and k-means (a statistical method for cluster analysis). Compared to sequential CPU execution on a Pentium 4, the results show maximum speedups under CUDA of $40\times$ for traffic simulation, $6.5\times$ for thermal simulation, and $8\times$ for k-means on an NVIDIA Geforce 8800 GTX [24]. As well as demonstrating further applications for which GPGPU with CUDA can provide speedup, the results show how different the level of speedup is depending on the specifics of the application. Additionally, the results for traffic simulation show that for small problem sizes the CPU implementation outperforms the CUDA implementation, but this gradually reverses as the problem size is increased. The graph demonstrating this is shown in Figure 2.10.

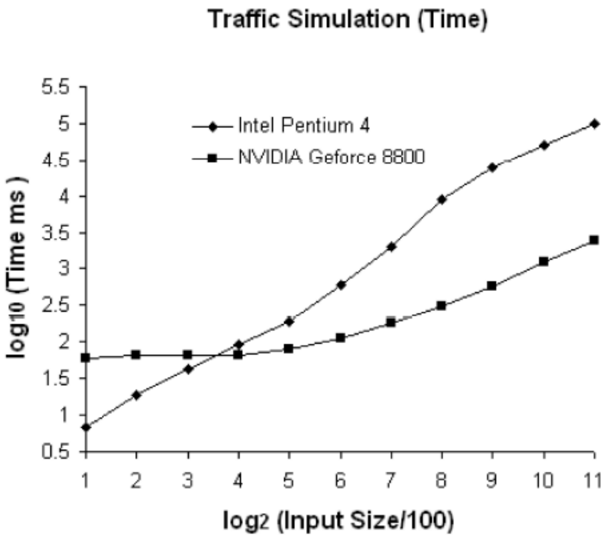


Figure 2.10: Results of traffic simulation from Che et al. [24]

Overall, [24] demonstrates that the level of performance increase which may be achieved from a CUDA implementation over a CPU implementation is dependent on both the characteristics of the application and the problem size over which the application is executed.

Yu et. al [16] was discussed in Section 2.2.4 in relation to using SSE to accelerate the FDTD method. The same paper also addresses the application of CUDA to the FDTD method. The results from [16] show that they are able to outperform a GPU with 240 cores using an SPMD parallel approach on 4 CPUs with SSE instructions (termed 'Vector Arithmetic Logic Unit' (VALU) in the paper). Figure 2.11 shows the performance results from [16].

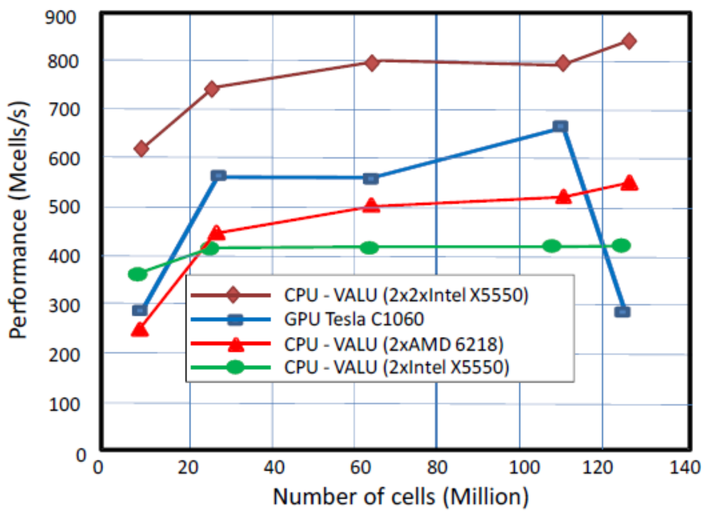


Figure 2.11: Performance of the FDTD method from Yu et al. [16]

As well as being outperformed by the best performing CPU system, the GPU results show a sudden drop off in performance once the problem size goes beyond 120 million cells (a cell refers to a single point in the three-dimensional problem space represented by **E** and **H** with values in the x,y and z direction, as described in Section 2.1). This implies that the FDTD method does not scale well when implemented in CUDA and executed on a GPU, and has negative implications for the suitability of GPU execution for accelerating the FDTD method.

Adams et al. [25] also presents the results of implementing a GPU version of the FDTD method. The results from [25] show that the FDTD method performs better on the GPU at larger problem sizes, since this results in all of the many cores on the GPU being utilised. [25] states that "Generally if the model space was large enough to keep the fragment processors busy, the GPU version was always several times faster, even for the slowest seven series GPU tested". It should be noted that [25] does not scale the experiments to a size which exceeds the memory capacity of a single GPU. In this circumstance, the costs of moving the data in and out of the GPU in partitions or the costs of communication between multiple GPUs would be incurred and the continually improving performance as the problem size increases is unlikely to be maintained.

The results from [16] and [25] are somewhat contradictory in that [16] suggests that the FDTD method does not scale well on a GPU and is easily outperformed by CPU implementations, whereas [25] suggests that it is only when scaled up that the FDTD method does perform well, and that it easily outperforms CPU implementations at larger scales. The differences between generations of GPU hardware and the issues of memory bandwidth and optimisation described in Section 2.3.1 may explain the discrepancies between these results. There are many different ways to implement the same algorithm on a GPU and many different types of hardware on which to execute the implementations. Section 3.4 describes the implementation of a CUDA version of the FDTD method, and investigates which optimisations can be used to maximise the performance on the available hardware. The performance results of experiments exploring the various optimisations are presented and discussed in Sections 4.3 and 4.4.

Chapter 3

Implementation Methods

3.1 Standard Fortran implementation

In order to judge the performance of the SSE and GPU implementations, a standard Fortran implementation of the FDTD method was produced. It was derived from an existing FDTD implementation provided by the project supervisor. The existing implementation required correction in places and was simplified in order to fit the scope of this project. This implementation represents each of E_x, E_y, E_z, H_x, H_y and H_z as a separate three-dimensional matrix where each element of each matrix is a double-precision floating-point number of 8 bytes. The dimensions of each matrix are denoted *i*, *j* and *k*. Therefore each point in the discretized representation of space is defined by 6 values $E_{x(i,j,k)}$, $E_{y(i,j,k)}$, $E_{z(i,j,k)}$, $H_{x(i,j,k)}$, $H_{y(i,j,k)}$ and $H_{z(i,j,k)}$. Each matrix requires a single cell of padding in each direction by way of a boundary condition. The values of each cell in the padding layer remain at zero throughout the execution. This emulates a boundary consisting of a Perfect Electric Conductor (PEC). A PEC causes perfect reflection at the boundaries of the problem space. As described in Section 2.1, there are a variety of boundary condition techniques available for use in FDTD implementations. Using a PEC boundary simplifies the implementation compared to more common techniques such as the Perfectly Matched Layer (PML). PML is an absorbing boundary condition which aims to model signals propagating infinitely into space. PEC is used here due to its simplicity, since the performance of boundary conditions is not the focus of this project, and adopting a simple option allows development to focus on other areas. Figure 3.1 shows the arrangement of one of these matrices. It represents a problem space of 4 cells in each dimension (shown in light grey) with a single cell of padding in each direction to form the PEC boundary (shown in dark grey), making a 6 × 6 × 6 matrix.

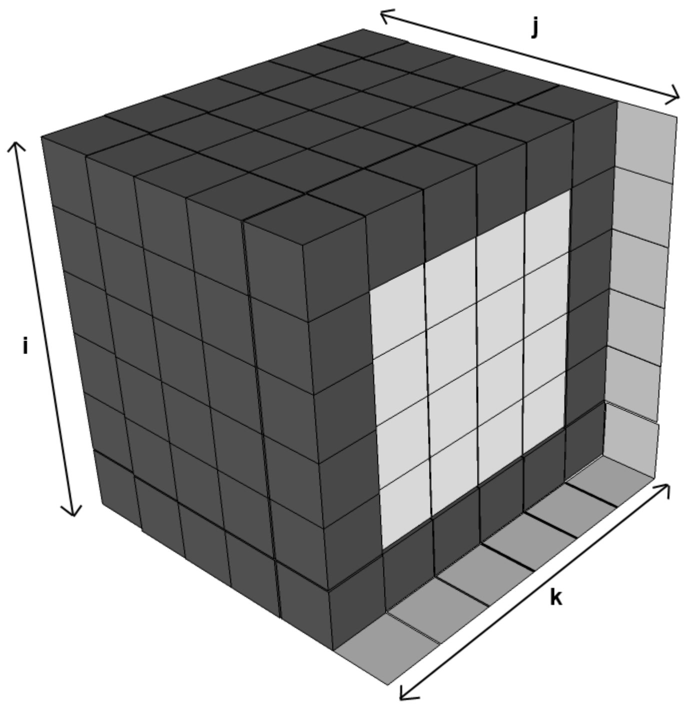


Figure 3.1: Representation of a matrix used in the standard FDTD implementation. The nearest face in the *j* dimension has been removed to show the internal cells.

Performing a single iteration of the FDTD method for the problem space shown in Figure 3.1 involves using equations 2.15 to 2.20 to update the values of the 64 internal cells (4 × 4 × 4) for each of E_x, E_y, E_z, H_x, H_y and H_z. The cells of the PEC boundary are not updated at any point but are referenced in some of the calculations. Due to the way Fortran arranges data structures in memory, a single increment in the *i* dimension means performing a single step in memory, so for example the cells $E_x(1,1,1)$ and $E_x(2,1,1)$ are adjacent in memory, with their addresses 8 bytes apart due to the size of a double-precision floating-point number. Similarly, a single increment in the *j* dimension performs *n* steps where *n* is the dimension of the problem space including padding, and a single increment in the *k* dimension performs n^2 steps. Therefore in the example shown in Figure 3.1, $E_x(1,1,1)$ and $E_x(1,2,1)$ are 6 places apart (48 bytes) and $E_x(1,1,1)$ and $E_x(1,1,2)$ are 36 places apart (288 bytes). As the matrices grow to more realistic problem sizes, a single increment in the *j* or *k* dimension means a large jump through memory. Standard practice when iterating over data structures is to single-step where possible, in order to make efficient use of cache[1].

To exploit single stepping, the algorithm solves the equations for each *(i,j,k)* index as shown in the code in Figure 3.2.

```
! Outer loop for number of iterations
do itimer = 1, itmax
! reversed for column major (efficient) access
do k = kmin +1, kmax -1
do j = jmin +1, jmax -1
do i = imin +1, imax -1
  ex(i,j,k) = ex(i,j,k) + dt/pmt*
   ((hz(i,j+1,k)-hz(i,j,k))/dy - (hy(i,j,k+1)-hy(i,j,k))/dz)
  ey(i,j,k) = ey(i,j,k) + dt/pmt*
   ((hx(i,j,k+1)-hx(i,j,k))/dz - (hz(i+1,j,k)-hz(i,j,k))/dx)
  ez(i,j,k) = ez(i,j,k) + dt/pmt*
   ((hy(i+1,j,k)-hy(i,j,k))/dx - (hx(i,j+1,k)-hx(i,j,k))/dy)
end do
end do
end do

!Maintain source value
ez((imax-2)/2,(jmax-2)/2,(kmax-2)/2) = -Jz(itimer)

! reversed for column major (efficient) access
do k = kmin +1, kmax -1
do j = jmin +1, jmax -1
do i = imin + 1, imax -1
 hx(i,j,k) = hx(i,j,k) -  dt/pma*
   ((ez(i,j,k)-ez(i,j-1,k))/dy - (ey(i,j,k)-ey(i,j,k-1))/dz)
 hy(i,j,k) =  hy(i,j,k) - dt/pma*
   ((ex(i,j,k)-ex(i,j,k-1))/dz - (ez(i,j,k)-ez(i-1,j,k))/dx)
 hz(i,j,k) = hz(i,j,k) -  dt/pma*
   ((ey(i,j,k)-ey(i-1,j,k))/dx - (ex(i,j,k)-ex(i,j-1,k))/dy)
end do
end do
end do
! end of iterations loop.
end do
```

Figure 3.2: Standard Fortran implementation of the FDTD method

The *i* dimension is controlled by the innermost loop, then *j*, then *k*, achieving the single-stepping approach. The variable *itimer* controls the number of times the FDTD method is applied to the problem space, thus controlling the length of time modelled by the algorithm. The values *imax*, *jmax* and *kmax* control the size of the problem space, and are set to the required dimension of the problem space plus 2 (to provide the padding of the PEC boundary). *imin*, *jmin* and *kmin* are set to 1. Therefore iterating from *imin+1* to *imax-1* and so on iterates over the entire problem space excluding the padding.

[1]Since populating a single cache line from memory loads several adjacent bytes, single-stepping minimises cache misses. Each step requests a data item from the same cache line as the previous step, except where the end of the cache line is reached.

Note that in equations 2.15 to 2.20, the values of permittivity (represented by ϵ) and permeability (represented by μ) are indexed by *(i,j,k)*. This allows them to vary at each point in the problem space to model a space containing variable materials. In the implementation used in this project, the entire problem space is modelled as a vacuum, so the values of permittivity and permeability are fixed for all points in the problem space. Permittivity is represented by *pmt* in the code above, while permeability is represented by *pma*. This further simplifies the implementation and reduces the overall amount of data required by the algorithm.

The values of E_x, E_y and E_z are updated in a separate nested loop from H_x, H_y and H_z in order to allow a single point of E_z in the middle of the problem space to be altered in between the calculations for $\mathbf{E}$ and those of $\mathbf{H}$ for each time increment. This is the statement in Figure 3.2 under the comment “Maintain source value”. When modelling a pulse at a single point to cause activity in an FDTD implementation, there is a choice between a hard-source and a soft-source [26]. The hard-source technique applies an electrical field directly in one direction at a single point, such as $E_{z(i,j,k)}$, where the values of *i*, *j* and *k* control the position of the pulse within the problem space [26]. When using a hard-source the values to be applied to $E_{z(i,j,k)}$ at each time step can be computed in advance. The soft-source technique models applying current at the point of the source, and requires the value of $E_{z(i,j,k)}$ to be calculated at each time step using a complex formula which relies on the current values of H_x and H_y [26]. The soft-source method is more complicated to implement than the hard-source method, although it produces better accuracy in some circumstances [26]. [26] shows that an example result from an FDTD execution modelling a vacuum in which the hard-source and soft-source produce near-identical results. The implementation used in this project uses the hard-source technique due to its relative lack of complexity, and the the observation from [26] that it can produce very similar results to the soft-source technique when modelling a vacuum. The step of updating the source value in between the calculations for the *E* matrices and the *H* matrices is a naturally sequential step which cannot be parallelised. Using a soft-source rather than a hard-source would be expected to increase the execution time of this sequential step due to the increased complexity, and would therefore reduce the overall performance benefit of parallelising other parts of the code. This should be considered with respect to the performance results in Chapter 4, since applying the same parallel programming techniques to implementations using a soft source may not experience the same benefit.

Since a hard-source is used to model the source signal, the values of the array *Jz* are pre-calculated before the FDTD algorithm begins. Jz represents an electromagnetic pulse, shaped as shown in Figure 3.3.

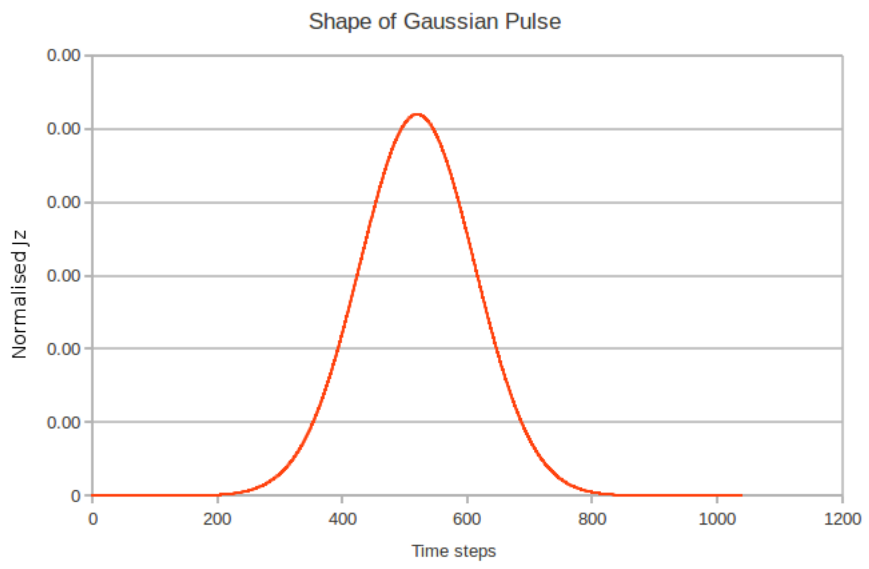


Figure 3.3: Shape of hard-source pulse used to excite the problem space

This standard Fortran implementation models the effect of applying this pulse to a point at the centre of the problem space. All further implementations aim to replicate this model while accelerating the performance of the calculations of E_x, E_y, E_z, H_x, H_y and H_z by exploiting the inherent parallelism available in the nested loops of the code sample above.

3.2 Implementation with SSE instructions in assembly

As described in Section 2.2, x86 architecture processors include Streaming SIMD Extensions (SSE) to allow an arithmetic instruction to be performed on multiple data items at once. This is achieved by populating special purpose registers in the processor with multiple data items and executing instructions using those registers. In general software development is performed at a higher level than using machine instructions on specific registers. A compiler is usually relied on to convert the high level code (such as Fortran) into the appropriate sequence of machine instructions. However in order to apply packed SSE instructions to the FDTD method the most direct method is to work explicitly at the lower level and directly specify the use of the SSE registers.

The C programming language provides support for inserting assembly language instructions into programs. Using the GCC compiler, the keyword *__asm__* specifies an assembly language sequence within a program [27]. [27] gives the code in Figure 3.4 as an example of the structure of an assembly sequence when using the GCC compiler.

```
int a=10, b;
        asm ("movl %1, %%eax;"
             "movl %%eax, %0;"
             :"=r"(b)          /* output */
             :"r"(a)           /* input */
             :"%eax"           /* clobbered register */
             );
```

Figure 3.4: Example of an assembly language sequence [27]

This simple example moves the contents of the register location representing variable *a* into the register *%%eax* and then moves the value from *%%eax* to the register location represented by variable *b*. The variable *a* is represented by the placeholder *%1* and *b* by the placeholder *%0* since *b* appears before *a* in the list of inputs and outputs. The example illustrates the following capabilities of an *__asm__* construct:

- Specific registers such as *%%eax* can be used in instructions when required.
- When it is not desirable to specify registers manually, variables from the C program such as *a* and *b* can be used. These variables are listed in the input and output statements and referenced in the assembly instructions as *%0*, *%1* and so on with the placeholder numbers corresponding to the order of the variables in the input and output lists.
- Where a specific register such as *%%eax* is used it is necessary to include it in the clobber list [27]. This list instructs the compiler that this register is altered by the assembly sequence so the compiler does not expect its value to preserved when generating instructions for the high-level language code surrounding the assembly sequence [27].
- If these rules are followed, complex assembly sequences can be interleaved with high level language sequences as required to produce correctly functioning programs.

In general, explicitly adding assembly sequences to code would be bad practice as it prevents the code from being portable. Whereas C code can be compiled on a wide array of systems, assembly language is specific to a particular architecture. However in this case, we are deliberately targeting x86 architecture processors capable of executing SSE instructions so the use of assembly is acceptable.

As described in section 2.2.2, there are eight special purpose registers for executing SSE instructions, each of 128 bits (16 bytes) in size. In assembly programming, they are named *%%xmm0* to *%%xmm7* [28]. There are a large number of instructions for manipulating these registers. These instructions are suffixed as 'pd' or 'ps' meaning packed double-precision or packed single-precision respectively. These suffixes indicate how the data in the registers should be interpreted. For example, the instruction *mmaddps %%xmm0 %%xmm1* would interpret the contents of each SSE register as being 4 single-precision numbers, and perform 4 add operations simultaneously. The instruction *mmaddpd %%xmm0 %%xmm1* would interpret the contents of each SSE register as 2 double-precision numbers and perform 2 add operations simultaneously. Since the assembly instructions are below the level of the compiler, such sequences are not protected from type-casting errors. Therefore if, for example, registers are loaded with single-precision variables and double-precision instructions are applied to those registers, the compiler will not catch this and unpredictable results will occur at run-time.

An important concept when using SSE instructions is that of byte alignment. Each SSE register is 16 bytes in size, and is most efficiently loaded to and stored from when using memory addresses which are aligned to a 16-byte boundary. Since each double-precision floating-point element in each matrix is 8 bytes in size, the elements in the matrix alternate between being aligned and not. So for example, if $E_x(1,1,1)$ is aligned, then $E_x(3,1,1)$ and $E_x(5,1,1)$ would also be aligned but $E_x(2,1,1)$ and $E_x(4,1,1)$ would not be (note that this is using standard Fortran convention, where indexing starts at 1). The SSE streaming implementations presented here are designed to be correct when the dimensions of the problem space are a multiple of 2. This gives the property that any increment or decrement in the index of a matrix in the *j* and/or *k* dimensions without altering the index in the *i* dimension is guaranteed to maintain alignment or lack of alignment to a 16 byte boundary. So, for example, if and only if $E_x(1,1,1)$ is aligned, $E_x(1,2,1)$, $E_x(1,1,2)$ and $E_x(1,2,2)$ are all also aligned. Loading into an SSE register loads the 16 bytes from the specified address. Therefore loading into an SSE register from $E_x(1,1,1)$ loads the 8-byte value of $E_x(1,1,1)$ into the lower 64 bits of the register and the 8-byte value of $E_x(2,1,1)$ into the higher 64 bits. It is possible to load on an unaligned address, but a separate, less efficient instruction is used to do this. By loading and operating on the values in pairs from aligned addresses, all the variables from each matrix can be accessed using aligned load instructions. To understand the pattern of memory access of each step of the FDTD algorithm, the code of the standard Fortran implementation can be inspected. This code is given in Figure 3.5.

```
ex(i,j,k) = ex(i,j,k) + dt/pmt*
  ((hz(i,j+1,k)-hz(i,j,k))/dy - (hy(i,j,k+1)-hy(i,j,k))/dz)
ey(i,j,k) = ey(i,j,k) + dt/pmt*
  ((hx(i,j,k+1)-hx(i,j,k))/dz - (hz(i+1,j,k)-hz(i,j,k))/dx)
ez(i,j,k) = ez(i,j,k) + dt/pmt*
  ((hy(i+1,j,k)-hy(i,j,k))/dx - (hx(i,j+1,k)-hx(i,j,k))/dy)
```

Figure 3.5: Code showing the memory access pattern of the Fortran implementation

The value of each matrix at a particular *(i,j,k)* index is updated based on values of other matrices for the same index and values from other matrices shifted in one of *i*, *j* or *k*. As described above, the shifts in the j and k dimensions maintain alignment. So in the code above, if each matrix is byte-aligned at *(i,j,k)*, only the accesses to $H_z(i+1,j,k)$ and $H_y(i+1,j,k)$ (highlighted in Figure 3.5) are unaligned. Out of 12 matrix accesses in the Figure 3.5, only the two highlighted will be unaligned if $E_x(i,j,k)$, $E_y(i,j,k)$, $E_z(i,j,k)$, $H_x(i,j,k)$, $H_y(i,j,k)$ and $H_z(i,j,k)$ are aligned. By double-stepping the value of *i* in the innermost loop, and therefore only considering the aligned *(i,j,k)* combinations, it can be ensured that 10 out of 12 of all memory accesses to the matrices are aligned. The double-stepping is acceptable because at each step the algorithm will now calculate the result for two elements of a matrix simultaneously, using the packed double-precision instructions. The operands for $E_x(1,1,1)$ and $E_x(2,1,1)$ are loaded and operated on together. Once this is complete, the algorithm makes a double-step, jumping to $E_x(3,1,1)$.

Ensuring 16-byte alignment at the start of the matrix in C code is straightforward when using the GCC compiler. For example the code in Figure 3.6 creates an array in the normal manner but ensures the beginning of the array is 16-byte aligned.

```
double __attribute__ ((aligned (16))) a [1000];
```

Figure 3.6: Achieving alignment in C when using the GCC compiler

Since there is a single element layer of padding around each matrix which is not operated on, the algorithm does not in fact start at first index (by default *(1,1,1)* in Fortran or *(0,0,0)* in C) for each matrix. With the standard Fortran implementation's single layer of padding, the first values to be calculated would be at *(2,2,2)*. If the matrices have been 16-byte aligned, addresses at *(2,2,2)* are guaranteed not to be aligned, which breaks the efficient memory access pattern. In order to rectify this, an extra layer of padding is added only in the *i* dimension. In the Fortran code indexing in the *i* dimension is altered to start at 0, so the first cell in the matrix is at *(0,1,1)*. This means that when calculations begin at *(2,2,2)* this location is aligned to a 16-byte boundary in memory. The structure of the matrices after this additional layer is added is shown in Figure 3.7.

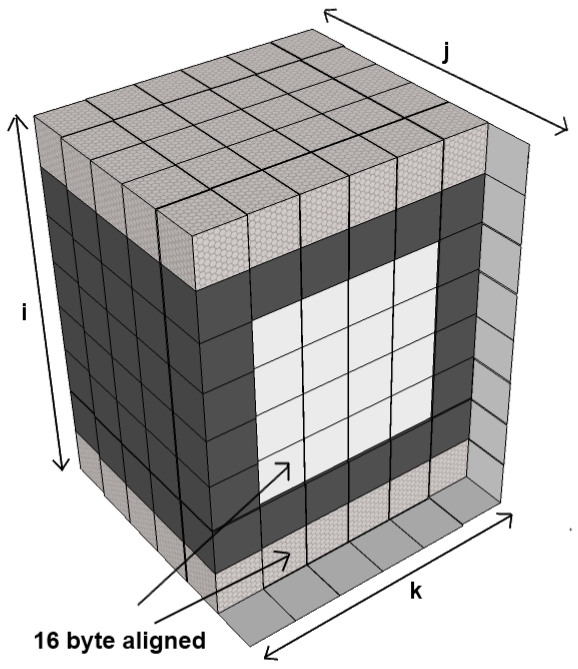


Figure 3.7: Representation of a matrix with additional padding in the *i* dimension to ensure alignment. The nearest face in the *j* dimension has been removed to show the internal cells.

A single line of the Fortran implementation expresses a sequence of arithmetic operations which require several machine instructions to complete. Consider the calculation for a value of E_x given in Figure 3.8.

```
ex(i,j,k) = ex(i,j,k) + dt/pmt*
  ((hz(i,j+1,k)-hz(i,j,k))/dy - (hy(i,j,k+1)-hy(i,j,k))/dz)
```

Figure 3.8: Calculation of E_x in Fortran

This line of code expresses 8 arithmetic operations. Each calculation of the FDTD method has the same 8 operation structure, differentiated only by differing variables and the replacement of subtractions with additions. The generic structure is given in Figure 3.9.

The values of *dx*, *dy* and *dz* are constant throughout the algorithm, as are the results

```
result = result + dt/(pmt or pma) * (
   (clause1_1 op clause1_2)/ (dx or dy or dz)
 - (clause2_1 op clause2_2)/(dx or dy or dz) )
```

Figure 3.9: The general structure of an FDTD method calculation

of dividing *dt* by *pmt* (permittivity) and *pma* (permeability). The divisions can therefore be pre-calculated and all of the values preloaded into SSE registers. This is done during program initialisation using the code given in Figure 3.10.

```
__asm__ ("movapd %0,%%xmm3 \n\t"
         "movapd %1,%%xmm4 \n\t"
         "movapd %2,%%xmm5 \n\t"
         "movapd %3,%%xmm6 \n\t"
         "movapd %4,%%xmm7 \n\t"
         :/*no output*/
         :"m" (ddx),
          "m" (ddy),
          "m" (ddz),
          "m" (ddtbpmt),
          "m" (ddtbpma)
         /*no clobbers*/);
```

Figure 3.10: Assembly sequence for storing constants in the SSE registers

The code in Figure 3.10 loads the 5 values into the highest 5 SSE registers, leaving the lowest 3 free for completing the arithmetic operations at each step in the algorithm. Figure 3.11 gives the code for the the calculation of a value of E_x in assembly language.

```
__asm__ ("movapd %1,%%xmm0 \n\t"  //load (hz(i,j+1,k)
         "movapd %2,%%xmm1 \n\t"  //load hz(i,j,k)
         "subpd  %%xmm1,%%xmm0 \n\t" //(hz(i,j+1,k)-hz(i,j,k))
         "movapd %3,%%xmm1 \n\t" //load (hy(i,j,k+1)
         "movapd %4,%%xmm2 \n\t" //load hy(i,j,k)
         "subpd  %%xmm2,%%xmm1 \n\t" //(hy(i,j,k+1)-hy(i,j,k))
         "divpd  %%xmm4,%%xmm0 \n\t" //divide by dy
         "divpd  %%xmm5,%%xmm1 \n\t" //divide by dz
         "subpd  %%xmm1,%%xmm0 \n\t" //subtract the two clauses
         "mulpd  %%xmm6,%%xmm0 \n\t" //multiply by dt/pmt
         "movapd %5,%%xmm1 \n\t"   //load ex(i,j,k)
         "addpd  %%xmm1,%%xmm0 \n\t" //add calculation to ex(i,j,k)
         "movapd %%xmm0,%0 \n\t" //store ex(i,j,k)
         :"=m" (*result)
         :"m" (*clause1_1),
          "m" (*clause1_2),
          "m" (*clause2_1),
          "m" (*clause2_2),
          "m" (*result)
            /*no clobber*/);
```

Figure 3.11: Calculation of E_x in assembly language using SSE

The same format of embedded assembly instructions is used to calculate each value for each matrix. The pointers for the variables passed into the assembly code are maintained in standard high-level C code.

The standard implementation described in Section 3.1 is based on an existing implementation in Fortran, and it is desirable to avoid a wholesale rewrite of the program. It is necessary to use C code to produce the assembly sequences, but procedures written in C can be called from a Fortran program by following a particular naming convention and using the GCC compiler to link the compilation units together. This allows the Fortran implementation to call procedures written in C only when required. Existing sequences in the program, such as the one which pre-calculates the values of *Jz* to model the signal can remain unchanged.

If a procedure in a C source file is named *calce_* then it can be referenced in a corresponding Fortran program as *calce*. The use of the under-score in the C code makes the procedure visible to the Fortran program. The C code for the assembly code version of the FDTD method contains three such functions: *calce_* for performing a calculation for all of E_x, E_y and E_z; *calch_* for performing a calculation for all of H_x, H_y and H_z; and *setconstants_* for pre-populating the values of *dx*, *dy* and so on into SSE registers as described above. Separately calling C code for calculating the $\mathbf{E}$ matrices and the $\mathbf{H}$ matrices allows the handling of source values from *Jz* to remain in the Fortran program.

Fortran, unlike C, does not include a method for dictating the alignment of data structures. Therefore in order to keep the matrix initialisation code expressed in Fortran, it is not possible to dictate 16-byte alignment. If the alignment of any of E_x, E_y E_z, H_x, H_y and H_z is incorrect, the algorithm will fail due to issuing instructions which expect alignment on unaligned addresses. Therefore in order to achieve alignment while keeping the bulk of the implementation in Fortran, it is necessary to dynamically test the starting address of each matrix and change the starting point of the algorithm within each matrix accordingly. An example of the code for this is given in Figure 3.12.

The alignment of the first element in the matrix is simply tested by taking the value of its address modulo 16. If aligned, the values of imin and imax are used as normal. If not aligned, the starting point is shifted by one in the *i* dimension. This means the FDTD algorithm begins at *(3,1,1)* rather than *(2,1,1)*, and alignment is maintained throughout the algorithm. The entire code is parameterised on *exitop*, *exibottom* and equivalent variables for the data structures *ey*, *ez*, *hx*, *hy* and *hz*. Therefore when shifting is performed on a matrix, all references to that matrix are also shifted. The effect

```
p=LOC(ex(0,1,1))
x= modulo(p,16)
if(x == 0) then
    exibottom=imin
    exitop=imax
else
    exibottom=imin+1
    exitop=imax+1
endif
```

Figure 3.12: Testing the alignment of a matrix in Fortran

of this shifting is shown in Figure 3.13.

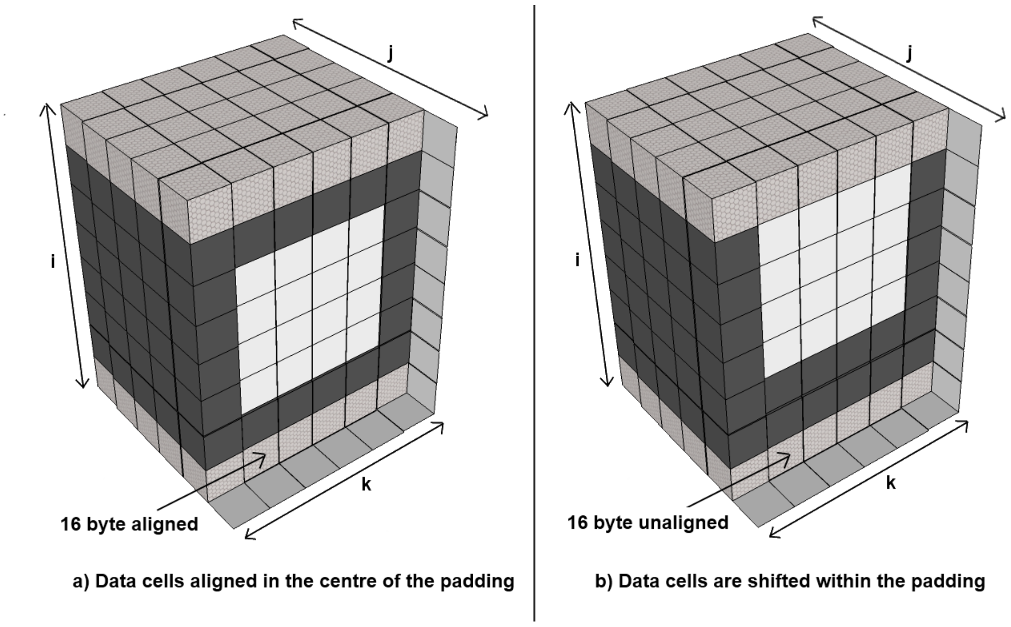

j
i
k
16 byte aligned
a) Data cells aligned in the centre of the padding
j
i
k
16 byte unaligned
b) Data cells are shifted within the padding

Figure 3.13: Shifting the problem space to ensure alignment to a 16-byte memory boundary. The nearest face in the *j* dimension has been removed to show the internal cells.

As shown in Figure 3.13, the problem space always has at least one layer of padding in each direction regardless of whether it is shifted or not. This is sufficient to ensure the algorithm operates correctly by maintaining a PEC boundary of at least one layer on each face.

The use of extra padding to facilitate an SSE based implementation adds additional memory usage to the algorithm. The additional memory used is a small fraction of the total memory usage of the algorithm, and the results in Section 4.1 demonstrate that even with this extra memory usage the SSE implementation out-performs the standard implementation in most cases. However as the problem size grows, eventually a point would be reached where the standard implementation would fit into memory but the SSE implementation would not. This is a consequence of requiring a layer of padding for the algorithm and requiring alignment of the first element within the problem size to be aligned in memory. To avoid the need for the extra padding, it would be necessary to have a facility to specify that the matrix is assigned to an unaligned address in memory, so that for example while $E_x(1,1,1)$ at the start of the matrix was unaligned, the first calculated address at $E_x(2,2,2)$ would be aligned. A facility to specify unaligned assignment in memory is not available in C or Fortran using the GCC compiler.

3.3 Implementation with SSE instructions using intrinsics

Programming using sequences of assembly language instructions is error-prone and it is desirable to avoid it where possible. A set of functions known as intrinsics are provided to simplify the task of programming using SSE instructions. Each intrinsic function represents a single assembly instruction but allows normal C variables to be specified as inputs and outputs and gives control of register assignment back to the compiler. The intrinsics library also provides the data-type *_m128d* which represents a pair of 64-bit double-precision values stored in a 128-bit register. This simplifies loading and storing from the matrices in memory to the registers. A pointer of type *_m128d* pointing to the address of $E_x(2,2,2)$ covers the data at this location and $E_x(3,2,2)$. A single increment of this pointer changes its target to $E_x(4,2,2)$, so the double-stepping described in Section 3.2 is achieved with a simple pointer increment operation. The use of the *_m128d* data-type and intrinsic functions greatly simplifies the implementation. The FDTD algorithm using intrinsics is structured as shown in the code in Figure 3.14.

```
__m128d temp1= _mm_sub_pd(*dhy_shift_k, *dhy);//(hy(i,j,k+1)-hy(i,j,k))
 temp1= _mm_div_pd(temp1, ddz);//divide by dz
__m128d temp2= _mm_sub_pd(*dhz_shift_j, *dhz);//(hz(i,j+1,k)-hz(i,j,k))
 temp2= _mm_div_pd(temp2, ddy);//divide by dy
temp2= _mm_sub_pd(temp2,temp1);//subtract the two clauses
temp2= _mm_mul_pd(temp2,ddtbpmt); //multiply by dt/pmt
*dex = _mm_add_pd(*dex,temp2);//add calculation to ex(i,j,k) and store
```

Figure 3.14: Calculating E_x using SSE intrinsic functions

Each intrinsic function operates on 128-bit values and therefore calculates the results for two elements of E_x simultaneously. This code completely replaces the assembly code instructions described in Section 3.2. The same issues of alignment and padding described in Section 3.2 apply to the intrinsic function implementation. The intrinsic functions do not dictate the use of specific registers, freeing the compiler to attempt optimisation in terms of register usage as is done for standard high-level code implementations.

Documentation on intrinsic functions and particularly material describing their correct use is not widely available. The best source of detailed specification is on the Microsoft Developer Network [29]. This source describes the intrinsic functions available in Microsoft's Visual Studio development environment. The same functions are available using the GCC compiler under Linux when using the header *emmintrin.h* in the C source files.

The results in Chapter 4 show that the use of intrinsics rather than direct assembly implementation has no significant impact on performance. In addition, while the assembly implementation failed to execute when moved from a 32-bit to a 64-bit platform, the intrinsics implementation executed correctly on all platforms. Therefore the intrinsics approach leads to simpler and more portable code without impacting performance, and it is recommended based on the experience of this project that intrinsics are used rather than assembly programming when implementing algorithms with SSE instructions.

3.4 CUDA GPU implementation

As described in Section 2.3.1, the CUDA programming language allows programs to be written to execute on the streaming processor array of an NVIDIA Graphics Processing Unit (GPU). The aim is to find sufficient parallelism in the algorithms of the program to maximise the use of the many processor cores on the GPU.

The available graphics hardware initially available for this project was a pair of Tesla T10 processors on a machine at the University of Kyushu in Japan and therefore accessed remotely. While this arrangement was suitable for batch execution of completed implementations, it was not practical during development where it was desirable to frequently execute intermediate builds to test progress. Subsequently additional GPU hardware was rented from the Amazon cloud computing service, where it was desirable to keep the amount of time spent executing on the hardware to a minimum to reduce cost. This hardware was therefore also no suitable for development work. In order allow offline development, a CUDA GPU environment was emulated on a laptop running Linux. The Ocelot project provides a framework allowing CUDA programs to be compiled for a variety of targets, including GPUs from manufacturers other than NVIDIA and x86 architecture CPUs [30]. The CUDA compiler produces intermediate output in Parallel Thread Execution (PTX) assembly language which is then usually further compiled to execute on the GPU [30]. Ocelot provides an alternative set of tools for compiling CUDA code which includes an emulator for executing PTX instructions on an x86 processor [30]. By using the capabilities of Ocelot it was possible to create a CUDA development environment on a laptop without CUDA capable GPU hardware. From the experience of this project, it was found that execution of CUDA programs on an x86 CPU using Ocelot is much slower than native execution on GPU hardware, but is sufficient to allow algorithms to be developed and small problem sizes to be tested before migrating the code to a GPU environment. Using this approach greatly improved productivity during CUDA development and reduced the amount of time required on the University of Kyushu machine, which is a shared resource. It also reduced the amount of execution time required on the Amazon cloud computing service which is a costly resource.

In Section 3.2, the stated aim was to maintain as much of the original Fortran implementation as possible with only the essential parts being implemented in C. In the case of the CUDA implementation, the NVIDIA CUDA compiler is designed only for C and C++ code. Fortran to CUDA compilers are available, but they are commercially licenced and less widely available so have not been considered here. Due to the significant differences between GPU and CPU programming, the GPU implementations presented here do not attempt to maintain any of the standard Fortran implementation, and are reimplemented from scratch in C code, following the pattern of the Fortran implementation as a guide.

Before developing a parallel version of the FDTD method for execution on a GPU, a sequential version was created to show that the GPU could be accessed and that its processors execute the calculations correctly. In the sequential implementation, each element of each matrix is calculated one after the other, on a single core of the GPU's streaming processor array. Before the GPU can manipulate data, it is necessary to move the data from main memory to the global memory of the GPU. Figure 3.15 shows the sequence of steps used to achieve this.

Lines 1 and 2 allocate space in main memory for the matrix E_x. The variable *VOL* gives the number of elements required in the matrix including padding. Unlike the Fortran based implementations in Sections 3.1 and 3.2, dynamic allocation of a block of memory is being used to assign enough memory for each matrix rather than assignment

```
1  fp_type *ex;
2  ex=(fp_type *)malloc( VOL * sizeof(fp_type) );
3  fp_type *dev_ex;
4  cudaMalloc((void **)&dev_ex,VOL*sizeof(fp_type));
5  cudaMemcpy(dev_ex,ex,VOL * sizeof(fp_type), cudaMemcpyHostToDevice );
6
7  //..execute FDTD algorithm
8
9  cudaMemcpy(ex,dev_ex,VOL * sizeof(fp_type), cudaMemcpyDeviceToHost );
```

Figure 3.15: Moving data from main memory to GPU memory

of memory based on multi-dimensional array indices. This is because the C programming language specification limits the size of array that can be statically allocated. It would be undesirable to produce an implementation whose scalability was limited by this factor. Additionally, the data structures need to be moved to the GPU using pointer-based memory copies regardless of the method of assignment, so assigning the memory using array notation would not significantly simplify the implementation.

Lines 3 and 4 show the allocation of space in the GPU's global memory, matching the size of the space allocated in main memory. Line 5 transfers the contents of main memory to the GPU, and line 9 transfers the contents of memory from the GPU back to main memory after the execution of the FDTD algorithm. This sequence of instructions is performed for each of E_x, E_y, E_z, H_x, H_y and H_z. Provided that the FDTD algorithm is executed correctly on the GPU between the two memory copy instructions, this will leave a correct FDTD solution in main memory at the end of execution. Note that the data type *fp_type* is defined in a macro as being equivalent to the standard type *double*. Changing this macro from *double* to *float* changes the entire program from double-precision to single-precision without any further changes to the program being required.

As described in Section 2.3.1, code to be executed on the GPU is termed a kernel. There is no guarantee of execution order for the threads launched within kernel, and while synchronisation between threads within a block is possible, synchronisation between blocks is not. For this reason, the maintenance of the source value at a single point of E_z to represent an electromagnetic signal should not be performed within a kernel. Maintaining the value of the source of excitation requires all threads to stop at the end of the processing of the elements of E_x, E_y and E_z during a particular timestep, and to resume again processing H_x, H_y and H_z after the source value is updated by a single thread. This would require application-wide thread synchronisation which CUDA does not support within a kernel. In order to work around this, the execution of

E_x, E_y and E_z for a single iteration in time is implemented as one kernel, and the execution of H_x, H_y and H_z as another. This means that there are two kernel executions for each time-step. This approach does not prevent the possibility of high performance execution as launching a kernel is a lightweight operation. The large memory copy operations are the bottle-neck of GPU performance, and these only need to be performed once at the beginning and end of the algorithm regardless of the number of kernel launches performed. The code which performs time-step iterations, launches kernels, and updates the source value is given in Figure 3.16.

```
for(t=1;t<=ITERATIONS;t++)
{
    //calculate all values of E
    eKernel<<<1,1>>>(dev_ex, dev_ey, dev_ez, dev_hx, dev_hy,
                     dev_hz, dx, dy, dz, pmt, dt);
    //Update source value in main memory
    *(ez+offset(DIM/2-1,DIM/2-1,DIM/2-1))=-jz[t];
    //Move updated source value to GPU
    cudaMemcpy(&dev_ez[offset(DIM/2-1,DIM/2-1,DIM/2 -1)],
               &ez[offset(DIM/2-1,DIM/2-1,DIM/2 -1)],
    sizeof(fp_type),cudaMemcpyHostToDevice);
    //calculate all values of H
    hKernel<<<1,1>>>(dev_ex, dev_ey, dev_ez, dev_hx,
                     dev_hy, dev_hz, dx, dy, dz, pma, dt);
}
```

Figure 3.16: Execution of time-steps in the GPU implementation

Note that there is a memory copy between each kernel invocation for each iteration in time, to move the updated source value to the GPU. This is moving a single element (4 bytes for single-precision or 8 bytes for double-precision) so does not expose the algorithm to the cost of the bottleneck of memory copy performance in the way that repeatedly copying the whole matrices would. If a soft-source rather than hard-source (see Section 3.1) was used to generate activity within the problem space, the current value of E_z at the point of the source would be required for the calculation. In that case, there would be an additional memory copy to bring the current value of E_z from the GPU memory to main memory before the calculation of the source value could be performed. The memory copy overhead of updating the source of excitation for a soft source would therefore be twice that of a hard source.

The addresses of the data-structures in GPU memory (*dev_ex*, *dev_ey*, *dev_ez*, *dev_hx*, *dev_hy* and *dev_hz*) are passed into the kernels as pointers, while *dx*, *dy*, *dz*, *dt*, *pma* (permeability) and *pmt* (permittivity) are passed by value as they are not altered by the kernels. All of these arguments to the kernel functions are stored as private register variables for each launched thread.

Since this is a sequential implementation, each kernel is launched as a single block containing a single thread, as denoted by $<<<$ *1,1* $>>>$ in the kernel launch statement. When a kernel is launched with parameters $<<<$ *a,b* $>>>$, *a* dictates the number of blocks within the grid for that kernel and *b* dictates the number of threads within each block. In this case, since a single thread will be responsible for every calculation for each matrix, the kernels can simply be implemented using a triply nested loop, similar to the standard implementation in Section 3.1. The code for the sequential *eKernel* function is shown in Figure 3.17 (the *hKernel* function is similarly implemented). Together, the code in Figure 3.16 and the kernels implemented as per the code in Figure 3.17 are analogous to the Fortran code in Figure 3.2. Both represent the code for performing all the time-steps of the FDTD method in a sequential manner.

```
for(k=1;k<=DIM;k++)
{
   for(j=1;j<=DIM;j++)
   {
     for(i=1;i<=DIM;i++)
     {
        int currentOffset=offset(i,j,k);
        dex[currentOffset] = dex[currentOffset] + dt/pmt * (
             ((dhz[offset(i,j+1,k)] - dhz[currentOffset])/dy) -
             ((dhy[offset(i,j,k+1)] - dhy[currentOffset])/dz)
           ) ;

        dey[currentOffset] = dey[currentOffset] + dt/pmt * (
           ((dhx[offset(i,j,k+1)] - dhx[currentOffset])/dz) -
           ((dhz[currentOffset+1] - dhz[currentOffset] )/dx)
          ) ;

         dez[currentOffset] = dez[currentOffset] + dt/pmt *  (
             ((dhy[currentOffset+1] - dhy[currentOffset])/dx) -
             ((dhx[offset(i,j+1,k)] - dhx[currentOffset])/dy)
           ) ;
     }
   }
}
```

Figure 3.17: Sequential kernel to update the values of E_x, E_y and E_z

In both C and Fortran, it is normal to reference three dimensional arrays using the variables *i*, *j* and *k* in that order. In C, the three dimensional structure is stored in a manner known as “row-major” meaning that for an array a, the location in memory of *a[i,j,k]* is followed immediately by the location of *a[i,j,k+1]*. Conversely, Fortran stores three dimensional arrays as “column-major” so the array element following *a[i,j,k]* in memory would be *a[i+1,j,k]*. The result is that to achieve single-stepping through the data in Fortran using a triply nested loop, the *i* loop should be the inner loop while the *k* loop is the outer loop. In the C the reverse is true, single-stepping requires *i* loop should be the outer loop while the *k* loop is the inner loop. In the case of this CUDA C implementation, the data structures have not been declared as three dimensional arrays. Figure 3.15 shows that the data-structure is declared as an allocated block of memory. This is interpreted as a three dimensional structure within the kernel implementation by using the *offset* function shown in Figure 3.18.

```
__host__ __device__ int offset(int i,int j,int k)
{
      return i+ ( j * (DIM+2) ) + ( k * (DIM+2) *(DIM+2) );
}
```

Figure 3.18: Definition of the *offset* function

The function *offset* converts a combination of 3 indices to a one-dimensional offset within the memory allocated for a particular matrix. It abstracts above the tasks of multiplying the j index by the dimension of the matrix and the k index by the dimension squared, and takes the padding for the PEC boundary into account. Because this is a custom-defined function, there is flexibility in how *i*, *j* and *k* are interpreted. In this case, the indices are being used in keeping with the Fortran convention. It is a single increment of the *i* index rather than the *k* index which leads to a single step in memory. By setting *k* to the outer loop and *i* to the inner loop in Figure 3.17 single stepping through memory is achieved. This is equivalent to referencing a statically assigned three-dimensional array in C using *[k,j,i]* rather than the more conventional *[i,j,k]*. This is done to maintain consistency with the Fortran implementations but in the case of a Tesla GPU there is no cache between the cores of the streaming processor array and its global memory, and therefore there is no particular benefit to the arrangement of *i*, *j* and *k* for this sequential implementation.

Declaring *offset* as both a host and device function allows it to be used throughout the program regardless of where the calling code will be executed. The compiler will compile this function twice, once for the CPU and once for the GPU, and take care of the correct version being called as required. The repeated use of the *DIM* variable means that this implementation is limited to FDTD applications where the problem space is cube-shaped. Where a cuboid shape for the problem space is required, the implementation would need to be altered to use a different variable to store the size of each dimension.

This sequential approach produces a correct FDTD implementation, but it is much slower than the most basic CPU implementation, since it takes advantage of only a single core of the GPU. Each core of a CUDA capable GPU is less powerful than a standard CPU core. For example, each core of the Tesla T10 GPU used in this project has a clock speed of 1.3GHz (see Table 4.6), while the CPUs used for the SSE experiments range from 1.9GHz to 3.0GHz (see Table 4.1). More detrimental to sequential performance than the lack of clock speed is that there is no cache between the GPU core and the global memory it accesses, so the execution is fully exposed to the difference between memory throughput and the clock speed of the core [8]. As shown in Table 4.1, all the CPUs used in this project provide some level of cache between the CPU and main memory. The implications of the lack of cache in a Tesla GPU are explored further in Section 4.3.

In order to achieve high performance on a GPU, the concurrency in the algorithm must be exploited. Each element of the three $\mathbf{E}$ matrices can be calculated independently of the others, and similarly, each element of the three $\mathbf{H}$ matrices are independent. Due to the way the kernels have been allocated, this means that all the calculations within one kernel are independent and can be executed in any order and distributed among many cores.

As described in Section 2.3.1, the data to be processed on a GPU is termed a grid, and each grid can be divided into blocks in up to two dimensions. Each block can be further divided into threads in up to 3 dimensions. Therefore there is a two level hierarchy allowing up to 5 dimensions which determines how many threads are created to execute a single kernel invocation. Figure 3.19 from [8] illustrates how the data used by a kernel are represented by a single grid and how the grid is deconstructed into blocks and threads.

In the case of the FDTD method, the data structures are naturally three dimensional. A grid of blocks in two dimensions is used here, with threads within each block defined in one dimension. In the first instance, an implementation was produced using multiple blocks but only a single thread per block. In order to instruct the CUDA environment to create multiple blocks for each kernel invocation, the code invoking the kernels had to be altered. The relevant alterations are shown in Figure 3.20.

The *dim3* data type is a built-in type of the CUDA environment which allows multi-dimensional quantities to be defined. In this case, the number of blocks is being defined as equal to the problem size in two dimensions while the number of threads per block remains at 1.

The two dimensions of the blocks were mapped to the j and k dimensions of each data structure representing the problem space. The number of blocks in a kernel invocation is equivalent to the square of the dimension of the matrices. The single thread

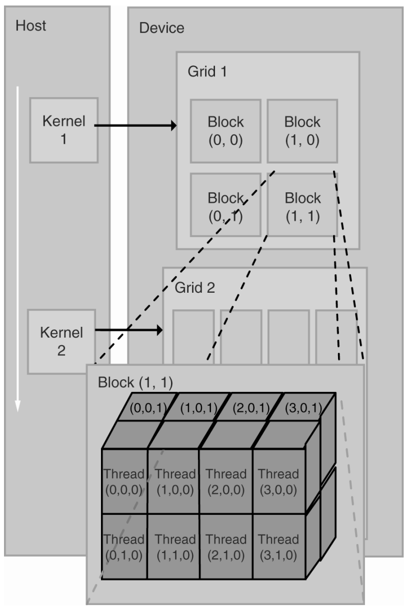


Figure 3.19: CUDA grid organisation [8]

```
dim3 blocks(DIM,DIM);
//...
eKernel<<<blocks,1>>>(dev_ex, dev_ey, dev_ez, dev_hx
                      ,dev_hy, dev_hz, dx, dy, dz, pmt, dt);
//...
 hKernel<<<blocks,1>>>(dev_ex, dev_ey, dev_ez, dev_hx,
                        dev_hy, dev_hz, dx, dy, dz, pma, dt);
```

Figure 3.20: Introducing multiple blocks to the kernel invocations

within each block is therefore responsible for calculating all the elements in the *i* dimension for a particular *(j,k)* combination, for each matrix. This allocation of work is shown in Figure 3.21.

Figure 3.21 shows the internals of a matrix for a 4 $\times$ 4 $\times$ 4 problem size (shown as light grey cubes) with one element of padding (shown as dark grey cubes) in each direction. Each block is indexed based on its position in the *j* and *k* directions and the single thread within each block calculates the values for the 4 elements of a single 'tower' in the *i* direction.

With this distribution of work determined, the CUDA framework is then responsible for launching a single thread running the kernel for each block. Each thread has access to the indices of its block within the grid in two dimensions, denoted in CUDA as *blockIdx.x* and *blockIdx.y*. These are built in variables of the CUDA environment and cannot be altered in name. This implementation logically maps *blockIdx.x* to *j* and *blockIdx.y* to *k*. The *k* and *j* loops are no longer required since they have been replaced

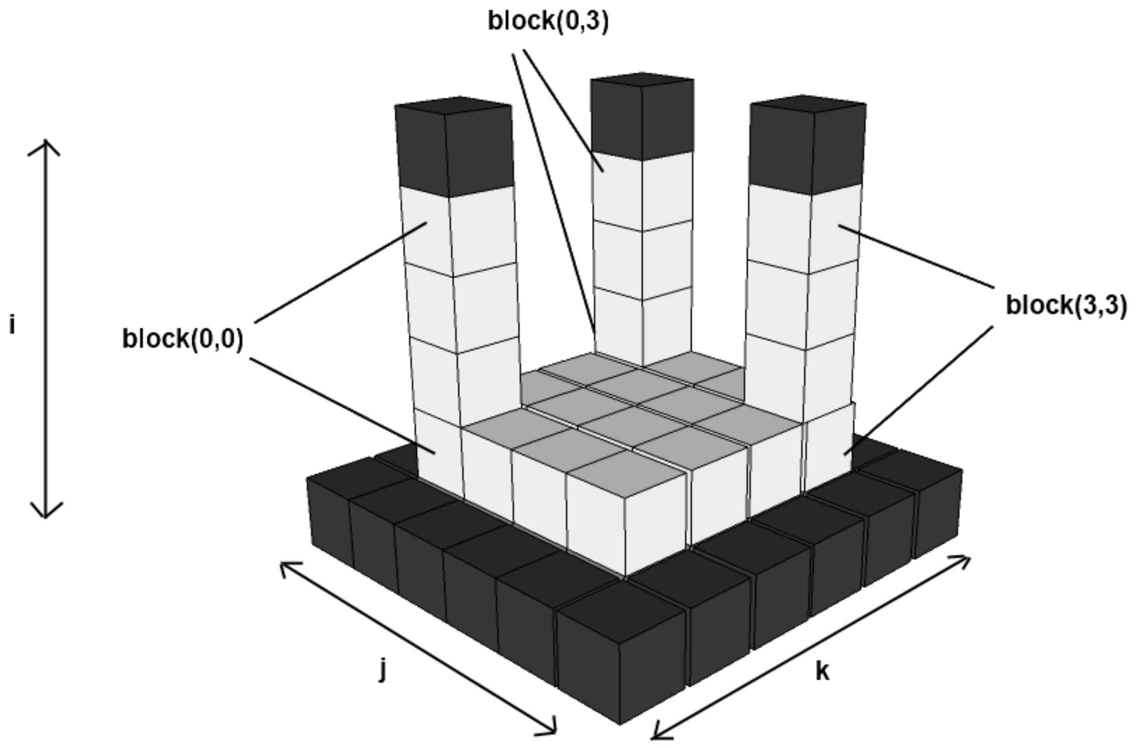


Figure 3.21: Representation of the division of a matrix into a two dimensional grid of blocks.

by the launching of multiple blocks, and the triply nested loop of the sequential version shown in Figure 3.17 is replaced with the code from Figure 3.22.

```
j=blockIdx.x+1;
k=blockIdx.y+1;
int dim=gridDim.x*blockDim.x
for(i=1;i<=(dim);i++)
{
        //Calculate Ex(i,j,k), Ey(i,j,k) and Ez(i,j,k)
}
```

Figure 3.22: Amendments to the kernel to handle multiple blocks

The items *gridDim.x* and *blockDim.x* are also built-in CUDA variables giving the size of the grid and of each block in the x dimension. Multiplying them together is a generic solution for calculating the total problem size in the x dimension. Equivalent variables in the y dimension for the grid and its blocks, and in the z dimension for blocks only (since blocks can be divided into threads in 3 dimensions) also exist. Note that since this implementation is limited to dealing with cubic problem spaces, multiplying *gridDim.x* by *blockDim.x* is sufficient to get the size of the problem space in every direction.

Each thread is now doing less work than the single thread in the sequential version due to have a fixed value for *j* and *k*, but multiple threads are launched, and these threads execute in parallel. Note that the *blockIdx.x* and *blockIdx.y* indices of a block within a CUDA grid always begin from 0, whereas due to the padding on the data structures of the FDTD method, *j* and *k* must start from 1. This is why both are incremented by 1 when assigned from the block indices. The dimension of the grid defined by multiplying *gridDim.x* and *blockDim.x* does not include the padding since no threads are required to operate on the PEC boundary which remains at 0 throughout the execution. The calculations within the *i* loop are unchanged from those of the sequential version in Figure 3.17.

As shown in Section 4.3, the performance of this one thread per block implementation is comparable to the slowest CPU based implementations. The cores of a CUDA capable GPU are combined in groups of 8 into a Streaming Multiprocessor unit, as described in Section 2.3.1. So for example the Tesla T10 GPU used in Section 4.3 has 240 cores, grouped into 30 Streaming Multiprocessor units. Each Streaming Multiprocessor can execute up to eight blocks simultaneously [21]. However, at any one clock cycle, the threads issued to the eight cores of the Streaming Multiprocessor must come from the same block [21]. In the case of the one thread per block implementation presented in Figures 3.20 and 3.22 there is only one thread to be executed for each block, so each Streaming Multiprocessor will only utilise one core at each clock cycle. Across a Tesla T10 GPU, this limits the execution to a maximum of 30 concurrent operations rather than 240 concurrent operations if all cores were fully utilised. In order to fully exploit the parallel processing power of a CUDA capable GPU, it is necessary to further increase the parallelism of the kernels by increasing the number of threads executing within a single block. As described in Section 2.3.1, a single block can accommodate up to 512 threads. In order to be flexible, an implementation is required where the number of threads is not tied to the dimensions of the problem. This means that each thread will have to execute a sub-set of the elements in the *i* dimension for a particular block. There are two different methods for achieving this, as shown in Figure 3.23.

In Figure 3.23 part a), each column of elements in the *i* direction, representing the workload of a single block, is divided into two segments. Each of two threads (denoted T0 and T1) is assigned a segment of the column to work on. This means an individual

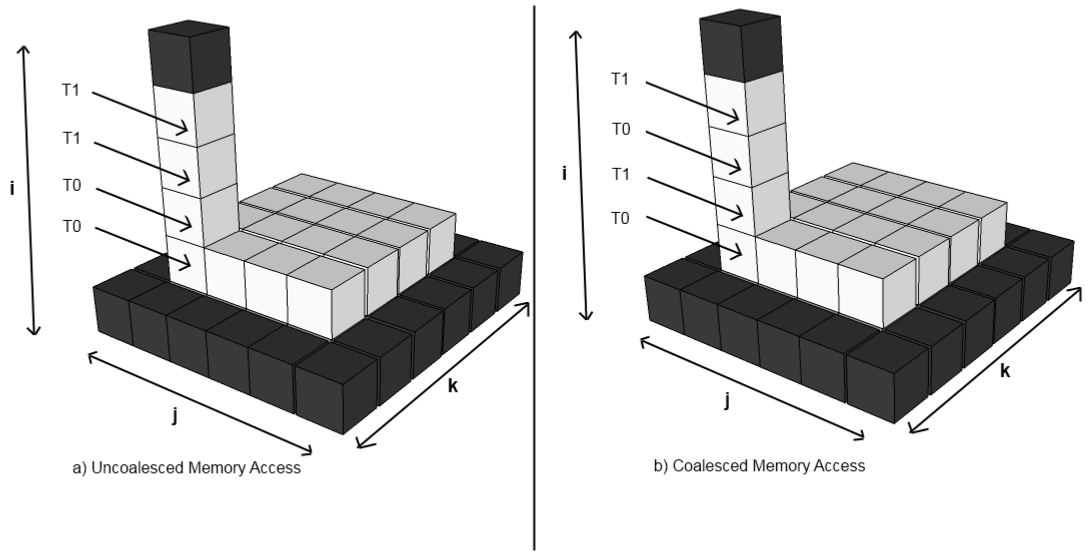


Figure 3.23: Patterns of memory access for multiple threads in a block.

thread single steps through through the elements in its segment. This would be the standard approach for dividing an array between threads on a multi-core CPU. Each core on such a CPU typically has one or more levels of private cache used only by that core, so single stepping allows a core to load a sequence of elements into a single cache line and efficiently work through it. However, in the case of Tesla GPUs, there is no cache. Due to the Single Instruction Multiple Thread (SIMT) approach described in Section 2.3.1, multiple CUDA threads in a group known as a warp move through the instructions of a kernel synchronously, meaning that the requests for elements from memory occur simultaneously [21]. In this case, T0 and T1 would be in the same warp and would simultaneously access their first element, which would lead to simultaneous requests for elements which are spaced far apart in memory. This is uncoalesced memory access as described in [8] and leads to poor performance. The structure of an uncoalesced kernel is shown in Figure 3.24.

```
j=blockIdx.x+1;
k=blockIdx.y+1;
int iterations=gridDim.x/THREAD_COUNT;
i=threadIdx.x*iterations+1;
while(i<=threadIdx.x*iterations+iterations)
{
  //Calculate Ex(i,j,k), Ey(i,j,k) and Ez(i,j,k)
  i++;
}
```

Figure 3.24: Structure of the uncoalesced kernel implementation

Now that the implementation is using multiple threads per block, the built-in variable *threadIdx.x* gives the index of this thread within the current block. If multiple dimensions of threads were being used, *threadIdx.y* and *threadIdx.z* would also be available.

The number of iterations is calculated dynamically at line 3 by dividing the size of the matrices by the number of threads. The starting point for this thread is then calculated at line 4 by multiplying the thread ID by the number of iterations, effectively moving through several segments to the correct one for this thread. The *for* loop is replace with a while loop checking for the index *i* to exceed the end of the segment assigned to this thread. The calculations for the individual elements of the matrices are unchanged from the sequential GPU version.

Figure 3.23 part b) shows an alternative where the elements are assigned to the threads in a round-robin fashion. This leads to threads simultaneously requesting adjacent elements from memory throughout the execution of the algorithm, which is coalesced memory access [8]. The Streaming Multiprocessor is able to combine adjacent memory requests into a single request which leads to better global memory throughput and improved performance [8]. The structure of a coalesced kernel is shown in Figure 3.25.

```
j=blockIdx.x+1;
k=blockIdx.y+1;
i=threadIdx.x+1;
while(i<=gridDim.x)
  //Calculate Ex(i,j,k), Ey(i,j,k) and Ez(i,j,k)
  i+=THREAD_COUNT;
}
```

Figure 3.25: Structure of the coalesced kernel implementation

In this case, the starting point for a thread is only based on its thread ID, since the starting elements for all threads are adjacent at the beginning of the column of elements in the *i* dimension. In this case, the termination condition for the while loop is when the index exceeds the dimension of the matrix, and within each loop, the index *i* is incremented by the number of threads. This causes each thread to jump multiple elements at each iteration (equal to the number of threads in the block) and achieves the interleaving of threads to elements shown in Figure 3.23 part b).

In both the uncoalesced and coalesced implementations, the kernel invocations need to be altered to specify the number of threads. The code for this is given in Figure 3.26.

```
dim3 blocks(DIM,DIM);
dim3 threads(THREAD_COUNT);
//...
eKernel<<<blocks, threads>>>(dev_ex, dev_ey, dev_ez, dev_hx
                            , dev_hy, dev_hz, dx, dy, dz, pmt, dt);
//...
 hKernel<<<blocks, threads>>>(dev_ex, dev_ey, dev_ez, dev_hx,
                               dev_hy, dev_hz, dx, dy, dz, pma, dt);
```

Figure 3.26: Launching kernels with multiple blocks and multiple threads

The *dim3* data type is used to specify the dimensions of both the blocks and the threads. The variable *THREAD_COUNT* is set by a macro within the source code.

The results in Section 4.3 show that when the number of threads per block is lower, there is a significant difference between the performance of uncoalesced and coalesced access, with coalesced executing with better performance. The reasons for and implications of this are discussed further in Section 4.3.

Due to the large amount of parallelism inherent in the FDTD algorithm and the flexibility in CUDA of changing the structure of kernels and the dimensions of grids and blocks, there are many other ways of dividing the algorithm for execution on a GPU. [8] states that there is not currently an accepted methodology for coming up with the optimum structure of CUDA based implementation of an algorithm without using experimental methods, and performing experiments for all possibilities at a variety of problem sizes can be tedious and time consuming. Here, an iterative process of refinement has been used to move from a sequential algorithm to one using parallelism expressed in blocks and threads with coalesced memory access. Decisions about the correct allocation of work to kernels and the mapping of blocks and threads to the dimensions of the matrices have been based on the advice in [8] and [20]. These are the two major texts on CUDA implementation available at the time of writing. The decision to limit the experiments to the implementations shown here and not proceed with further exploration was mainly due to managing the scope of the project, in terms of the total time available to spend on implementation. With more time, further possibilities such as storing the values of the variables *dx*, *dy*, *dz*, *dt*, *pmt* and *pma* in constant memory and utilising shared memory for cooperation between threads would have been explored. The theory behind the application of these techniques to this project is explored further in Section 4.3.

3.5 Implementation accuracy

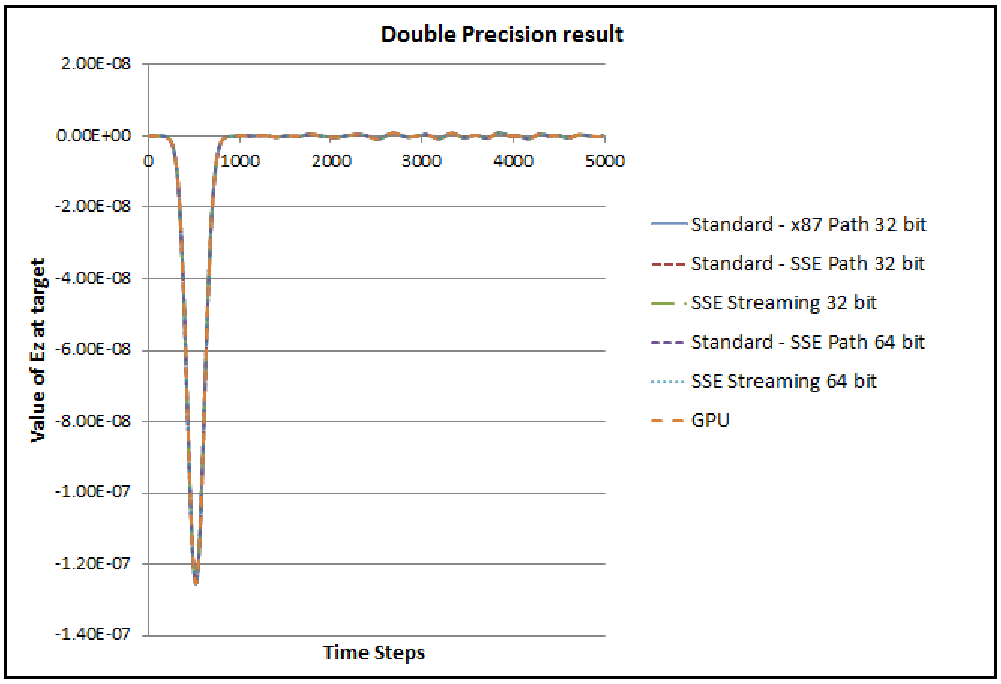
As detailed in Section 2.2.3, the same floating point arithmetic calculations can produce different results depending on a variety of environmental factors such as hardware, compiler, and level of precision. In this project, there are 3 different hardware types (32 bit CPUs, 64 bit CPUs, CUDA capable GPUs), 3 compilers (GNU Fortran, GNU C, and CUDA C), and two possible levels of precision (8 byte double precision or 4 byte single precision). It is important to know whether these differing options produce outputs resulting in significantly varying solutions to an application of the FDTD method. If the implementations produce significantly different output they cannot be considered equivalent in terms of accuracy and therefore their comparative performance does not hold as much relevance. In order to test this, the value of E_z at a point displaced 10 places in the i dimension from the source was measured and recorded at each time step during an execution of the FDTD method. The source is at the centre of the problem space using the pulse described in Section 3.1 and illustrated in Figure 3.3. A problem dimension of 256 was used, and the implementations were executed for 5000 time-steps. Three different CPU based implementations were measured:

- The standard Fortran implementation compiled to use the default x87 floating-point path.
- The standard Fortran implementation compiled to use the SSE floating-point path.
- The SSE streaming Fortran implementation.

The different floating point paths available on x86 architecture processors are described further in Section 2.2.3. The Intel Core 2 Duo 32-bit machine and the Intel Xeon 64-bit machine used in performance experiments were used for measuring accuracy. The exact specifications of these machines are detailed in Table 4.1. Note that on the 64-bit architecture, the SSE floating-point path is the default so in that case, the x87 FPU result was not measured.

The GPU result was measured using the Tesla T10 GPU described in Table 4.6. The GPU implementation with 64 threads per block and coalesced memory access was used, as described in Section 3.4. Note that all GPU implementations were observed to produce identical output, and therefore the results in this section represent the accuracy of the whole class of GPU implementations produced in this project.

The accuracy measurements were performed for both double and single precision. The results are shown in Figure 3.27.



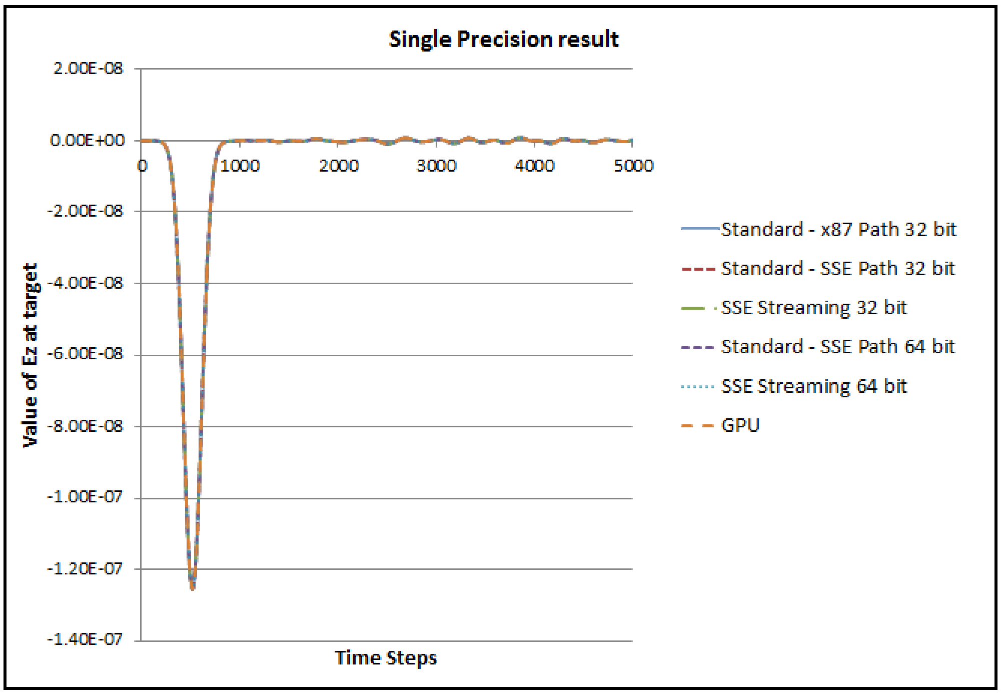


Figure 3.27: Results over time of measuring the effect of the source pulse at a problem size of 256 for several implementation options.

The results of each implementation are so similar that the individual lines cannot be distinguished on the graphs in Figure 3.27. There are in fact slight differences between some of the results. The CPU result for the x87 floating-point path differs from the SSE results. The standard implementation compiled for the SSE registers produces identical results to the streaming SSE implementation, indicating that the two implementations are exactly equivalent when executed on the same hardware. There is no difference between the SSE results on 32-bit and 64-bit hardware. Furthermore, the GPU implementation produces slightly different results from both CPU paths. There are therefore 3 different result-sets at each level of precision, corresponding to the 3 different floating-point hardware options in use.

In the double-precision results, the 3 result-sets are equal to at least 4 significant figures at each time step, for single-precision this is reduced to 3 significant figures. The different approaches taken to accelerate the FDTD computation in this project produce results which are very close to that of the standard sequential Fortran implementation, indicating that they are suitable alternatives in terms of accuracy. Chapter 4 explores the performance benefits of each approach.

As can be observed in Figure 3.27, the single and double precision implementations produce very similar results. In fact, the single-precision results match the double-precision results to 3 significant figures. For this application, a switch to single-precision arithmetic does not result in a major loss of accuracy. This challenges the assumption that all scientific algorithms require double-precision accuracy, and is similar to the results of the Magnetic Resonance Imaging (MRI) example given in [8]. [8] shows that for their MRI application, single precision execution gives acceptable accuracy and greatly improved performance over double precision. These results do not extend to concluding that single-precision is acceptable in the general case, since every application has different requirements. For example, here the most simple form of the FDTD method is being used which does not consider the electric flux density field component (denoted as *D*). This field component is of the order of 10^{-12} of the size of the values of the *E* component, therefore more complex applications of the FDTD method which include *D* require greater precision. However these results do suggest that the necessity for double-precision should be scrutinised on a case by case basis when implementing algorithms which rely heavily on floating-point arithmetic, due to the potential performance benefits of executing with single-precision.

Chapter 4

Results of Experimentation

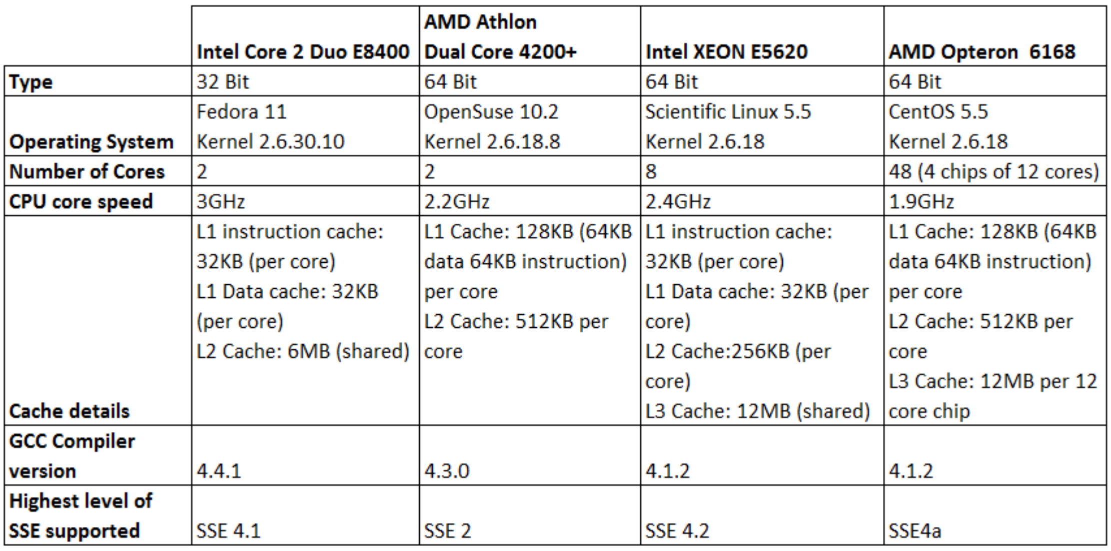
4.1 Results of SSE experiments

The SSE implementations described in Sections 3.2 and 3.3 were performance tested on four hardware architectures, described in Table 4.1. Note that each architecture has a varying number of cores available. The implementations produced in this project only exploit the SIMD hardware available on a single processor core, and do not exploit multiple cores using multi-thread or multi-process techniques. Since only a single core is being used when executing on each architecture, the number of cores is not expected to impact the performance results.

The GCC compiler was used on each architecture to compile both the Fortran and the C components of the implementation. As indicated in Table 4.1, the architectures do not all share the same version of the GCC compiler. In addition to the SSE implementations, the standard Fortran version described in Section 3.1 was executed to provide a baseline against which the performance of the SSE implementations could be compared.

Initially, four problem sizes of 64, 128, 192 and 256 in each dimension were used. The FDTD method was executed for 5000 time steps at each problem size. The execution was timed from the beginning of the 5000 time-steps to the end of the 5000 time-steps. The routines which initialise the matrices and pre-calculate the values used for the source of excitation were not included in the measured execution time. These areas of the code do not contain any optimisations so by excluding them, the results focus only on the level improvement experienced in the optimised section of the code.

Each timed result was produced ten times at each problem size on each architecture, with the average execution time of the ten being used. Figures 4.1 to 4.4 show the



	Intel Core 2 Duo E8400	AMD Athlon Dual Core 4200+	Intel XEON E5620	AMD Opteron 6168
Type	32 Bit	64 Bit	64 Bit	64 Bit
Operating System	Fedora 11 Kernel 2.6.30.10	OpenSuse 10.2 Kernel 2.6.18.8	Scientific Linux 5.5 Kernel 2.6.18	CentOS 5.5 Kernel 2.6.18
Number of Cores	2	2	8	48 (4 chips of 12 cores)
CPU core speed	3GHz	2.2GHz	2.4GHz	1.9GHz
Cache details	L1 instruction cache: 32KB (per core) L1 Data cache: 32KB (per core) L2 Cache: 6MB (shared)	L1 Cache: 128KB (64KB data 64KB instruction) per core L2 Cache: 512KB per core	L1 instruction cache: 32KB (per core) L1 Data cache: 32KB (per core) L2 Cache:256KB (per core) L3 Cache: 12MB (shared)	L1 Cache: 128KB (64KB data 64KB instruction) per core L2 Cache: 512KB per core L3 Cache: 12MB per 12 core chip
GCC Compiler version	4.4.1	4.3.0	4.1.2	4.1.2
Highest level of SSE supported	SSE 4.1	SSE 2	SSE 4.2	SSE4a

Table 4.1: Details of architectures used for SSE experiments

results of these executions. The execution time in seconds is plotted on the y-axis and the different implementations are grouped by architecture on the x-axis.

Note that at all problem sizes, there are only SSE assembly implementation results for the 32-bit machine. The hand-coded assembly was developed using the 32-bit machine and did not run correctly on the 64-bit machines. The cause of the failure to run on the 64-bit machines was unclear. It could be that the 64-bit x86 architecture requires different SSE instructions, although there is no indication of this in Intel's technical documentation. Alternatively, it could be that the older compiler versions on the 64-bit machines are not able to correctly handle the inline assembly instructions in C code or that the 64-bit compilers make use of the SSE registers for their own purposes and this clashes with the hand-coded use of these registers in the assembly sequences. Figures 4.1 to 4.4 show that the implementation using SSE streaming with assembly and SSE streaming with intrinsic functions perform almost identically on the 32-bit architecture. As described in Chapter 3, it was found during implementation that programming with intrinsic functions was simpler than programming with assembly instructions. These results show in addition that the intrinsic function implementation is more portable, without significant detriment to performance. The conclusion is that using intrinsic functions is the better approach for making use of streaming SSE capabilities, and for this reason the hand coded assembly implementation was not pursued further to make it work on the 64-bit architectures.

The Standard Fortran implementation listed in Figures 4.1 to 4.4 was compiled

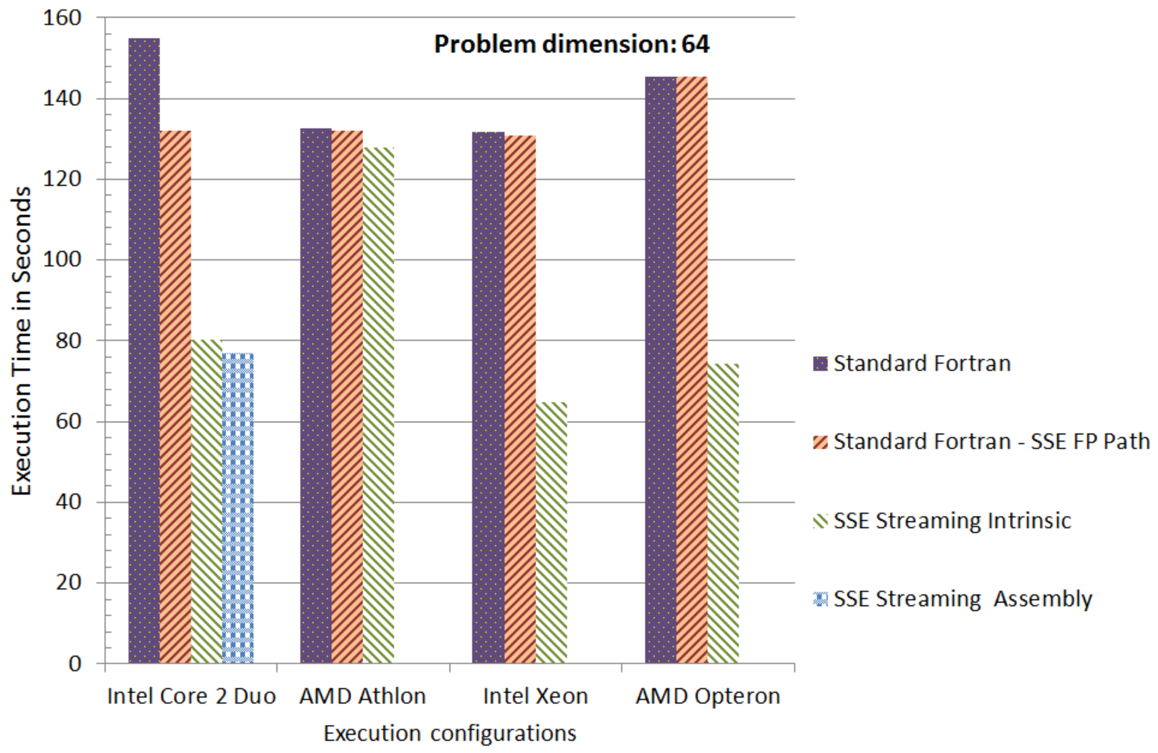


Figure 4.1: Performance results for the 64 × 64 × 64 problem size.

without forcing the use of a particular floating point unit, allowing the default floating point unit to be used. The command-line compilation scripts are given in Figure 4.5.

This means that on the 32-bit architecture this compilation results in execution on the x87 Floating point unit, whereas on the 64-bit architectures it results in execution using the SSE registers and the related SSE instructions (these are the scalar SSE instructions for performing one floating-point calculation at a time, not the packed instructions for performing SIMD operations).

The "Standard Fortran - SSE FP Path" implementation included in Figures 4.1 to 4.4 is the same code as the Standard Fortran implementation but compiled to force the use of the scalar SSE instructions to perform floating point arithmetic. The command-line compilation script used is also given in Figure 4.5.

Since the SSE path is the default path on the 64-bit architectures, the two compilations labelled "Standard Fortran" and "Standard Fortran - SSE FP Path" in Figures 4.1 to 4.4 are identical on the 64-bit machines. This is reflected in the Figures since the two compilations produce virtually identical performance results on each of the 64-bit machines. On the 32-bit machine, the two compilations differ and produce different

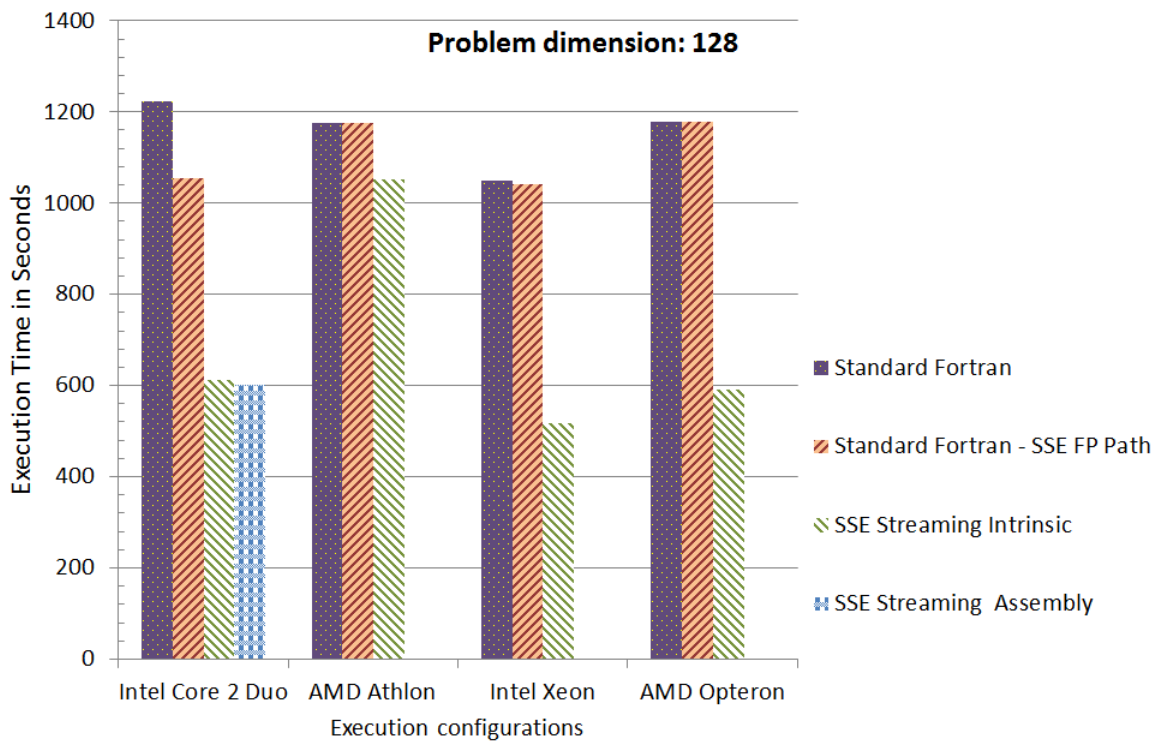


Figure 4.2: Performance results for the 128 × 128 × 128 problem size.

performance results. The default compilation of the standard implementation is outperformed by about 10% when the same implementation is compiled explicitly to use the SSE floating point path on the same 32-bit machine. This indicates that the default x87 floating-point unit is suboptimal and that instructing the compiler to execute floating point instructions using the SSE registers may in general provide a performance improvement without any code changes being required.

The SSE streaming implementations require a compilation script which separately compiles the Fortran wrapper code and the C code containing the optimised SSE streaming routines, and then combines them into a single executable. The command-line compilation script is also given in Figure 4.5.

When using SSE streaming with double precision, two arithmetic instructions are performed at once rather than one, so the theoretical ideal speedup is 2. On the 32-bit architecture, the SSE streaming implementations give a speed up 1.64 at a problem size of 64, increasing gradually to 1.75 for a problem size of 256. This is a reasonable increase in performance but falls short of ideal speedup. This is in comparison to the standard Fortran implementation compiled to use the SSE floating point path. On the Intel Xeon, the SSE streaming implementation provides a speedup of 2 compared

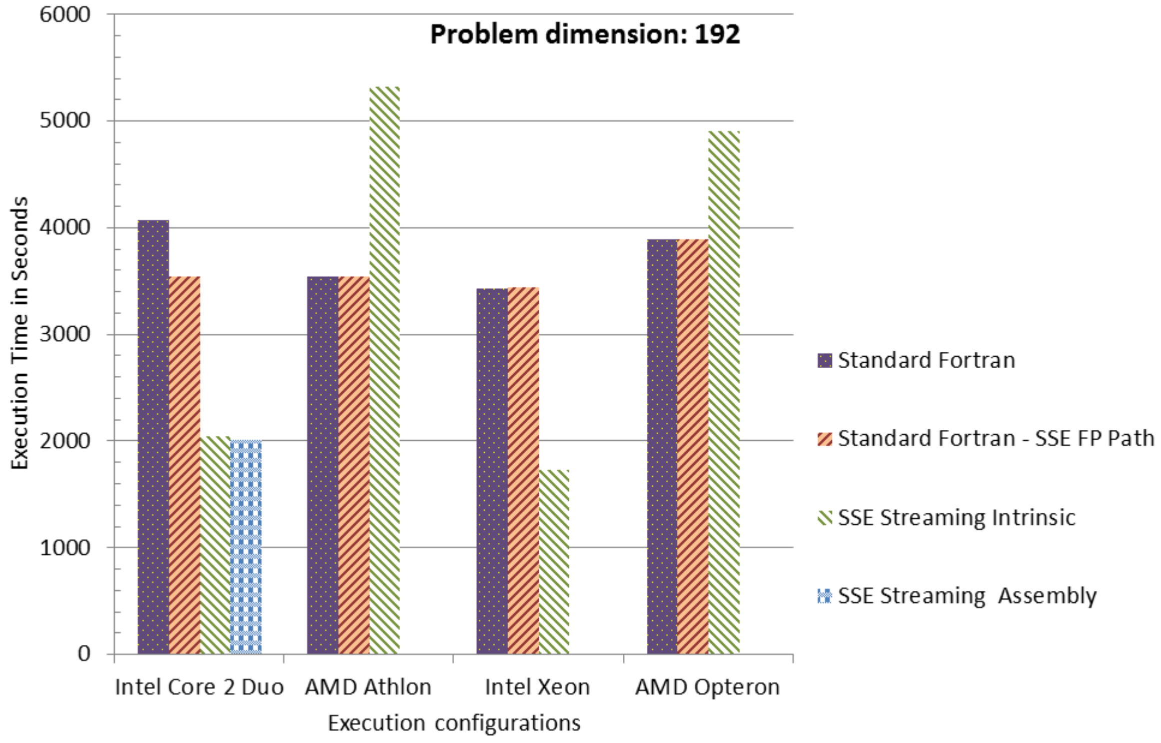


Figure 4.3: Performance results for the 192 × 192 × 192 problem size.

to the standard Fortran implementation (with either compilation since the standard implementation is the same for both compilations on this machine). Therefore on the Intel Xeon, the use of SSE instructions is able to deliver ideal speedup for this FDTD algorithm.

On the AMD Athlon machine, the SSE streaming implementation does not give the high levels of speedup seen on both Intel architectures. At problem sizes of 64 and 128 the performance of the SSE streaming implementation is very slightly better than the standard implementation, and at a problem size of 256 the performance is identical. At the problem size of 192, the SSE streaming implementation is significantly slower than the standard implementation.

The AMD Opteron achieves a speedup of 2 using the SSE streaming implementation at problem sizes of 64, 128 and 256, similar to the results on the Intel Xeon machine at these sizes. Execution on the AMD Opteron is in general a little slower than on the other architectures, which is not unexpected given that each core on the AMD Opteron has a slower clock speed than the other architectures, and the implementations are not exploiting the large number of cores available on the machine. The AMD Opteron suffers the same loss in performance observed on the AMD Athlon

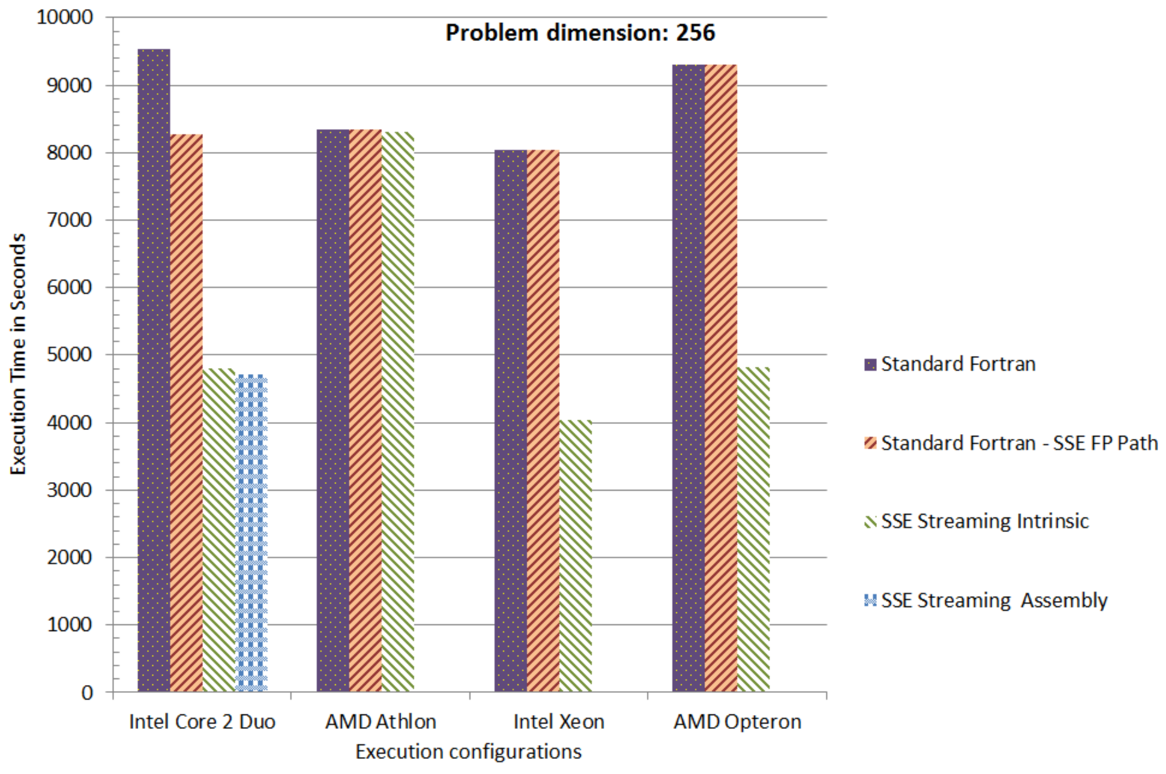


Figure 4.4: Performance results for the 256 × 256 × 256 problem size.

```
#For compiling Fortran using the default floating point path
#!/bin/bash
gfortran -O2 -o $1.x $1.f90

#For compiling Fortran using the SSE floating point path
#!/bin/bash
gfortran -O2 -mfpmath=sse -msse2 -o $1.sse.x $1.f90

#For integrating Fortran and C code into a single executable
#!/bin/bash
gfortran   -O2 -c -msse2 -mfpmath=sse -o $1_f.o $1.f90
gcc -c    -O2 -msse2 -mfpmath=sse -std=c99 -o $1_c.o $1.c
gfortran   -O2 -msse2 -mfpmath=sse -o $1.x $1_c.o $1_f.o
```

Figure 4.5: Compilation scripts for the CPU implementations

when executing the SSE streaming implementation at a problem size of 192. This loss of performance at a particular problem size on both AMD machines stands out as the most anomalous result in Figures 4.1 to 4.4 and is explored separately in Section 4.2.

At problem sizes of 64, 128 and 256, the AMD Athlon machine is the only one not to offer a large performance increase when using SSE streaming. The investigation into this observation focused on why the AMD Athlon machine did not offer the same performance benefits as the Xeon machine and AMD Opteron machine, since these machines also implement the 64-bit version of the x86 instruction set. The differing compiler versions were first considered. While the AMD Athlon uses GCC 4.3.0, the other two 64-bit machines use GCC 4.1.2. The assembly output that the compiler on each machine produces for the key parts of the SSE intrinsic function implementation was inspected. A summary of the instructions produced is shown in Table 4.2.

The instructions output by the compiler on each architecture are very similar. There is a difference between the AMD Athlon machine and the other machines in that the AMD Athlon machine produces a few extra *addq* instructions. These instructions are integer additions and should not cause the sort of performance difference observed in the results. In order to check this, the assembly output from the Intel Xeon machine was cross-compiled to execute on the AMD Athlon machine. The C code including the intrinsic functions was compiled to object code on the Intel Xeon machine, and this was combined with the Fortran wrapper on the AMD Athlon machine to be compiled into a complete program. This allowed a compatible binary for the AMD Athlon machine to be produced while the performance critical sections matched the assembly output produced by the compiler on the Intel Xeon machine. When executed on the AMD Athlon machine, the result was a slight degradation in performance compared to the native compilation. This supported the conclusion that the additional *addq* instructions were not responsible for the failure of the AMD Athlon machine to provide speedup when using the streaming SSE instructions.

Since the AMD Athlon machine has a newer version of the GCC compiler, one possibility considered was that the code was being subject to auto-vectorisation on this machine, but not on the others. As detailed in [31], the GCC compiler has the capability to detect loops which are suitable for SIMD operation and convert sequential loops into a structure suitable for vectorisation on the available SIMD hardware. This is known as auto-vectorisation due to the close relationship between SIMD and computing with vectors. If the compiler on the AMD Athlon was autovectorising the standard Fortran implementation by introducing SIMD instructions, it could result in a compiled program equivalent to the SSE streaming implementation, which would account for the lack of difference in performance. This possibility was rejected for several reasons. Auto-vectorisation requires the compiler to be able to detect the data independence within the loop to be converted to SIMD operation [31]. In the case of the FDTD method in use in this project, each iteration of each triply nested loop updates an element from each of three matrices, using elements from six other matrices. While the data independence may be easily apparent to a person with an understanding of the

Calculation of e values

Athlon		**Xeon**		**Opteron**	
Instruction	**Number of occurences**	**Instruction**	**Number of occurences**	**Instruction**	**Number of occurences**
addpd	3	addpd	3	addpd	3
movapd	7	movapd	7	movapd	7
movupd	2	movupd	2	movupd	2
addl	1	addl	1	addl	1
subpd	9	subpd	9	subpd	9
addq	12	addq	3	addq	3
divpd	6	divpd	6	divpd	6
mulpd	3	mulpd	3	mulpd	3
cmpl	1	cmpl	1	cmpl	1
jg	1	jg	1	jg	1
Total	**45**	**Total**	**36**	**Total**	**36**

Calculation of h values

Athlon		**Xeon**		**Opteron**	
Instruction	**Number of occurences**	**Instruction**	**Number of occurences**	**Instruction**	**Number of occurences**
movapd	12	movapd	12	movapd	12
movupd	2	movupd	2	movupd	2
addl	1	addl	1	addl	1
subpd	12	subpd	12	subpd	12
addq	12	addq	5	addq	5
divpd	6	divpd	6	divpd	6
mulpd	3	mulpd	3	mulpd	3
cmpl	1	cmpl	1	cmpl	1
jg	1	jg	1	jg	1
Total	**50**	**Total**	**43**	**Total**	**43**

Table 4.2: Summary of assembly instructions produced by the GCC compiler on each architecture.

FDTD method, it may not be apparent to a compiler without this domain knowledge. The FDTD method would not be a trivial case in which to apply autovectorisation. Additionally, the release notes of the GCC 4.1 series shows that autovectorisation was in place for that generation of the compiler [32]. Therefore the compilers on the Intel Xeon and AMD Opteron machines are also capable of autovectorisation. In fact, analysis of the assembly code produced by the compilers shows that no autovectorization took place on any of the machines, since no packed SSE instructions were present in the assembly code produced for the standard Fortran implementation on any machine.

An alternative cause of the performance difference is how the architectures execute the code. As shown in Table 4.1, the AMD Athlon processor has no level 3 cache, while the AMD Opteron and Intel Xeon each have a large level 3 cache. The Intel Xeon and AMD Opteron machines can therefore fit a much greater proportion of the data used in the algorithm into cache, which may result in better performance. However if this was the cause, it would be expected to affect the standard Fortran implementation as well as the SSE streaming implementation. This is not the case, since for the standard Fortran implementation, performance on the AMD Athlon is close to that of the Intel Xeon and AMD Opteron. If there was any general architectural benefit to the Xeon machine and AMD Opteron machine over the AMD Athlon machine for this FDTD algorithm, it should manifest itself in both the standard implementation and the streaming implementation since they are very similar apart from the use of SSE streaming instructions.

Since the compilers produce nearly identical code on all three 64-bit machines, and these three machines provide similar performance for the standard implementation, the performance difference of the SSE streaming implementation is considered likely to be caused by the speed with which each processor executes SSE instructions. As described in Section 2.2.2, the SSE2 instruction set is sufficient for implementing the FDTD algorithm. Table 4.1 shows that while the processor in the AMD Athlon processor implements SSE2, the Intel processors and the AMD Opteron processor implement variants of SSE4. The documentation of SSE3 and SSE4 only covers the additional instructions added beyond SSE2, there is no documented improvement in execution of the existing instructions. However, it may be that processors implementing more recent versions of SSE are able to execute all SSE instructions with higher performance. No documentation was found for any of the processors regarding the expected execution speed of SSE instructions, therefore it is difficult to form firm conclusions. However based on the investigations performed in this project, the most likely cause of the difference in performance between the AMD Athlon processor and the other three processors when executing the SSE streaming implementation is that the AMD Athlon processor has an inferior implementation of the relevant SSE instructions due to being of an older generation of SSE and requires more CPU time to execute the instructions.

4.2 Detrimental effects of memory allocation behaviour

The most unusual result in Section 4.1 is the SSE Streaming result of the AMD architectures in Figure 4.3. Both AMD machines execute the SSE streaming implementation significantly slower than the standard implementation. This is the only problem size at which this large drop in performance is observed in the results in Section 4.1. In order to understand the cause of this result, it is useful to observe the performance at a greater range of problem sizes. Figures 4.6 and 4.7 shows the performance of various implementations at all problem sizes from 16 to 256 at intervals of 2. Since packed SSE instructions require pairs of 64 bit operands which are aligned in memory on 16 byte boundaries, the SSE streaming implementations only works with problem sizes which are even numbers. Compared to the experiments described in Section 4.1, the number of time-steps for these experiments was reduced from 5000 to 1000, and the number of instances of each metric taken to form an average was reduced from 10 to 4. This was necessary in order to acquire all the data within the time-scales of this project.

Performance is defined as the execution time in microseconds divided by the cube of the problem size. The cube of the problem size gives the number of elements in each matrix, so this measurement of performance represents the amount of time spent by the processor on each point in the problem space. A smaller number therefore represents better performance. A line parallel to the x-axis would represent an implementation where the execution time scales linearly with the problem size. This is equivalent to stating that the time required to calculate the result for a single element remains constant as the problem size changes. On each architecture, the Standard Fortran implementation using the default floating-point path and the SSE Streaming implementation using intrinsics was executed.

Although originally executed to investigate the anomalous result on the AMD processors, the execution at a large range of problem sizes also reveals an interesting result on the Intel machines. Figure 4.6 shows that both the standard implementation and the SSE streaming implementation experience a sudden move to inferior performance at

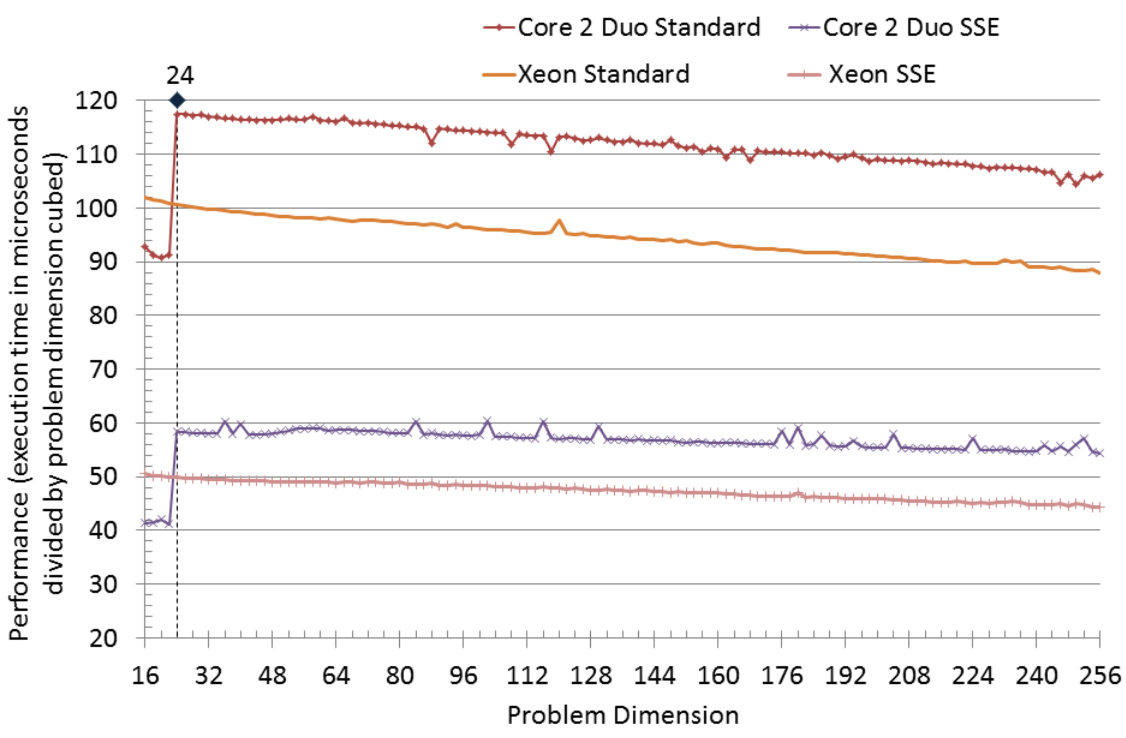


Figure 4.6: Performance of FDTD implementations on Intel architectures

a problem dimension of 24 on the Intel Core 2 Duo machine. Often when such drops in performance are observed in algorithms dealing with large data-sets, these are associated with the problem size reaching a point that the data no longer fit entirely into a level of cache, causing memory accesses which were not required at smaller problem sizes. However, the size of the matrices at problem dimensions of 22 and 24 do not correspond to any of the caches of the Intel Core 2 Duo processor. Figure 4.8 shows the cache organisation of each processor. For the standard Fortran implementation, the matrix size in bytes is calculated as $(Dimension + 2)^3 \times 8$. This is because the matrix is a cube with a one-element padding on each side and each double-precision floating-point value is 8 bytes. For a problem dimension of 22, this gives 110,592 bytes. This is well above the 32KB size of the level 1 data cache of the Intel Core 2 Duo processor but well below the 6MB size of the level 2 cache. In this case, data no longer fitting entirely into cache does not appear to be the cause of the sudden performance drop at a problem size of 24.

Analysing the memory allocation behaviour of a program using the Unix command *pmap* shows which areas of main memory are being used by the program. Using

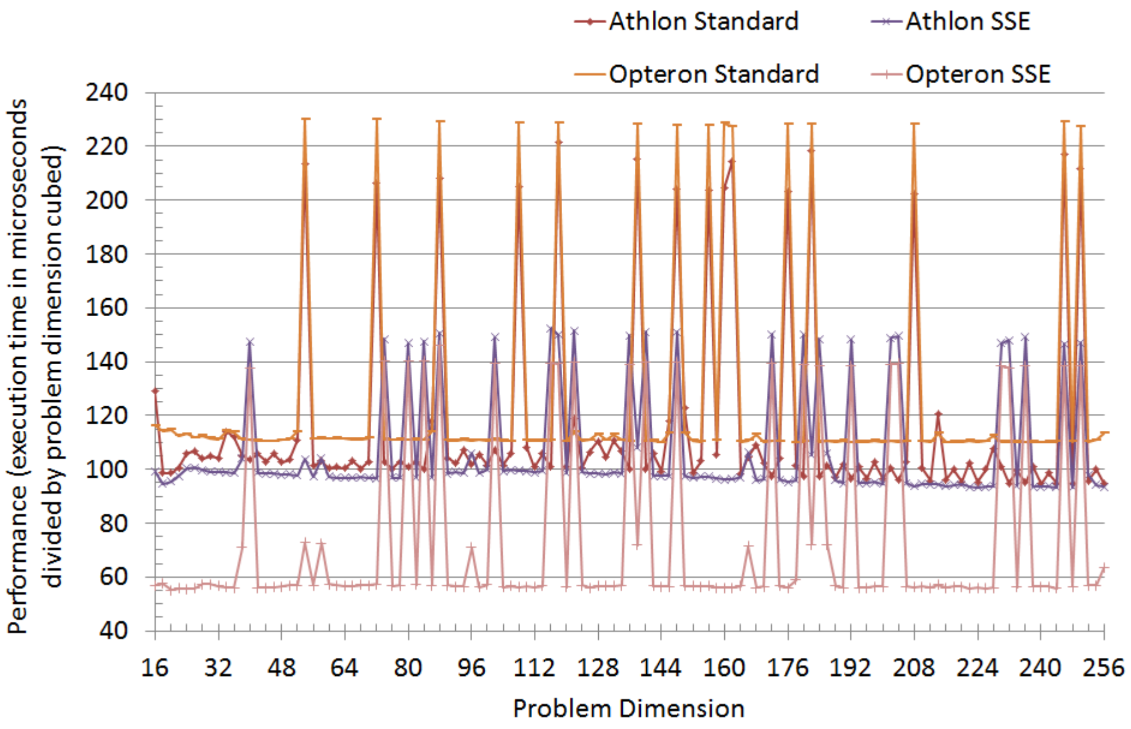


Figure 4.7: Performance of FDTD implementations on AMD architectures

this command, it was observed that at all problem sizes the standard Fortran implementation's machine code was stored at relatively low addresses, such as 0804800_{16} (hexadecimal). The program stack was allocated to a relatively high address such as $\mathrm{bfd8000}_{16}$. Repeated executions of the program on the Core 2 Duo machine show that this same memory allocation policy was used each time regardless of problem dimension. This was also the case for the SSE streaming implementation. The address of the memory block used to store the arrays representing the matrices E_x, E_y, E_z, H_x, H_y and H_z differs depending on problem size. This memory block is simple to identify among the output of the *pmap* command as it corresponds closely to the size of the arrays when calculated using $(Dimension + 2)^3 \times 8$. At problem sizes of 22 and below, the memory for the arrays is allocated immediately after the program code, with a low memory address. At problem sizes of 24 and above, the memory for the arrays is allocated just below the stack, at a relatively high memory address. This was verified using the Fortran function *LOC()* which gives the address in memory of an array. The memory addresses reported by this function corresponded to the observations taken from using the *pmap* command. The sudden drop in performance shown in Figure 4.6 corresponds exactly with the point at which memory allocation of the arrays changes from low memory addresses to high memory addresses. This could be due to different address translation requirements from virtual addresses to physical addresses, or could be due to the high memory addresses competing for space in the cache with other activity running on the machine which does not compete with the lower memory addresses. It was found that the same memory allocation behaviour occurs on the Intel Xeon machine, with the array allocation moving from high to low memory addresses depending on problem size. However as shown in Figure 4.6, the Intel Xeon does not suffer a sudden drop in performance at any point for either implementation. As shown in Table 4.1 and Figure 4.8 there are many differences between the software set-up and hardware organisation of the Intel Core 2 Duo and Intel Xeon machines. It appears that there is a particular characteristic of the Intel Core 2 Duo machine which causes the FDTD algorithms developed in this project to exhibit poorer performance when the arrays representing the matrices are stored near the stack rather than the program code, and that this characteristic is not shared by the Intel Xeon machine. It was not possible to identify the specific characteristic causing this effect. In real world applications of the FDTD method, a problem size much larger than 24 would typically be used. From a problem size of 24 and upwards, the Intel Core 2 Duo shows a fairly uniform gradual improvement in performance as the problem size increases, with the SSE streaming implementation performing about twice as well as the standard implementation at each problem size. This should be considered the useful performance level of this machine since the improved performance observed at smaller problem sizes is not useful to real world applications of the FDTD method. The Intel Xeon machine shows the same gradual improvement in performance and the same difference in performance between the SSE streaming and standard implementations, which corresponds to the speedup of 2 seen in the results in Section 4.1.

Figure 4.7 shows that both the standard and SSE streaming implementations show frequent and significant dips in performance at various problem sizes on both AMD machines. The most serious dips represent a 50% reduction in performance for the standard implementation on both architectures. The SSE streaming implementation suffers up to a 30% reduction in performance on the AMD Athlon and greater than 50% on the AMD Opteron. It can be seen that the drop in performance at a problem size of 192 observed in Figure 4.3 is not an isolated anomaly but rather one instance of a recurring loss of performance.

There is no immediately discernible pattern to the dips in performance in Figure 4.7. Since the dips in performance happen at particular problem sizes, but performance

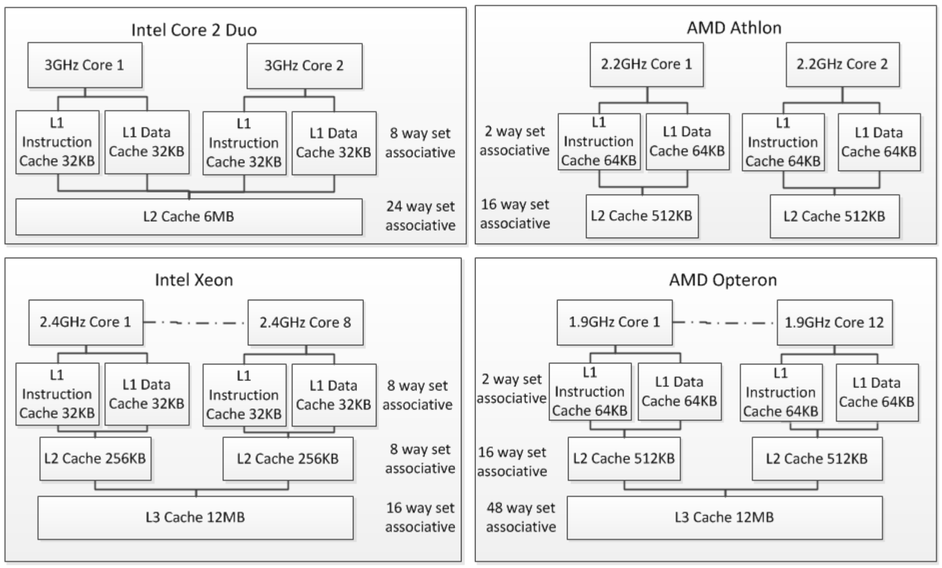


Figure 4.8: Cache organisation of the various processors

returns to normal for subsequent problem sizes, scalability of the algorithm is not the likely cause. Issues of scalability in high performance computing are characterised by a gradual loss in performance as the problem size grows and the execution performance fails to keep-up. The first hypothesis investigated as a possible cause for the sporadic dips in performance observed in Figure 4.7 was that the memory access pattern at particular problem sizes causes inefficient use of the cache.

As shown in Figure 4.8, the AMD processors each have a 64KB (kilo-byte) level 1 data-cache which is two-way set associative. Figure 4.9 illustrates the layout of a 64KB two-way set associative cache. Each 64B (byte) cache line can hold a contiguous 64B block from main memory. Figure 4.9 shows how a 64B line can hold eight double-precision values (since each is 8 bytes in size). Since the 64KB cache is split in to two sets, each 64B block in memory competes with other 64B blocks at 32KB intervals. In this context, “compete” means that separate 64B blocks from main memory with different addresses occupy the same cache line location. This is inevitable since cache is always much smaller than main memory. When a 64B block from main memory is moved into a line in the cache, the existing data in that line is evicted back to main memory. Where two or more 64B blocks would reside in the same cache line, and therefore would evict each other when loaded into cache, they are said to be competing. In the case of the AMD processors used in these experiments, the main memory address of the 64B block modulo 32768 identifies the cache line address to which that block should be stored. These cache-line addresses range from 0 to 32704 at 64B intervals. Since the level 1 data-cache is two-way associative, there are two positions for each cache line address and two competing blocks can reside in cache together, one in each set. However as soon as a third competing block is accessed, one of the existing blocks must be evicted from its cache line to make room for the new block.

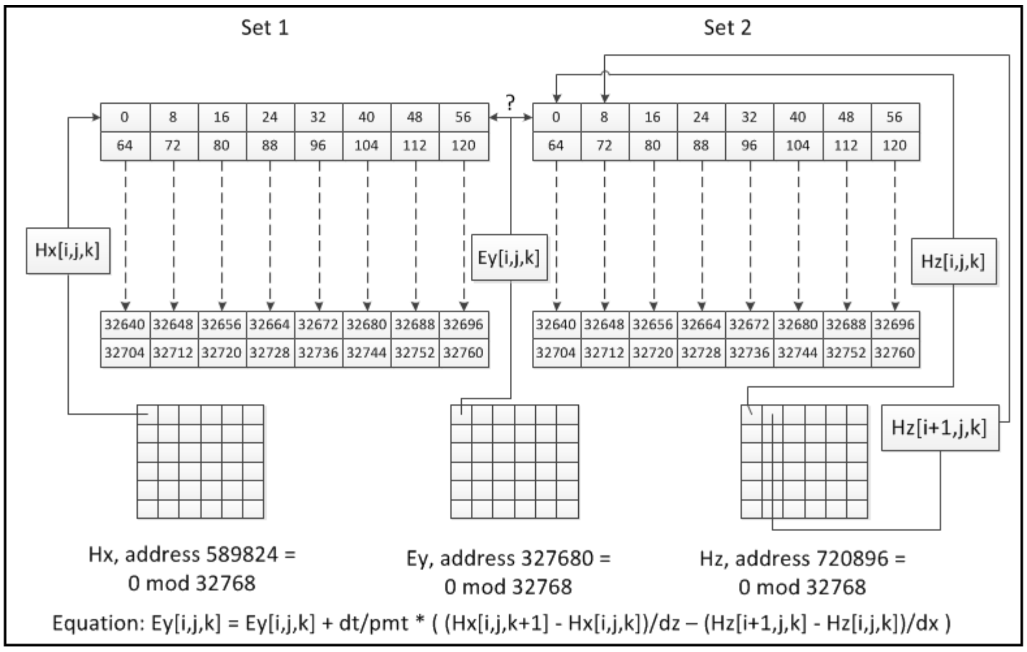

Set 1
Set 2
?
0 8 16 24 32 40 48 56
64 72 80 88 96 104 112 120
32640 32648 32656 32664 32672 32680 32688 32696
32704 32712 32720 32728 32736 32744 32752 32760
Hx[i,j,k]
Ey[i,j,k]
Hz[i,j,k]
Hz[i+1,j,k]
Hx, address 589824 = 0 mod 32768
Ey, address 327680 = 0 mod 32768
Hz, address 720896 = 0 mod 32768
Equation: Ey[i,j,k] = Ey[i,j,k] + dt/pmt * ((Hx[i,j,k+1] - Hx[i,j,k])/dz – (Hz[i+1,j,k] - Hz[i,j,k])/dx)

Figure 4.9: Level 1 cache behaviour of the AMD processors

The level 2 cache of both AMD processors is 512KB, and is 16-way set associative. Since the 512KB is split 16 ways, competing addresses at this level of cache also lie at 32KB intervals in memory. However since the level 2 cache has 16 sets to which each address can be cached it is more resilient to clashes, since 16 competing addresses would need to be accessed to evict a particular address from the cache. Similarly, the level 3 cache of the AMD Opteron processor is 48-way associative and therefore very resilient to clashes between competing cache addresses.

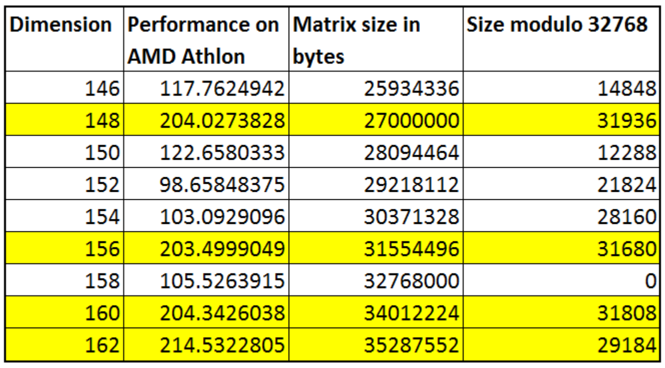
For the AMD processor-based systems used in these experiments, if the main memory addresses of the beginning of each of the six matrices E_x, E_y, E_z, H_x, H_y and H_z happened to give the same result modulo 32768 (32KB), the matrices can be said to be aligned with respect to the level 1 data-cache. When this alignment is present,the 64B blocks starting at $E_x(0,0,0)$, $E_y(0,0,0)$, $E_z(0,0,0)$, $H_x(0,0,0)$, $H_y(0,0,0)$ and $H_z(0,0,0)$ will all resolve to the same cache line address. Since all six matrices are identical in size and structure, any two elements with the same indices but from different matrices would compete for the same position in cache when the matrices are aligned.

Equations 2.15 to 2.20 show that in the FDTD algorithm, the calculation for a single element of a particular matrix is dependent on the values of four elements from other matrices. In each equation, two of these four elements share the same indices as the dependent element being calculated. The algorithm single steps through memory, meaning that the addresses used to calculate a particular element are only 8 bytes away from the addresses used to calculate the previous element. Since each cache line is 64B, the eviction of a cache line removes eight double-precision floating-point values from cache, data required by up to the next 7 steps in the algorithm. If the matrices are aligned modulo 32768, the elements with the same indices during each step through the equations of the FDTD algorithm would all target the same cache line address and frequent cache evictions would occur as these elements are accessed. At the next step through the equations the number of cache misses and accesses to main memory would be increased due to the eviction of data during the previous step.

Figure 4.9 shows how the cache line eviction may occur in the level 1 data-cache of the AMD processors when executing a calculation for an element of E_y. When calculating $E_{y(i,j,k)}$, the values of $H_{x(i,j,k)}$ and $H_{z(i,j,k)}$ are required operands. In the example shown in Figure 4.9, the memory addresses of these values both resolve to the first cache-line, and therefore each is loaded into one of the two sets available. In addition, the value of $H_{z(i+1,j,k)}$ used in the calculation is loaded since it is adjacent to $H_{z(i,j,k)}$ and resides in the same 64B block of main memory. When the value of $E_{y(i,j,k)}$ is subsequently required to complete the calculation, it must be placed in the first cache line of one of the two sets. This will lead to either the values of $H_{x(i,j,k)}$ to $H_{x(i+7,j,k)}$ or $H_{z(i,j,k)}$ to $H_{z(i+7,j,k)}$ being evicted, even though these values will be required in several calculations immediately following the calculation of $E_{y(i,j,k)}$ being considered in Figure 4.9. The resulting cache misses lead to main memory accesses and impair the performance of algorithms. This could account for the observed loss in performance at certain problem sizes shown in Figure 4.7.

In order to test this hypothesis, it was necessary to compare the problem sizes at which a loss of performance occurs to problem sizes at which memory alignment occurs. To determine whether the matrices are aligned with respect to the level 1 data-cache, the size of each matrix in bytes was considered. Assuming that each matrix is assigned contiguously in memory to the others, then any problem size which leads to a matrix whose size divides exactly by 32768 would lead to the cache alignment problem.

For the standard Fortran implementation, matrix size in bytes is calculated as $(Dimension + 2)^3 \times 8$ as before. Table 4.3 shows this for a subset of problem sizes. Those problem sizes showing a loss in performance are highlighted.



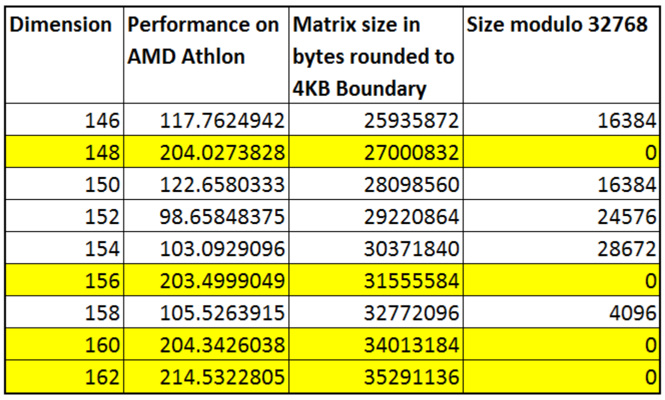
Dimension	Performance on AMD Athlon	Matrix size in bytes	Size modulo 32768
146	117.7624942	25934336	14848
148	204.0273828	27000000	31936
150	122.6580333	28094464	12288
152	98.65848375	29218112	21824
154	103.0929096	30371328	28160
156	203.4999049	31554496	31680
158	105.5263915	32768000	0
160	204.3426038	34012224	31808
162	214.5322805	35287552	29184

Table 4.3: Matrix size information at various problem sizes for the standard Fortran implementation

Table 4.3 shows that matrix sizes with poorer performance do not divide perfectly by 32768. In fact, the only problem size in the table which does produce 0 modulo 32768 is a size of 158, which is not one of those experiencing loss in performance. Those experiencing performance loss are those that come close to being a multiple of 32768 bytes without quite doing so. They have the highest values modulo 32768 meaning they are close to wrapping around to 0, which would lead to perfect alignment of the matrices. Almost perfect alignment does not cause the caching effects described above, as it would not bring elements with the same indices from different matrices to the same cache line[1].

The conclusions drawn from Table 4.3 are based on the assumption that each matrix is assigned to memory in a directly contiguous fashion to the other matrices. Analysis of the addresses assigned to the matrices on the AMD machines shows that this is not the case. Each matrix is assigned to start in memory slightly beyond the end of the previous matrix, aligned to the next 4KB (4096 bytes) boundary. Where the matrices would align perfectly to a 4KB boundary (such as a size of 158), an extra 4KB space is left between each matrix. For those cases in the Table 4.3 where the alignment is close to an exact multiple of 32KB, the next 4KB boundary is also a 32KB boundary. The

[1]Note that since the sizes of the matrices in Table 4.3 are much bigger than the size of the caches in the AMD processor, cache eviction will occur regardless of alignment. However, it is only when matrices are aligned in memory that the repeated eviction of values required by the immediately following steps in the algorithm occurs. When matrices are not aligned, eviction occurs on cache lines which are likely to have been utilised earlier and which are no longer useful to the algorithm.



Dimension	Performance on AMD Athlon	Matrix size in bytes rounded to 4KB Boundary	Size modulo 32768
146	117.7624942	25935872	16384
148	204.0273828	27000832	0
150	122.6580333	28098560	16384
152	98.65848375	29220864	24576
154	103.0929096	30371840	28672
156	203.4999049	31555584	0
158	105.5263915	32772096	4096
160	204.3426038	34013184	0
162	214.5322805	35291136	0

Table 4.4: Matrix size information adjusted to compensate for alignment to 4KB boundaries

memory assignment is being rounded up to cause perfect 32KB alignment, while when 32KB alignment would occur in a contiguous assignment, the rounding up is breaking the alignment. Calculating matrix size again but adjusting for a shift of memory allocation to the next 4KB boundary, we find that those dimensions which lead to dramatic loss in performance correlate perfectly with those dimensions calculated to be aligned on a 32KB boundary. Table 4.4 shows the updated calculations.

In addition to the serious drop in performance caused by 32KB alignment, it is observed that those dimensions which cause alignment to a 16KB boundary also suffer a drop in performance. This is the case for problem sizes of 146 and 150 in Table 4.4. The loss in performance is less severe when this "half alignment" occurs but is still noticeable. In this case each matrix is 32KB aligned with half of the other matrices, so less cache contention occurs. Figure 4.10 isolates the performance result for the standard Fortran implementation on the AMD machines, and shows the problem sizes which incur perfect alignment to 32KB boundaries and half alignment to 16KB boundaries.

Figure 4.10 shows that severe drops in performance correlate exactly with perfect alignment to 32KB boundaries, while the lesser drops in throughput correlate closely with half alignment to 16KB boundaries. This indicates very strongly that alignment in memory, leading to excessive eviction of cache lines, is the cause of the drop in performance.

The SSE streaming implementation exhibits the same relationship between alignment and loss of performance, however the alignment occurs at different points since the algorithm uses an extra layer of padding in the *i* dimension, as described in Section 3.2. Table 4.5 shows a sample of problem sizes leading to alignment for the SSE

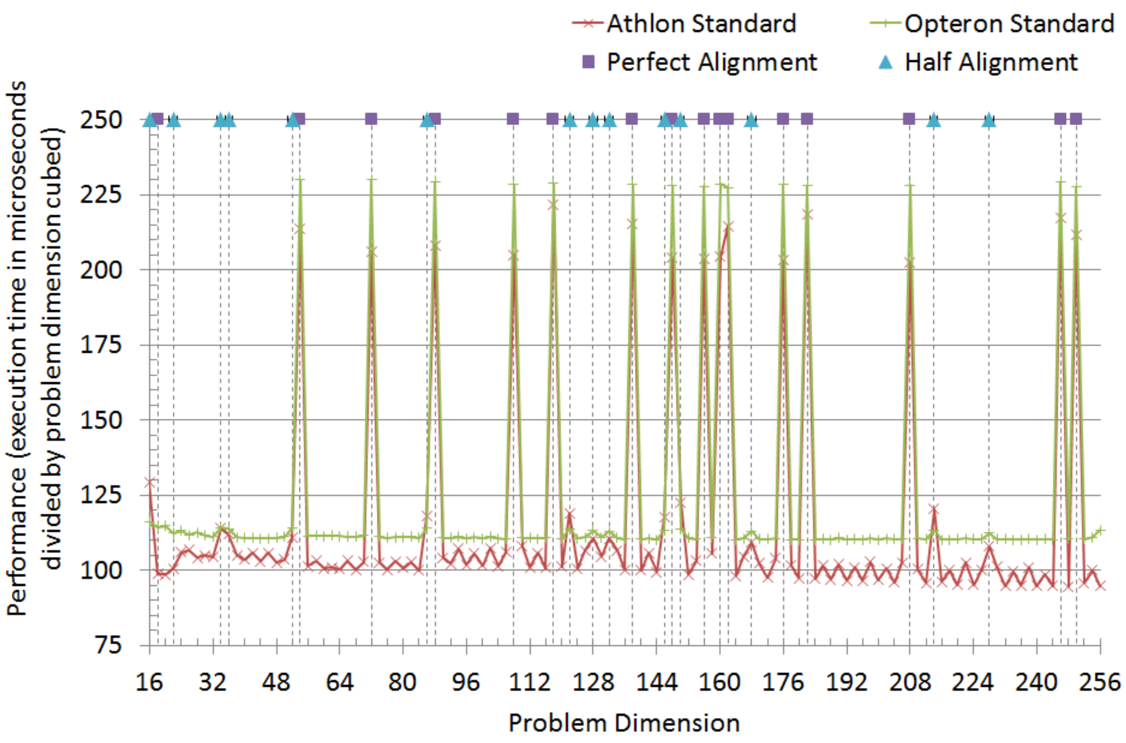


Figure 4.10: Performance of the standard Fortran implementation on the AMD machines, cross-referenced with alignment to 16KB and 32KB boundaries

streaming implementation. In a similar fashion to the information shown in Table 4.4 for the standard implementation, it is observed for the SSE streaming implementation that when alignment to a 32KB boundary occurs, the performance suffers a significant reduction.

Figure 4.11 shows the performance of the SSE streaming implementation on the AMD machines, and shows the alignment points in this case. This demonstrates that the larger drops in performance correlate exactly with those problem sizes causing perfect alignment and the smaller drops in performance correlate exactly with those problem sizes causing half alignment.

A possible solution to the memory alignment problem is to introduce memory allocation logic into the code in order to increase the space between each matrix and ensure that alignment does not occur. This needs to be done selectively, otherwise while alignment will be removed for some problem sizes it will be introduced to other problem sizes which previously did not suffer from it. A solution should therefore test whether the size of the matrices is such that alignment would occur before taking any remedial action.

Dimension	Performance on AMD Athlon	Matrix size in bytes rounded to 4KB Boundary	Size modulo 32768
230	146.854093	100761600	0
232	148.0579584	103383040	0
234	93.86835139	106045440	8192
236	149.1664933	108756992	0
238	93.58900629	111513600	4096
240	93.59953161	114319360	24576

Table 4.5: Matrix size information for the SSE streaming implementation

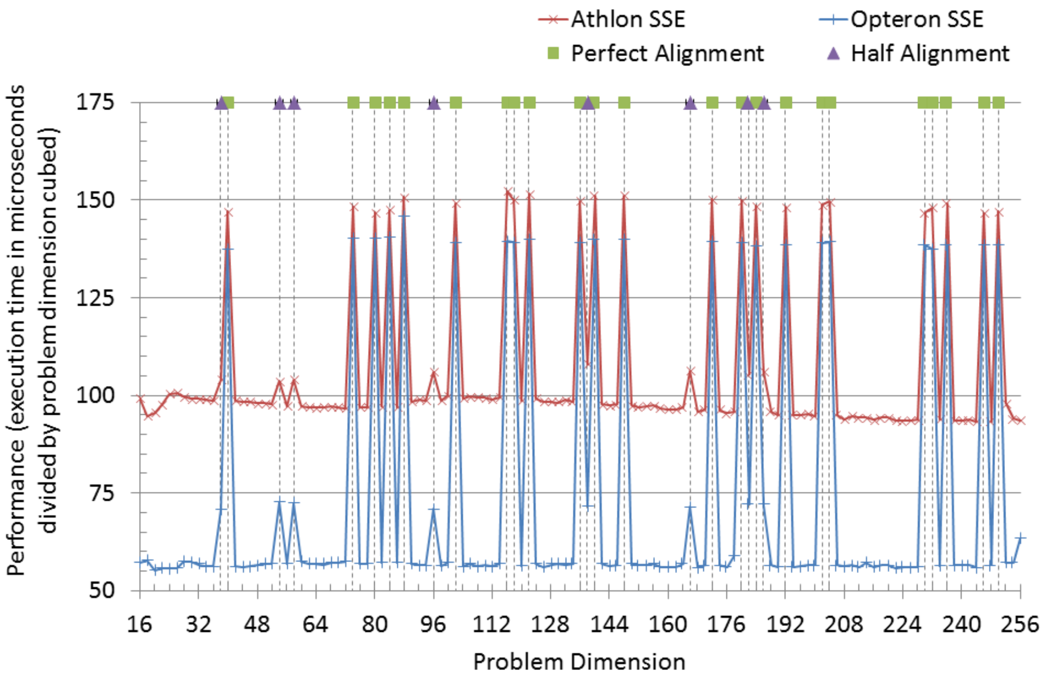


Figure 4.11: Performance of the SSE streaming implementation on the AMD machines, cross-referenced with alignment to 16KB and 32KB boundaries

Having calculated that a particular problem size would cause alignment and requires adjustment, the simplest solution is to allocate some unneeded memory of greater that 4KB between each matrix, forcing misalignment. In order to reduce excessive memory usage, it is desirable to use as little memory as possible in order to achieve this. Unfortunately, experimentation with Fortran memory allocation showed that large data structures get assigned elsewhere in memory than relatively small ones. This meant that for small problem sizes such as 20, alignment could be prevented by including statements allocating memory for redundant 4KB data structures between the statements allocating memory for the matrices required by the algorithm. However with larger problem sizes such as 100, the redundant 4KB structures are much smaller than the matrices and are allocated to a different part of memory so do not cause the required misalignment effect.

An alternative approach is to add redundant elements to the beginning of each matrix. In Fortran, array indices can begin at any integer including negative numbers. For all problem sizes which would suffer from perfect or half alignment, extra space is allocated at the beginning of each matrix without affecting the algorithm. The extra space is given negative indices and therefore is ignored by the code which traverses the matrices using positive indices. This approach was implemented using a simple method which sets the starting index in one axis to the negative of one eighth the dimension for the problem size when memory alignment is detected. It was found that for larger matrices, an extra 8th in size causes a significant increase in the amount of memory used. In these cases, a single extra layer in one dimension is sufficient to increase the size of a matrix by more than 4KB. Using logic to determine when a single layer would be sufficient, the two techniques were combined and the code for this is shown in Figure 4.12.

Figure 4.13 shows the performance results for the standard and SSE streaming implementations on the AMD machines after the memory alignment fix was in place.

There are many fewer instances of loss of performance with the memory alignment fix in place. Where the drop in performance still occurs, this is caused by problem sizes where adding an extra layer alters the size of the matrices by an exact 32KB amount. This is the case for example with a problem size of 88. The original size of a matrix at this problem size for the standard implementation is $(88 + 2)^3 \times 8 = 5832000$ bytes. This result rounds up to 5,832,704 bytes as the next 4KB boundary, which divides exactly by 32768 to give 178. An extra layer at this size adds $(88+2)^2 \times 8 = 64800$ bytes for a total of 5,896,800 bytes. This rounds to 5,898,240 bytes as the next 4KB boundary which divides exactly by 32768 to give 180. The single extra layer leads to matrices which are still assigned at 32KB intervals in memory so fails to prevent alignment with respect to the level 1 data-cache on the AMD machines and the corresponding drop in performance.

More complex algorithms for preventing memory alignment could be pursued, but these would lead to increasingly complex logic for assigning memory to matrices, harming the readability and reusability of the code. Moreover, such fixes would only be relevant to processors with the particular cache size and organisation being used in the two AMD processors described here. Other processors with different cache

```
!m_size is total bytes used in one matrix, rounded to
!4KB boundary
m_size= ceiling(imax**3*8/4096.0)*4096
!allocate with padding if alignment
! or half alignment will occur
if(modulo(m_size,32768) .eq. 0 .or. &
& modulo(m_size,32768) .eq. (32768/2))then
!for small matrices, add an extra 8th of volume
     if((imax)**2<4096) then
      allocate(ex(imin:imax,jmin:jmax,(-kmax/8):kmax))
      allocate(ey(imin:imax,jmin:jmax,(-kmax/8):kmax))
      allocate(ez(imin:imax,jmin:jmax,(-kmax/8):kmax))
      allocate(hx(imin:imax,jmin:jmax,(-kmax/8):kmax))
      allocate(hy(imin:imax,jmin:jmax,(-kmax/8):kmax))
      allocate(hz(imin:imax,jmin:jmax,(-kmax/8):kmax))
!for large matrices, one extra layer of padding is sufficient
     else
      allocate(ex(imin:imax,jmin:jmax,(kmin-1):kmax))
      allocate(ey(imin:imax,jmin:jmax,(kmin-1):kmax))
      allocate(ez(imin:imax,jmin:jmax,(kmin-1):kmax))
      allocate(hx(imin:imax,jmin:jmax,(kmin-1):kmax))
      allocate(hy(imin:imax,jmin:jmax,(kmin-1):kmax))
      allocate(hz(imin:imax,jmin:jmax,(kmin-1):kmax))
     end if
!allocate as normal if alignment will not occur
else
      allocate(ex(imin:imax,jmin:jmax,kmin:kmax))
      allocate(ey(imin:imax,jmin:jmax,kmin:kmax))
      allocate(ez(imin:imax,jmin:jmax,kmin:kmax))
      allocate(hx(imin:imax,jmin:jmax,kmin:kmax))
      allocate(hy(imin:imax,jmin:jmax,kmin:kmax))
      allocate(hz(imin:imax,jmin:jmax,kmin:kmax))
end if
```

Figure 4.12: Allocating extra memory when initializing the matrices to avoid alignment to the L1 cache

sizes and different cache allocation policies would exhibit different behaviour and require their own solutions. Additionally, other compilers may use different libraries to handle memory allocation and always allocating data structures to start at 4096 byte boundaries may not be the behaviour of all such libraries. Due to the complexity of implementing memory alignment prevention which would work at all problem sizes, and due to the lack of portability that such a solution would offer, a more practical solution may be to be aware of those dimensions which cause cache contention due to alignment on a particular machine, and avoid those sizes. For example, the standard implementation suffers severe loss of performance at a dimension of 208 on the AMD machines. By adjusting to a problem size of 204 or 212, performance is returned to normal. In typical applications of the FDTD method, such an adjustment would not invalidate the results.

Figure 4.6 shows that the Intel machines do not exhibit the same pattern of performance drops seen with the AMD machines. This indicates that alignment leading to

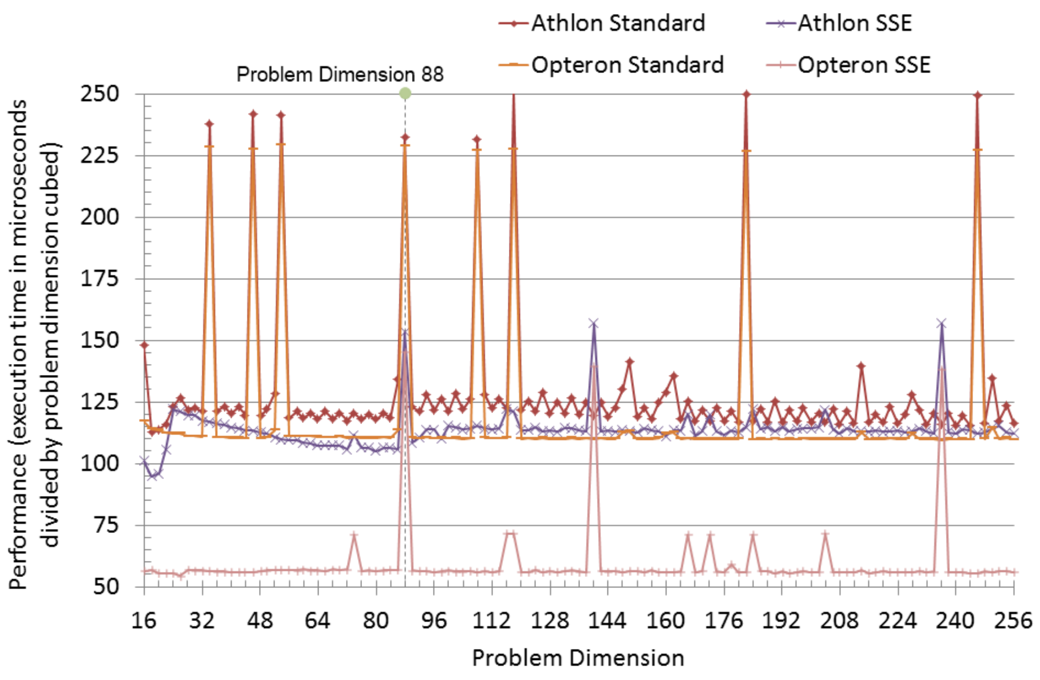
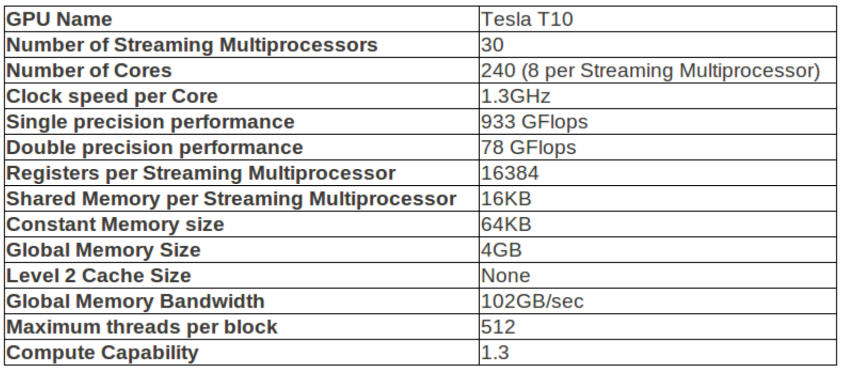


Figure 4.13: Performance of on the AMD machines following the memory alignment fix

excessive cache evictions is not an issue on the Intel machines. This is in spite of the Intel machines having level 1 data-caches which are half the size of those on the AMD machines, as shown in Figure 4.8. The level 1 data-caches on the Intel machines are 8-way set associative rather than 2-way. This means that the 32KB cache is split 8 times, into 8 groups of 4096 bytes. Competing memory addresses would therefore be found at 4KB intervals in memory. This is much more frequent than the 32KB interval of the AMD level 1 data caches. In fact, the Intel machines have the same policy of aligning the data structures to 4KB boundaries in main memory, so matrices are aligned with respect to the level 1 data-cache at every problem size. For a particular *(i,j,k)* index, the element with that index from each matrix would correspond to the same level 1 cache line address at every problem size on the Intel machines. However, because the level 1 data-cache is 8-way set associative, each cache line address corresponds to 8 slots in the cache. Since there are only 6 matrices in the FDTD algorithm, the 64B cache lines containing the elements indexed by a particular *(i,j,k)* index from each matrix can all reside in cache simultaneously, using 6 of the available 8 positions for that cache line address. For the FDTD algorithm being used in this project, an 8-way set associative data cache is more beneficial than a 2-way set associative data cache in terms of avoiding excessive eviction of useful data from cache caused by alignment of the matrices in memory.

4.3 Results of GPGPU Experiments

The CUDA implementations described in Section 3.4 were executed on a machine containing two NVIDIA Tesla T10 CUDA processors. The specification of a single Tesla T10 GPU is shown in Table 4.6. The information in this table is taken from executing the "enum_gpu" program from chapter 3 of [20] and from the information published by NVIDIA in [33].



GPU Name	Tesla T10
Number of Streaming Multiprocessors	30
Number of Cores	240 (8 per Streaming Multiprocessor)
Clock speed per Core	1.3GHz
Single precision performance	933 GFlops
Double precision performance	78 GFlops
Registers per Streaming Multiprocessor	16384
Shared Memory per Streaming Multiprocessor	16KB
Constant Memory size	64KB
Global Memory Size	4GB
Level 2 Cache Size	None
Global Memory Bandwidth	102GB/sec
Maximum threads per block	512
Compute Capability	1.3

Table 4.6: Specification of a Tesla T10 GPU

Each Tesla T10 is a CUDA capable GPU processor supporting CUDA compute capability 1.3. Compute capabilities describe the generations of NVIDIA's CUDA capable GPUs. Capability 1.3 means that the processor provides double precision support, but not at the levels of performance of the more recent Fermi architecture (compute capability 2.0) graphics processors [22] [34]. As shown in Table 4.6, the stated double precision performance of 78 GFlops (Giga Floating Point Operations Per Second) is well below the single-precision performance of 933 GFlops. Each T10 contains 30 streaming multiprocessors. As described in Section 2.3.1 a single streaming multiprocessor has 8 cores, meaning that each Tesla T10 contains 240 cores in total. The implementations described in Section 3.4 were designed for use on a single GPU, so only one of the two available GPUs was used during execution of these experiments.

As with the CPU experiments performed in Section 4.1, the GPU implementations were executed for 5000 time-steps at problem sizes of 64, 128, 192 and 256. Each metric was taken ten times and the average used as the timed result. The performance results when using double precision arithmetic are shown in Figures 4.14 to 4.17. In each case, the performance of the SSE streaming implementation on the Intel Xeon machine (as described in Section 4.1) is used to provide a comparison between CPU and GPU performance. This represents the highest performing CPU implementation based on the results in Section 4.1. The sequential GPU implementation described in Section 3.4 is not included as it was not designed to offer high performance and takes impractically long to execute at these problem sizes.

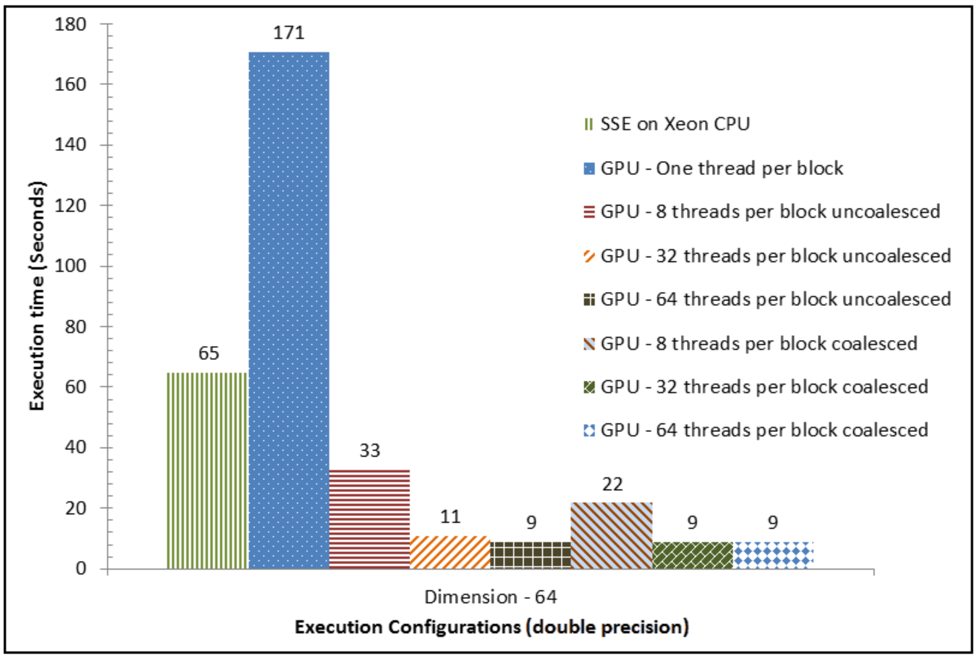


Figure 4.14: Double precision performance of GPU implementations at a problem size of 64

At each problem size, GPU implementations with increasing thread counts per block are used. Thread counts of 1, 8, 32 and 64 are used in each case. Where more than one thread per block is used, the results for both coalesced and uncoalesced memory access are given. At the larger problem sizes, even greater thread counts were possible. At the problem size of 128, 128 threads per block was executed. At the problem size of 192, 192 threads per block was executed. At the problem size of 256, 128 threads per block and 256 threads per block were executed. Note that for the approach used in these GPU implementations, it is necessary for the problem size to be an exact multiple of the thread count, and the thread count may not exceed the problem size.

At each problem size, the single thread per block GPU implementation is around

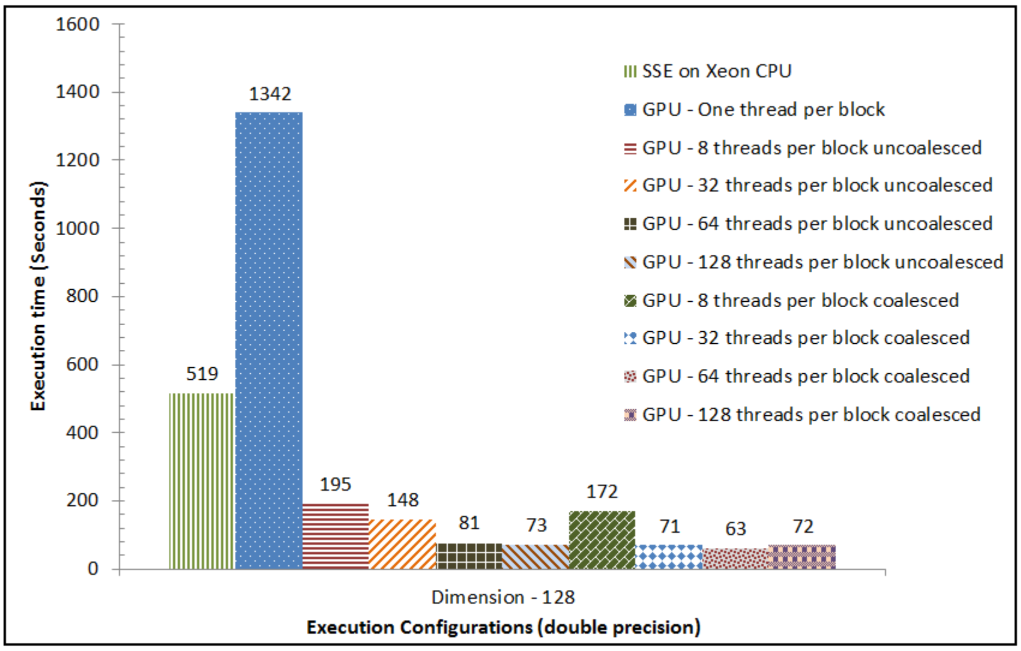


Figure 4.15: Double precision performance of GPU implementations at a problem size of 128

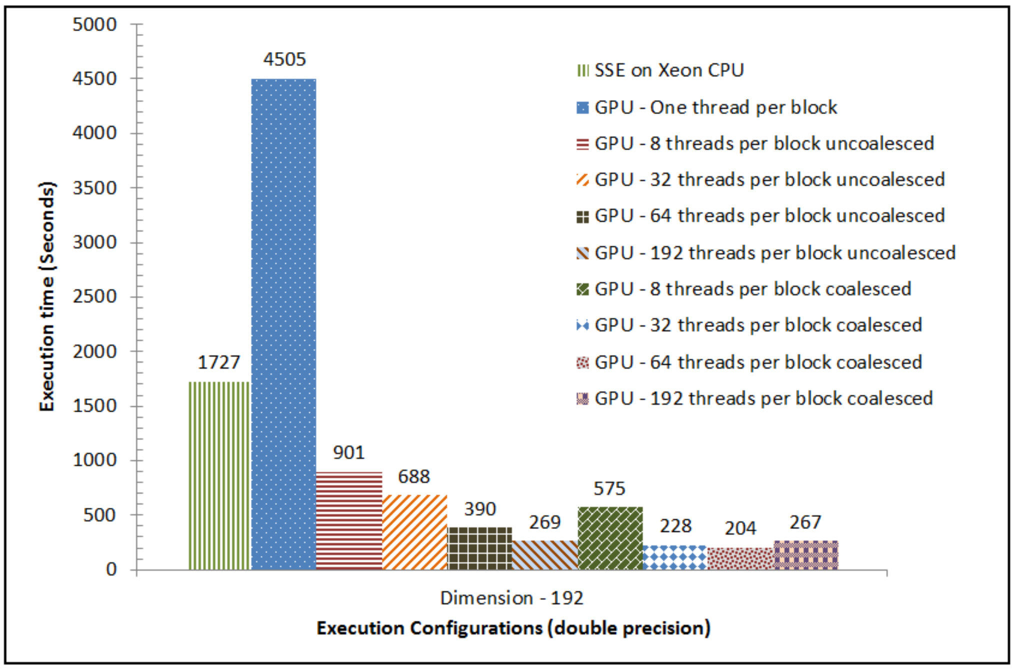


Figure 4.16: Double precision performance of GPU implementations at a problem size of 192

2.5 times slower than the SSE streaming CPU implementation. In this case, exploiting parallelism at the block level alone is not sufficient to get acceptable performance from

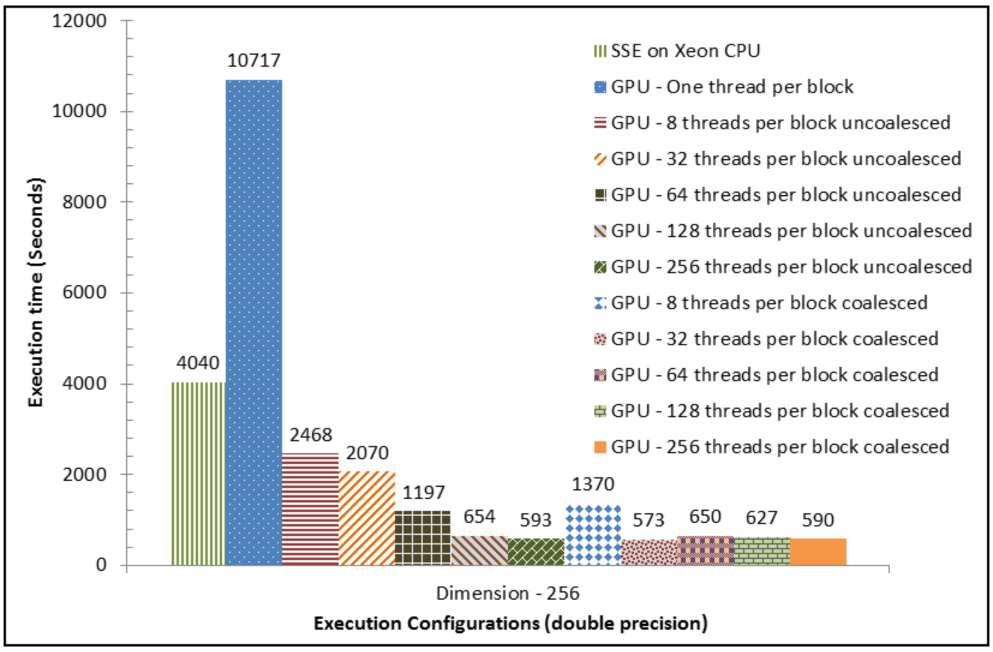


Figure 4.17: Double precision performance of GPU implementations at a problem size of 256

the GPU. When uncoalesced memory access is used, increasing the thread count consistently improves performance at each problem size. Coalesced memory access gives higher performance than uncoalesced memory access at the lower thread counts. This difference gradually diminishes as the thread count increases. At each problem size, the highest thread count gives very near equal performance between uncoalesced and coalesced memory access. As described in Section 3.4, the coalescing approach relates to how the elements in the *i* dimension of each matrix are assigned to the threads. When the thread count is equal to the number of items in the *i* dimension, as is the case at the largest thread count shown for each problem size in Figures 4.14 to 4.17, both the coalesced and uncoalesced approaches assign one element per thread. This means the behaviour at run-time of the coalesced and coalesced implementations is almost identical, with only slight differences in the calculation used to determine the index for each thread. Therefore the similarity in execution time when the thread count equals the problem size is to be expected.

The best performing implementation varies with problem size. At the problem size of 64, there are three implementations providing the best execution time of 9 seconds. At the problem size of 128, it is the coalesced approach using 64 threads per block (so each thread is assigned two elements in the *i* dimension of each matrix) which provides the best performance. At the problem size of 192, it is also the coalesced approach using 64 threads per block (in this case each thread is assigned three elements in the *i* dimension of each matrix) which provides the best performance. At the problem size of 256, it is the coalesced approach with 32 threads per block (meaning 8 elements per thread) which performs best. It is therefore not possible to provide a single solution which gives the best performance at all problem sizes. However, it can be seen that when using coalesced memory accesses the results for 32 threads and above differ by only a few percent from each other. If a uniform solution was required, these results show that using the coalesced approach with a 32 or 64 threads per block gives an implementation which will execute at each problem size and give performance close to the highest observed.

Compared to the best CPU implementation, the best GPU implementation gives a speedup of between 7 and 8.5 at each problem size. While the CPU implementation uses the SSE registers to execute in parallel, it uses only a single CPU core. Modern CPUs typically contain multiple cores, and [6] shows that multi-core versions of the FDTD algorithm can provide speedup with efficiency of greater than 80%. If multi-core execution was applied to the SSE streaming CPU implementation with this level of efficiency, it would require 8 cores to provide performance comparable to the best GPU implementations presented here. Typical modern desktop processors have up to 4 cores but rarely 8 or more. Therefore it could be expected that a modern desktop containing a CUDA capable GPU would execute the FDTD algorithm with higher performance using the GPU rather than the CPU. However, servers with 8 cores or more are now common place and therefore in these environments, a multi-core CPU implementation may offer higher performance than the GPU implementation. This is conjecture based on the ability execute with SSE streaming in parallel on multiple cores, which would be a useful follow up to the implementation described in Section 3.3.

As described in section 2.3.1, CUDA capable GPUs are expected to give significantly better single precision performance than double precision. To test this, the same experiments were performed using single precision versions of the GPU implementations. To provide a useful comparison, a single precision version of the SSE streaming with intrinsics implementation was also developed, and executed on the Intel Xeon machine. As described in Section 2.2, single-precision SSE instructions can perform four floating-point operations at once, compared to two operations for double-precision.

The SSE streaming implementation therefore had to be altered to step through the matrices 4 elements at a time rather than two, and the intrinsic functions for performing packed double-precision instructions were replaced with intrinsic functions for packed single-precision instructions. These were the only changes required to develop a correct single-precision implementation. The results of the single-precision experiments are shown in Figures 4.18 to 4.21.

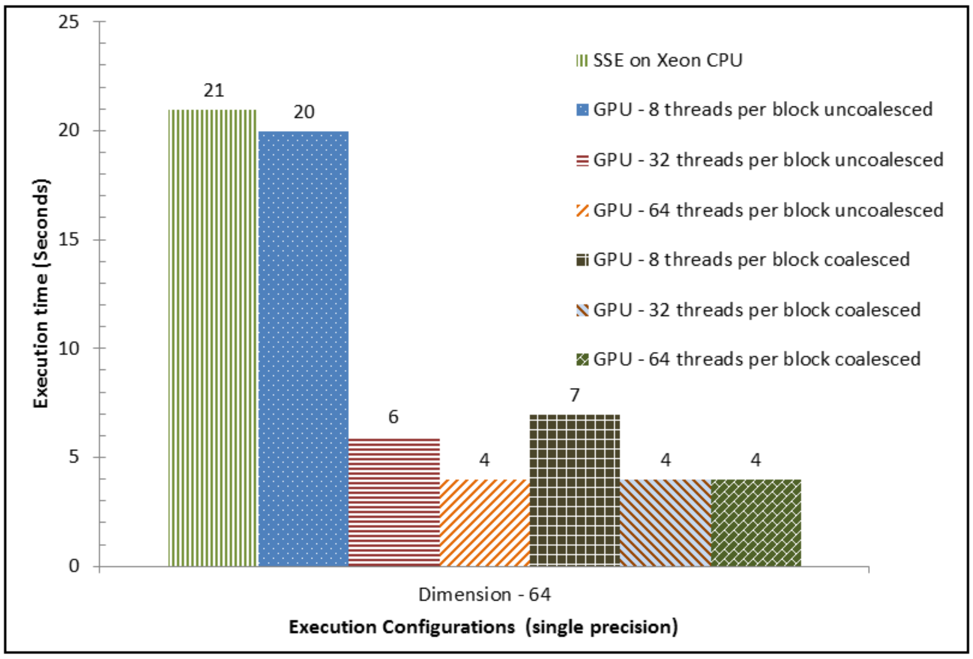


Figure 4.18: Single precision performance of GPU implementations at a problem size of 64

At each problem size, the single-precision SSE streaming CPU implementation is about 3 times faster than the equivalent double-precision CPU implementation. Using single-precision to perform 4 operations at once rather than two operations doubles throughput, and the additional speedup beyond the doubling of throughput is likely due to the lower memory bandwidth required by single-precision. Since a single-precision floating-point value is only 4 bytes rather than 8 bytes, a single cache line contains twice as many single-precision values, effectively halving the cache miss rate when stepping sequentially through memory as is the case with this algorithm.

In general, the pattern of performance for the single precision GPU implementations is quite similar to that of double precision. Increasing the thread count is more beneficial to uncoalesced than coalesced performance, and coalesced performance is generally better than uncoalesced performance, especially at lower thread counts. The

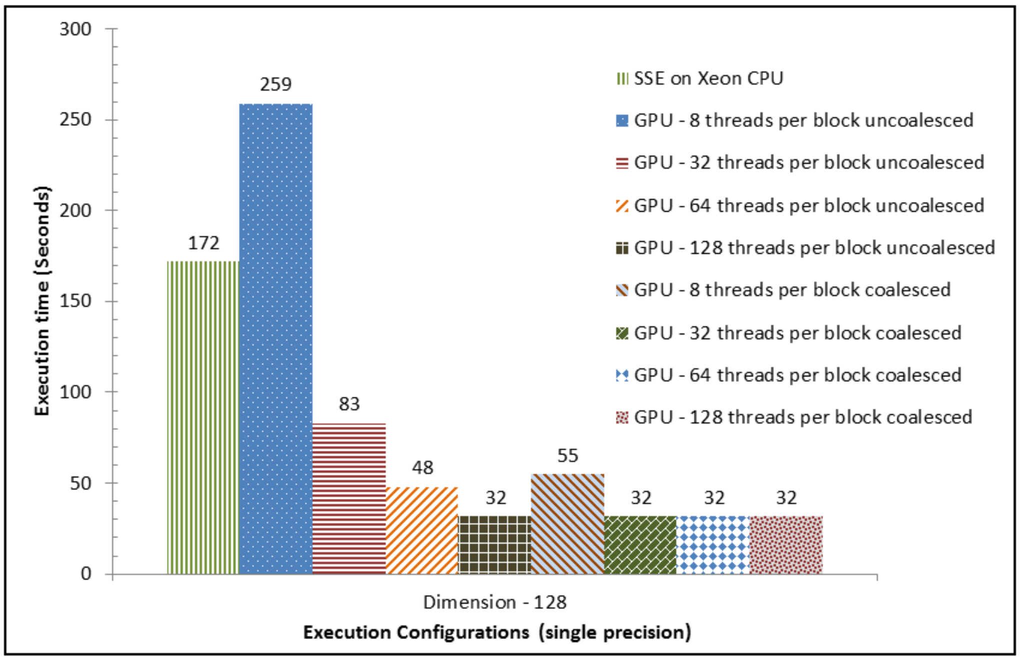


Figure 4.19: Single precision performance of GPU implementations at a problem size of 128

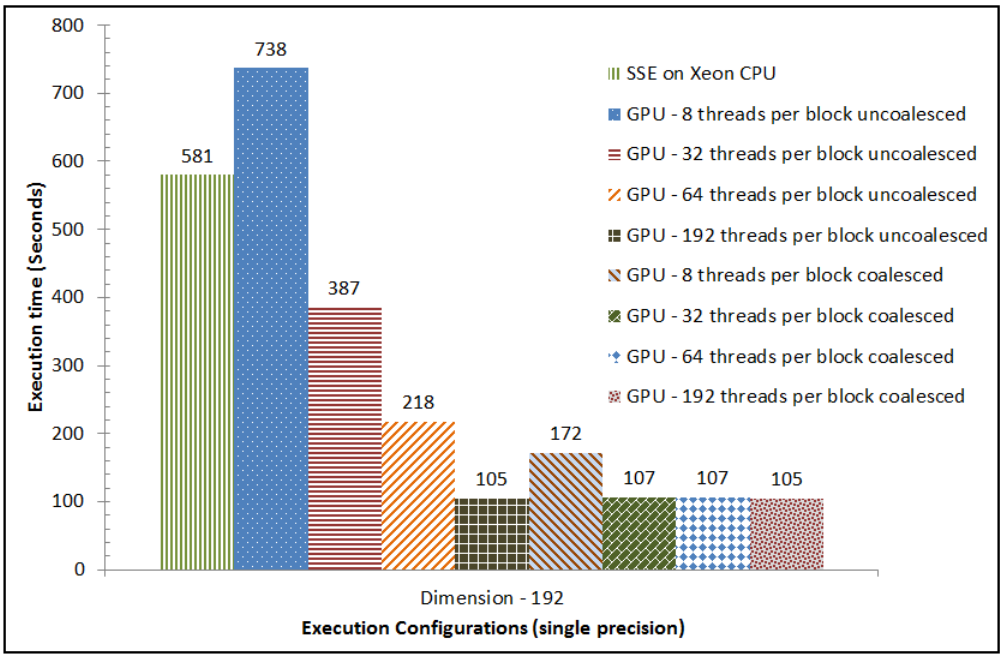


Figure 4.20: Single precision performance of GPU implementations at a problem size of 192

single-precision results differ from the double-precision results on the GPU in that the

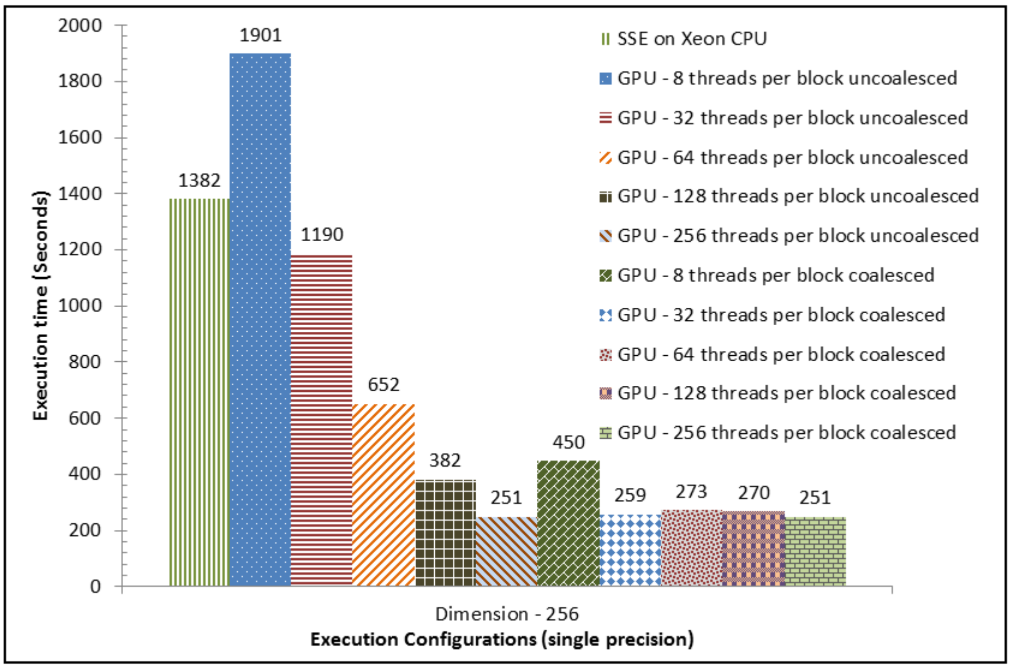


Figure 4.21: Single precision performance of GPU implementations at a problem size of 256

coalesced implementation with the highest thread count at each problem size is always the best performing, although jointly with the uncoalesced implementation at the highest thread count in each case. This differs from the double precision results where the best performing configuration differed slightly at different problem sizes. With the single precision implementation, high performance can be reliably achieved by setting the thread count to equal the problem size. This could be done all the way up to a problem size of 512, since 512 is the maximum number of threads per block allowed in CUDA [8].

The best performing single precision GPU implementation at each problem size is around 2 times faster than the best performing double precision GPU implementation. This is unexpected, since the double-precision performance of GPUs with compute capability 1.3 (those that precede the Fermi architecture) is expected to be around 8 times slower than single-precision performance (see Table 4.6). This expected difference in performance is because each streaming multiprocessor in the GPU contains 8 single-precision floating-point units but only 1 double-precision floating-point unit [34]. The scheduler for a warp of 32 threads (see Section 2.3) can issue a single-precision floating-point instruction to all 32 threads in 4 cycles, but requires 32 cycles to issue a double-precision floating-point instruction to all 32 threads [34]. Since this difference in performance is not observed in the results presented here, the implication is that arithmetic throughput is not the limiting factor for these GPU implementations. It is likely that memory bandwidth is the limiting factor, and therefore each streaming multiprocessor is not able to fully exploit the processing power of its cores. Since single-precision values are half the size of double-precision values, the overall global memory bandwidth required by the single-precision implementations is half that of the double-precision implementations. This correlates with the observed doubling in performance at single-precision and reinforces the conclusion that it is memory bandwidth which is the limiting factor. As described in Section 2.3, the ratio of arithmetic instructions to memory accesses is called the Compute to Global Memory Access (CGMA) ratio. To measure this ratio for these implementations, it is necessary to look at a typical line from the kernels, given in Figure 4.22.

```
dex[currentOffset] = dex[currentOffset] + dt/pmt * (
   ((dhz[offset(i,j+1,k,gridDim.x)] - dhz[currentOffset])/dy) -
   ((dhy[offset(i,j,k+1,gridDim.x)] - dhy[currentOffset])/dz)
) ;
```

Figure 4.22: Typical FDTD calculation from the CUDA kernels

The code in Figure 4.22 performs 5 global memory reads to access values from the matrices, and 1 global memory write to assign the result, making 6 global memory accesses in total. There are 8 floating point arithmetic instructions and two integer additions (*j+1* and *k+1*). There are also two calls to the *offset* function, shown in Figure 4.23.

```
__host__ __device__ int offset(int i,int j,int k)
{
      return i+ ( j * (DIM+2) ) + ( k * (DIM+2) *(DIM+2) );
}
```

Figure 4.23: The offset function

Assuming the compiler optimizes to only perform the *DIM+2* calculation once, each call to this function performs 6 arithmetic operations. This makes a total of 22 arithmetic operations (8 floating point and 2 integer along with two instances of the 6 operation offset function) for every 6 global memory accesses for this line of code, which is representative of the majority of work done by both kernels in the implementation. This gives a CGMA ratio of $22 \div 6$, approximately 3.7. According to [35], the maximum single-precision performance of a Tesla T10 GPU is 933 Gigaflops (one

Gigaflop is 2^{30} Floating Point Operations Per Second). The peak global memory bandwidth is 102 gigabytes per second, which equates to a maximum memory throughput of $25.5 * 2^{30}$ single-precision floating-point values per second. The arithmetic throughput is 37 times greater than the memory bandwidth (calculated as $933 \div 25.5$) and therefore in order to achieve the maximum possible single-precision performance, the CGMA ratio would need to be 37 or greater.

According to [8], increasing the CGMA ratio is typically done by introducing either constant memory or shared memory access. As described in Section 2.3, constant memory allows frequently accessed values which do not change during program execution to be broadcast to multiple threads. However in this implementation the program constants, such as *dt*, *pma* and *pmt* are already held in each thread's local registers, the fastest type of storage available. The constants are not held in global memory and therefore introducing constant memory would not reduce the number of global memory accesses and would not be expected to improve performance.

Shared memory is intended to allow a thread to store values in fast local storage which is accessible to the other threads in the same block. If more than one thread uses the same value, having one thread store the value in shared memory and having the other threads read it from shared memory rather than global memory reduces the total number of global memory accesses.

```
dez[currentOffset] = dez[currentOffset] + dt/pmt * (
  ((dhy[currentOffset+1] - dhy[currentOffset])/dx) -
  ((dhx[offset(i,j+1,k,gridDim.x)] - dhx[currentOffset])/dy)
);
```

Figure 4.24: Calculation of E_z in the CUDA kernel

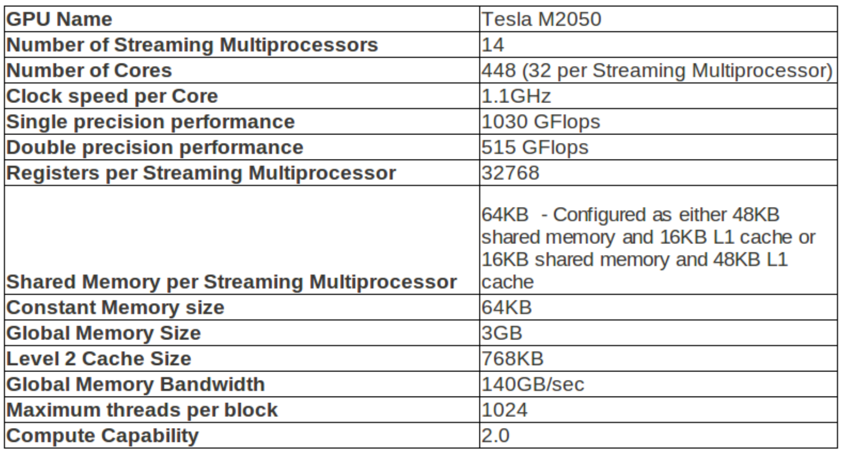
An example of this can be observed in the calculation for a value of the matrix E_z given in Figure 4.24. Consider the thread executing this instruction as T0. T0 accesses both *dhy[currentOffset]* and *dhy[currentOffset+1]* which are adjacent in memory. When using coalesced memory access, the thread adjacent to T0 (call it T1) will have a value of *currentOffset* which is greater by 1. Therefore when T0 accesses *dhy[currentOffset+1]* and T1 accesses *dhy[currentOffset]*, they are accessing the same memory location. If T0 stores the value in shared memory, T1 can access it from shared memory, reducing the number of global memory accesses. Four out of six of the matrix element calculation statements include a single memory access fitting this pattern (the calculations for E_y, E_z, H_y and H_z). This means an average reduction per statement of two thirds of an access for threads which benefit. However, only half of the threads do benefit, since one out of every two threads would be responsible for accessing the data from main memory and storing it in shared memory. Therefore exploiting this technique would reduce the average number of global memory accesses for a single statement by one third of one access, from 6 to 5.7. This increases the CGMA ratio to 3.9. The logic of the kernel would become significantly more complex for only a slight improvement in the CGMA ratio. The cost of synchronising between threads to enable coordination of sharing values in this way would most likely cancel out any benefit from increasing the CGMA ratio and would probably lead to an overall drop in performance.

A better approach may be to exploit the fact that when calculating E_x, E_y and E_z, values from H_x, H_y and H_z are used multiple times and vice-versa. For example, in Figures 4.22 and 4.24, the calculations of E_x and E_z both rely on *dhy[currentOffset]*. In the existing algorithm, this value would be loaded from global memory twice by the same thread. As an alternative, this value could be stored locally in a register variable and then reused as required. There are three such paired accesses across the three statements that make up one iteration within a kernel so this would reduce the number of global memory accesses on average by one per statement, from 6 to 5. This increases the CGMA ratio to 4.4, a greater increase than the shared memory technique, but still not enough to fully maximise the arithmetic throughput of the GPU.

The need to increase the the CGMA ratio is due to the lack of cache between global memory and the processing units in a CUDA capable GPU with compute capability 1.3. GPUs with compute capability 2.0 (commonly known as Fermi GPUs) do include cache between the processing units and the global memory. Section 4.4 demonstrates the use of a Fermi GPU to execute the GPU based FDTD implementations.

4.4 Experiments with a Fermi GPU

In order to test the performance of the CUDA FDTD implementations on Fermi hardware, a GPU Cluster instance was leased from Amazon's Elastic Cloud Computing (EC2) service. EC2 provides virtual machine instances of various configurations that are paid for by the hour. Performing these experiments required nine hours of computing time on a single GPU Cluster instance resulting in a total cost of £22.72. Amazon EC2 offers the GPU Cluster instance type containing two NVIDIA Fermi M2050 graphics processors. The specification of an M2050 processor is shown in Table 4.7. The information in this table was obtained executing the "enum_gpu" CUDA program



GPU Name	Tesla M2050
Number of Streaming Multiprocessors	14
Number of Cores	448 (32 per Streaming Multiprocessor)
Clock speed per Core	1.1GHz
Single precision performance	1030 GFlops
Double precision performance	515 GFlops
Registers per Streaming Multiprocessor	32768
Shared Memory per Streaming Multiprocessor	64KB - Configured as either 48KB shared memory and 16KB L1 cache or 16KB shared memory and 48KB L1 cache
Constant Memory size	64KB
Global Memory Size	3GB
Level 2 Cache Size	768KB
Global Memory Bandwidth	140GB/sec
Maximum threads per block	1024
Compute Capability	2.0

Table 4.7: Specification of a Fermi M2050 GPU

from chapter 3 of [20] and from the information published by NVIDIA in [22] and [36].

The GPU Cluster instances available on the EC2 service have 16 Intel Xeon processor cores and 22GB memory, making them powerful machines before taking account of their graphics processing capability. However for this exercise the compute intensive parts of the GPU implementations execute on the GPU hardware only and so the overall CPU processing power and main memory of the machine have no bearing on the performance results.

Comparing Table 4.7 to Table 4.6, there are several noticeable differences between the Fermi based Tesla M2050 GPU and the Tesla T10 GPU used in Section 4.1. Although the Fermi GPU has significantly fewer Streaming Multiprocessors (14 compared to 30), the Fermi architecture specifies 32 cores per Streaming Multiprocessor rather than 8, giving an overall count of 448 cores in the M2050 GPU, compared to 240 cores in the Tesla T10 GPU. The Fermi GPU also has slightly lower clock speed. The net result of the different organisation, quantity and speed of the cores is that the Fermi GPU has a maximum performance of 1030 GFlops (Giga Floating Point Operations Per Second) for single-precision arithmetic, only slightly greater than the maximum performance of 933 GFlops in the Tesla T10. The difference for double-precision arithmetic is much greater, with the Fermi GPU giving 515 GFlops compared to 78 GFlops in the Tesla T10 GPU. Note that while NVIDIA lists these performance figures in their documentation in [33] and [36], they do not explain exactly how the GFlops figure is derived from the number of cores and the scheduling of threads to them. The Fermi GPU has twice as many registers per Streaming Multiprocessor, but has four times as many cores per Streaming Multiprocessor, meaning that there are as half as many registers available on a per core basis compared to the Tesla T10 GPU. This may adversely affect programs which rely on a large number of registers per thread since a Streaming Multiprocessor will only support as many threads as it can provide registers for, potentially limiting the number of threads running on a Streaming Multiprocessor at run-time [8].

The Fermi GPU has twice as much shared memory for each Streaming Multiprocessor and this shared memory can be configured so that either 16KB or 48KB are used as an level 1 cache, with the remainder operating as user programmable shared memory [22]. In the Tesla T10, shared memory is only user programmable, there is no option to use it as level 1 cache. The Fermi GPU also includes an level 2 cache unified across all streaming multiprocessors which does not exist at all in the Tesla T10. Additionally, the Fermi GPU has a global memory bandwidth of 140GB per second, greater than the 102GB per second on the Tesla T10. The combination of two levels of cache and additional memory bandwidth means that the Fermi GPU is better able to satisfy the memory requests of the threads in a CUDA kernel in a timely manner to allow the processor cores to remain busy for a greater portion of the time.

The experiments executed on the Fermi GPU used the same configuration as previous experiments, with four problem sizes of 64, 128, 192 and 256 with 5000 time-step iterations. Since access to EC2 virtual machines has an associated cost, only 3 executions were performed for each implementation at each problem size, with the average of these 3 executions being used. While this is less than the 10 executions used in Section 4.3, the timing results of the 3 executions were very uniform in each instance, so confidence in the validity of these timing results remains high. Similarly, due to the cost associated with executing on the Fermi GPU, the lower thread counts of 1 and 8 threads per block used in Section 4.3 were not executed in this case. The results in Figures 4.14 to 4.17 and 4.18 to 4.21 show that these thread counts do not offer good performance compared to higher thread counts on the Tesla T10 gpu so it was decided it was not worth executing them on the Fermi GPU as this trend was likely to be repeated. The GPU implementations in this project do not use the programmable shared memory, so the level 1 cache of each Streaming Multiprocessor was set to its maximum 48KB. This configuration is the default option on Fermi GPUs so did not require any special parameters during compilation.

Figures 4.25 to 4.28 show the double-precision performance of various thread counts per block for both uncoalesced and coalesced memory access when executed on the Fermi GPU. The best performing CPU based implementation from Section 4.1

and the best performing Tesla T10 GPU based implementation from Section 4.3 are included at each problem size to allow for comparison.

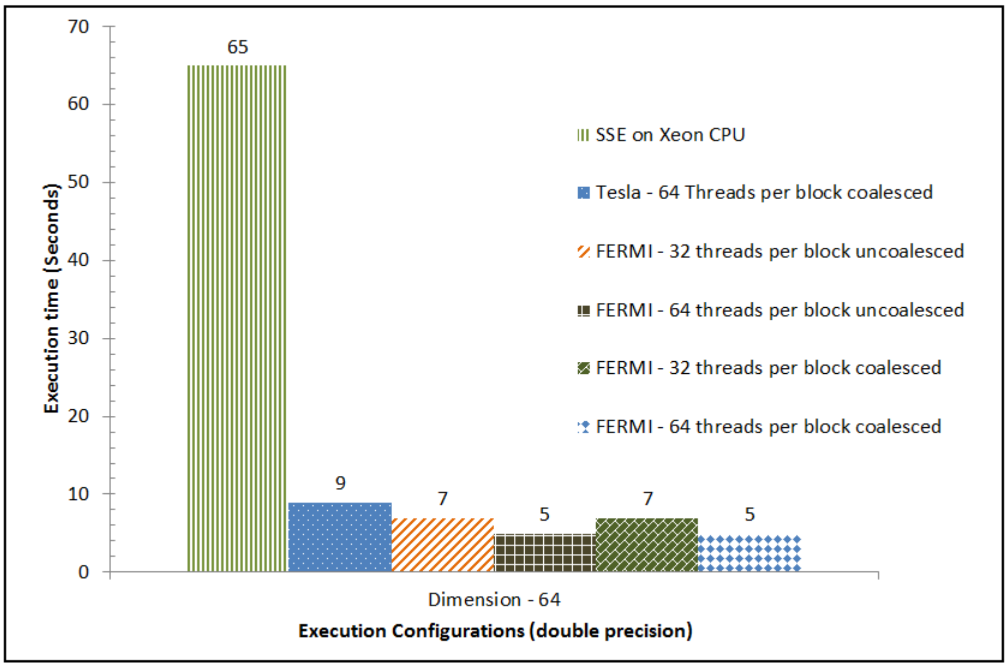


Figure 4.25: Double precision performance of GPU implementations on the Fermi GPU at a problem size of 64

At a problem size of 64, the best performing double-precision implementations on the Fermi GPU are those using 64 threads per block, for both uncoalesced memory and coalesced memory (as explained in Section 4.3, when the thread count is equal to problem size there is essentially no difference between the uncoalesced and coalesced memory option in terms of the execution at run-time). This result is nearly twice as fast than the best performing GPU implementation on the Tesla T10 GPU, and 13 times faster than the best performing CPU implementation.

At a problem size of 128, the best performing double-precision implementation on the Fermi GPU is again at 64 threads per block, but this time exclusively for coalesced memory access. This is more than twice as fast as the best performing GPU implementation on the Tesla T10 GPU and 18 times faster than the best CPU based implementation.

Similar results are seen at a problem size of 192, with 64 threads per block and coalesced memory access on the Fermi GPU performing more than twice as fast as the best GPU implementation on the Tesla T10 GPU and 19 times faster than the best CPU based implementation.

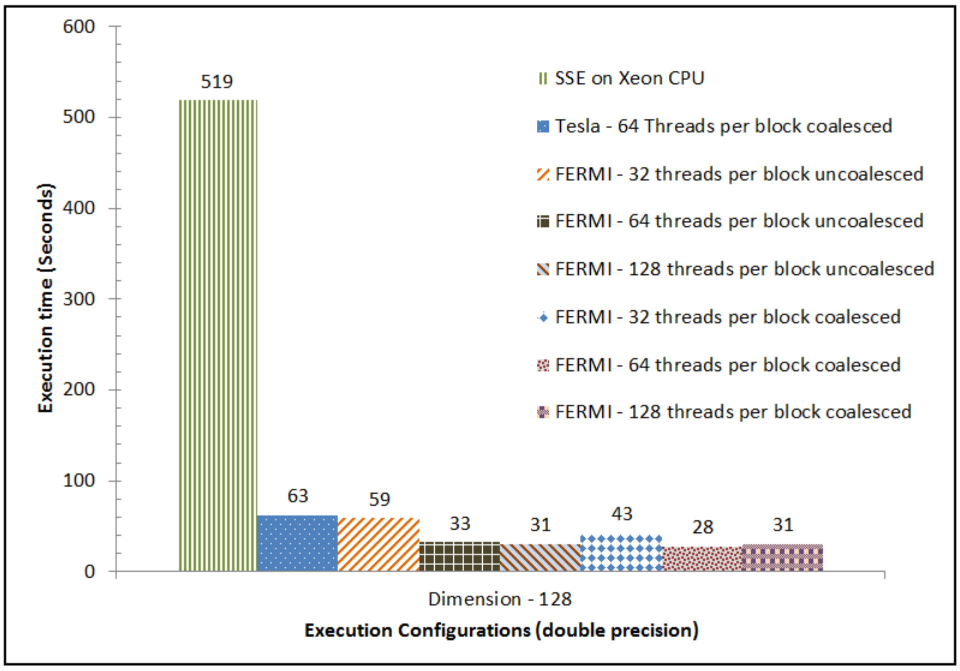


Figure 4.26: Double precision performance of GPU implementations on the Fermi GPU at a problem size of 128

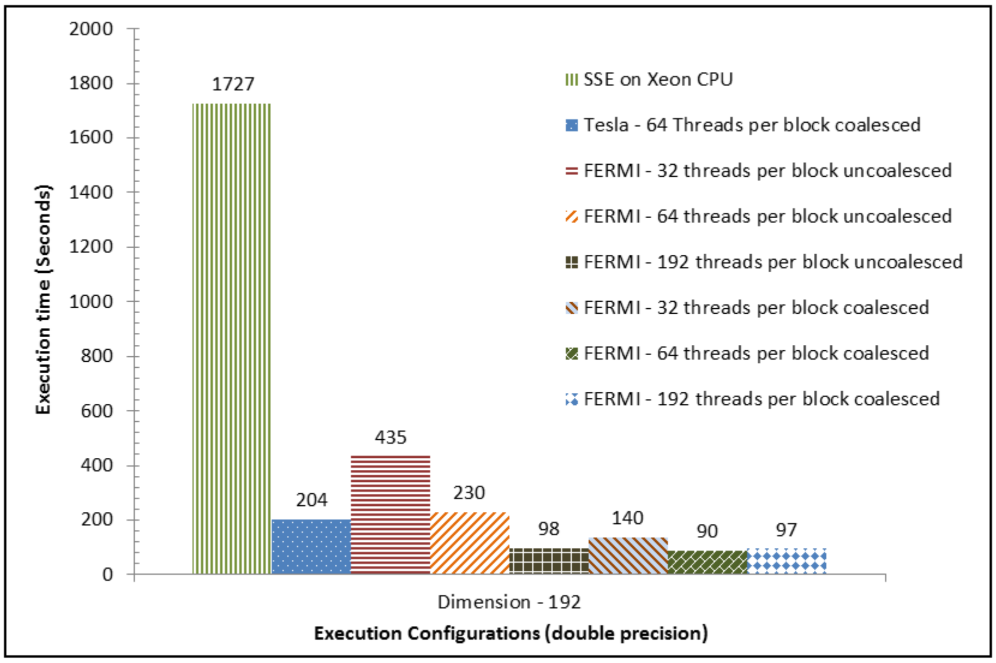


Figure 4.27: Double precision performance of GPU implementations on the Fermi GPU at a problem size of 192

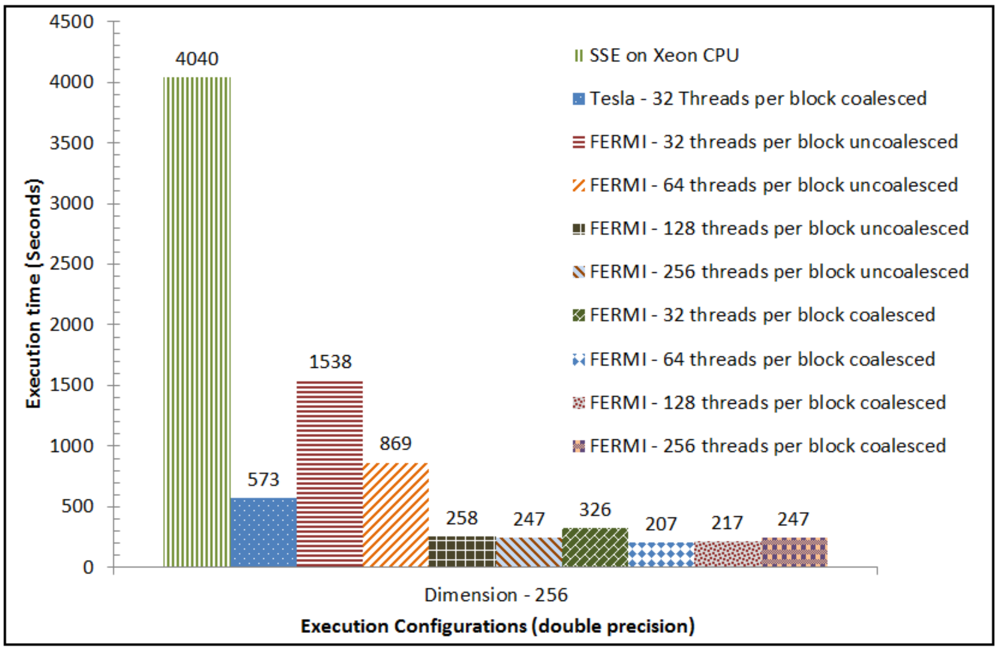


Figure 4.28: Double precision performance of GPU implementations on the Fermi GPU at a problem size of 256

At the largest problem size of 256, the best implementation on the Fermi GPU is again the 64 threads per block with coalesced memory configuration, and it performs even better compared to the Tesla T10 GPU at over 2.5 times faster, while being 19.5 times faster than the best CPU based implementation.

In general it can be seen that the Fermi GPU offers much greater performance than the Tesla T10 GPU. In addition, the performance difference increases as the problem size is increased, indicating that the CUDA implementation of the FDTD method scales better on the Fermi GPU than the Tesla T10 GPU. This is likely due to the fact that as the problem size increases, so the number of blocks created for each kernel increases and therefore the overall number of threads launched increases. This is due to the design decision from Section 3.4 that the number of blocks is set to the square of the problem dimension. Since the Fermi GPU has a greater number of cores, it can run more threads in parallel than the Tesla T10 GPU, and needs a larger number of threads to achieve its maximum performance.

There are many features of compute capability 2.0 (present in the Fermi M2050 GPU) which would allow for this improvement over compute capability 1.3 (present in the Tesla T10 GPU). Compute capability 2.0 offers much greater double-precision performance, has greater memory bandwidth, and has L1 and L2 cache to reduce the number of requests made to global memory. It is not possible to isolate directly how each feature contributes to the increase in performance. However if the performance increase was entirely due to the greater double-precision performance and not due to the global memory bandwidth and cache features, then it would be expected that the performance increase would not be repeated for the single-precision implementations since there is not a large difference in the maximum single-precision performance of the Fermi GPU and the Tesla T10 GPU (1030 GFlops compared to 933 GFlops). The same experiments were repeated for single-precision, with the results shown in Figures 4.29 to 4.32.

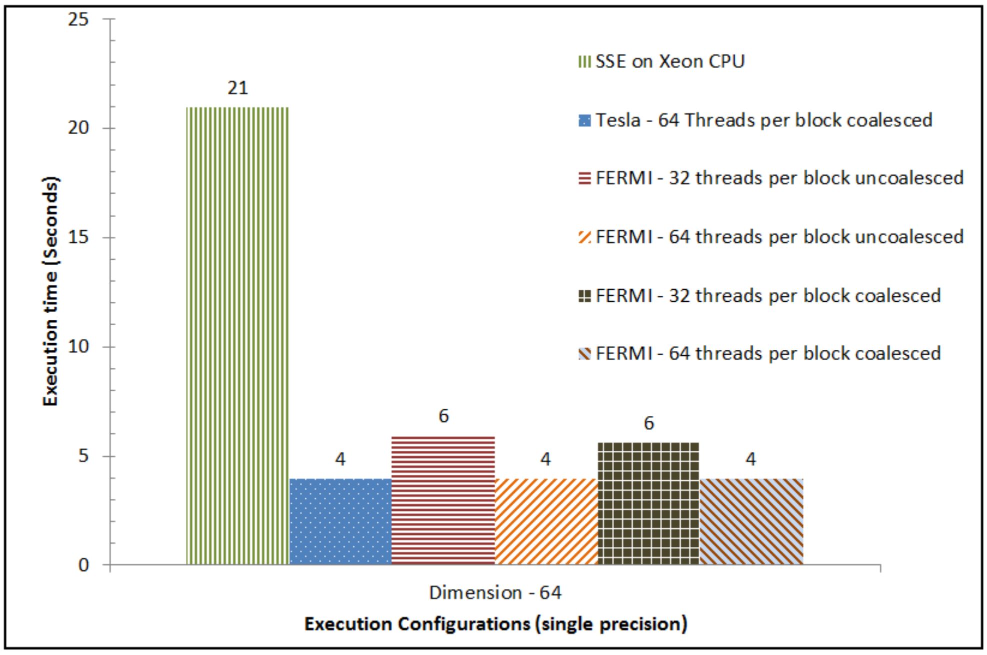


Figure 4.29: Single precision performance of GPU implementations on the Fermi GPU at a problem size of 64

At a problem size of 64 using single-precision arithmetic, the uncoalesced and coalesced 64 threads per block implementations give the best performance. However this performance is equal to that seen on the Tesla T10 GPU and only 5 times faster than the CPU based implementation.

At a problem size of 128, it is coalesced memory access with 128 threads per block which performs best on the Fermi GPU, which is 1.5 times faster than the best Tesla T10 implementation and 8 times faster than the best CPU implementation.

At a problem size of 192, the coalesced 64 threads per block implementation gives

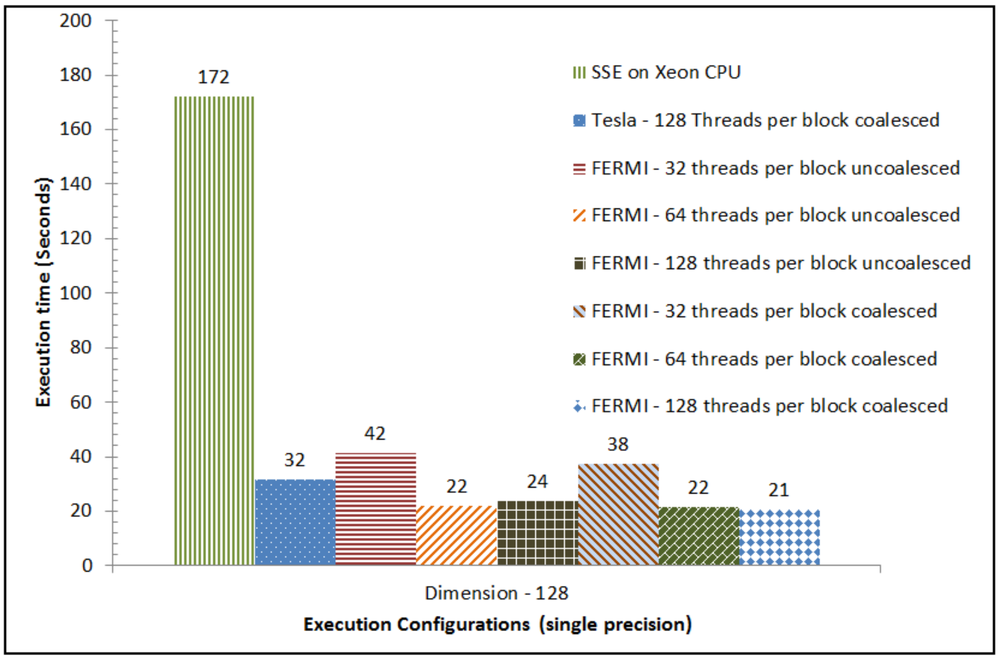


Figure 4.30: Single precision performance of GPU implementations on the Fermi GPU at a problem size of 128

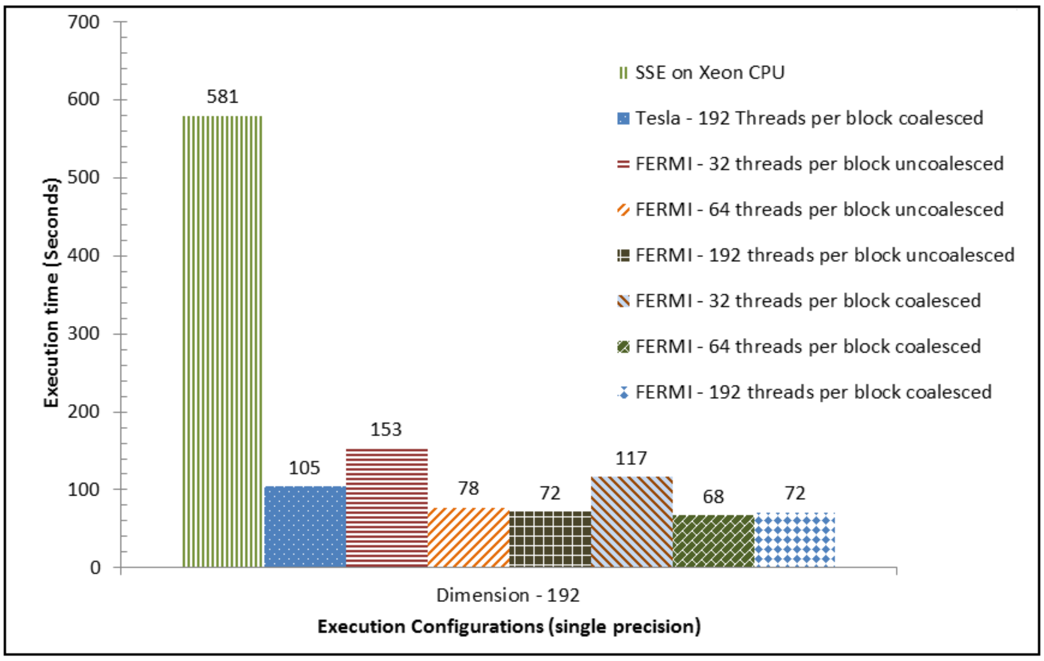


Figure 4.31: Single precision performance of GPU implementations on the Fermi GPU at a problem size of 192

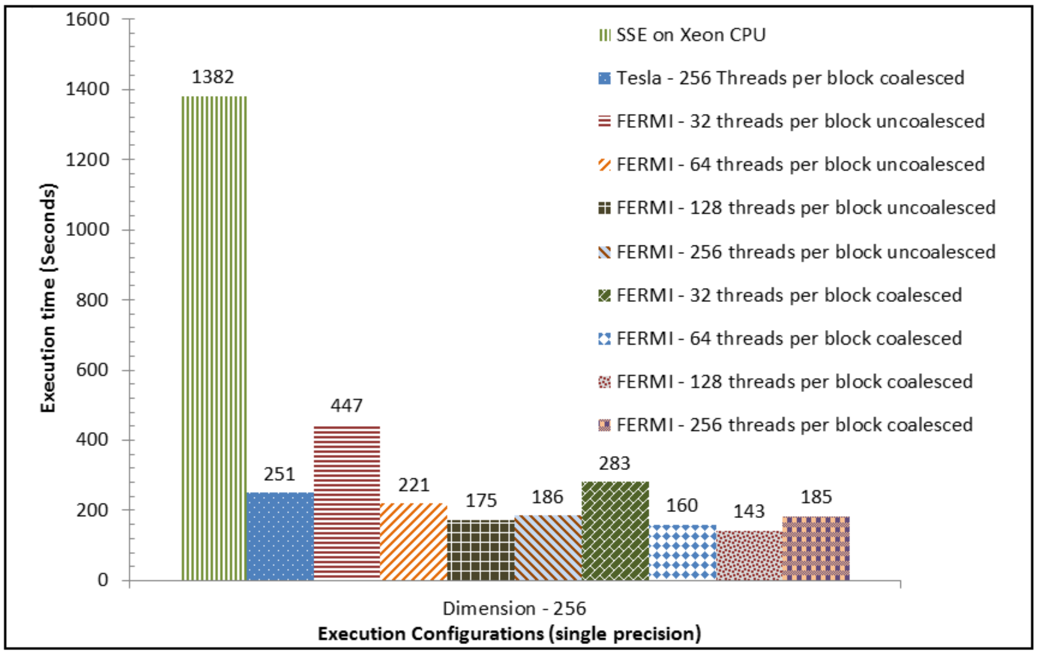


Figure 4.32: Single precision performance of GPU implementations on the Fermi GPU at a problem size of 256

the best performance and the performance difference to the best Tesla T10 implementation is again 1.5, while being 8.5 times faster than the best CPU implementation.

At a problem size of 256, coalesced memory access with 128 threads per block performs best on the Fermi GPU. This is 1.75 times faster than the best Tesla T10 GPU implementation and nearly 10 times faster than the best CPU based implementation.

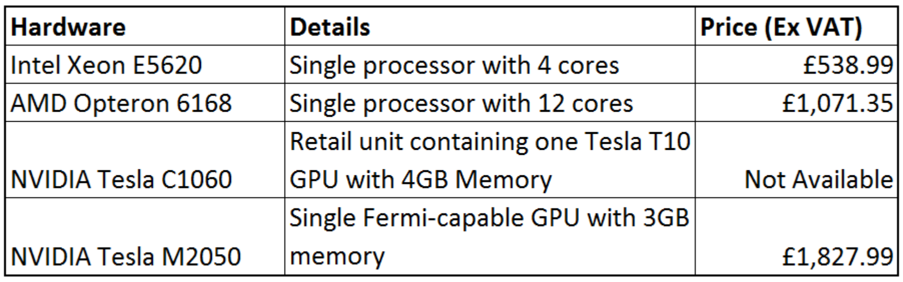
The single-precision performance difference between the Fermi GPU and both the Tesla T10 GPU and the CPU increases gradually as the problem size increases, repeating the observation seen with double-precision. However, across the problem sizes, the difference in performance is much lower for the single-precision implementations than double-precision. This suggests that it is the increased double-precision performance of the Fermi GPU which accounts for much of the performance difference seen for double-precision arithmetic. However, there is some performance difference with single-precision, so this indicates that the memory bandwidth and cache enhancements also provide some performance improvement to the implementations in this project.

Overall, the Fermi GPU with compute capability 2.0 provides performance improvements over the Tesla T10 GPU with compute capability 1.3 when executing the CUDA based FDTD implementations developed in this project. This is particularly the case when using double-precision arithmetic. Compared to the best CPU based implementation, the Fermi GPU offers up to a 20 times performance improvement when using double-precision, with this improvement increasing as the problem size increases. It would require a parallel implementation run across at least 20 cores (depending on efficiency) to match this performance using CPU hardware, even when incorporating the SSE streaming techniques from Section 3.2. A single Fermi GPU device provides a level of performance which would require significant amounts of parallel CPU hardware to match. This suggests that a Fermi GPU is a good choice for executing the FDTD method. It should be noted however that the size of the global memory limits the maximum problem size to which the GPU implementations in this project can scale. The total memory usage of an implementation is measured as the dimensions of the problem (plus padding) cubed, multiplied by the size of a floating point value (8 bytes for double-precision), multiplied by 6 for the number of matrices used by the algorithm. This means that the 3GB global memory of the M2050 Fermi GPU used here can support a maximum problem size 400 for double-precision, since $402^3 \times 8 \times 6$ gives 3,118,310,784 bytes, slightly less than the 3GB limit. This limit could be worked around by moving the matrices on to the GPU in segments for processing. However the cumulative effect of moving the necessary segments around would mean copying the whole of the matrices E_x, E_y, E_z, H_x, H_y and H_z on to the GPU and back to main memory for each time step, introducing a significant overhead to the algorithm.

As shown in [36], there are other Fermi GPUs such as the Tesla M2090 available with up to 6GB of global memory, allowing a double-precision problem size of up to 510 to be used ($512^3 \times 8 \times 6$ is exactly 6GB). In addition, the Tesla M2090 GPU has an increased global memory bandwidth of 177 GB per second and an increased peak double-precision performance of 665 GFlops. This GPU would therefore likely offer greater performance improvements as well as supporting larger problem sizes.

4.5 Performance to price comparison

Table 4.8 gives the cost of some of the processing hardware used in this project to execute the various FDTD implementations. Prices of computing hardware can be volatile over time and from different retailers, but the prices here were all taken from a single retailer, *uk.insight.com*, on the same day, so represent a relatively fair comparison. In each case, the price includes the processing unit only and not an entire machine. The CPU hardware would require a compatible motherboard and other components to make a working system, and similarly the GPUs would require a motherboard with a PCI express slot available and other components to form a complete system. For the purposes of this comparison the assumption is being made that the cost of the additional hardware would be equal for each processing unit and therefore comparing the cost of the processing hardware alone is sufficient.



Hardware	**Details**	**Price (Ex VAT)**
Intel Xeon E5620	Single processor with 4 cores	£538.99
AMD Opteron 6168	Single processor with 12 cores	£1,071.35
NVIDIA Tesla C1060	Retail unit containing one Tesla T10 GPU with 4GB Memory	Not Available
NVIDIA Tesla M2050	Single Fermi-capable GPU with 3GB memory	£1,827.99

Table 4.8: Cost comparison of various processing hardware

The Tesla T10 GPU used for the experiments in 4.3 is sold as the internal processing unit of packaged units with different names. One of these, the Tesla C1060, is still listed at the retailer being used for this comparison but is no longer available for sale. Where the Tesla C1060 can be found at other retailers, it is similarly priced to the Fermi GPU. In general the latest generation of Fermi capable GPUs are similarly priced to the previous generation Tesla GPUs and as described in Section 4.4, the Fermi GPU offered better performance for the experiments in this project. The implication of this is that it is clearly more cost effective from a price to performance perspective to acquire a Fermi GPU rather than an older Tesla GPU. For this reason the Tesla T10 GPU is considered obsolete and will not be considered further in this comparison.

Performance comparisons are made using the results at a problem size of 256 (since larger problem sizes are more representative of real-world FDTD applications) and using double-precision (since double-precision results are available for all processing units and as described in Section 3.5 double precision is considered necessary for more complex FDTD applications). At this problem size, the AMD Opteron executes the SSE streaming implementation in 4827.5 seconds. The Intel Xeon executes this implementation in 4030 seconds. Since the Intel Xeon is approximately half the cost of the AMD Opteron and provides better performance, it clearly offers superior price to performance for the implementations in this project. The best configuration on the Fermi M2050 GPU executes in 207 seconds at a problem size of 256 for double-precision. This is a speed up of 19.5 compared to the Intel Xeon, while costing only 3.4 times more.

By measuring throughput as the number of elements in the problem size ($256^3 = 16,777,216$) divided by execution time, this gives a measure of the number of elements executed by a processor for each second of execution time. Dividing this throughput metric by the cost of a processor, the result gives the number of elements processed per second per pound spent. This can be used as a price-performance metric allowing objective comparison between the different processing units. Table 4.9 shows this calculation for each processing unit for which cost information is available and clearly shows that the Fermi GPU provides much better value in terms of performance per cost of hardware.

Hardware	Best execution time at a problem size of 256	Throughput (256^3 divided by execution time)	Throughput per pound spent
Intel Xeon E5620	4030	4163.08	7.72
AMD Opteron 6168	4827.5	3475.34	3.24
NVIDIA Tesla M2050	207	81049.35	44.34

Table 4.9: Throughput per cost of hardware

However, the CPU implementations in this project do not attempt to make use of the multiple cores available on the CPUs, whereas the GPU implementations attempt to make use of every single core available on the GPUs. Therefore the comparison in Table 4.9 is not representative of the true potential performance per cost of hardware. As described in Section 2.1, [6] demonstrates that a parallel multi-processor implementation of the FDTD method can achieve greater than 80% efficiency. Assuming an 80% figure for efficiency and assuming usage of all the cores on the CPUs in Table 4.8, it is possible to reason about the potential performance that may be achieved if the SSE streaming implementation was extended to execute on multiple cores. These updated calculations are given in Table 4.10.

Hardware	Best execution time at a problem size of 256	Number of CPU cores	Potential speedup assuming 80% efficiency	Throughput (256^3 divided by execution time)	Throughput per pound spent
Intel Xeon E5620	4030	4	3.2	13321.86	24.72
AMD Opteron 6168	4827.5	12	9.6	33363.29	31.14
NVIDIA Tesla M2050	207	-	-	81049.35	44.34

Table 4.10: Potential throughput per cost if all cores were utilised

When the potential performance offered by multiple cores is taken into account, the AMD Opteron processor now offers greater value than the Intel Xeon processor.

The Fermi GPU still offers superior value to the two CPUs, but the difference is significantly reduced. This is of course based on conjecture regarding the scalability of the CPU based SSE streaming implementation of the FDTD method, but shows that a GPU based system is likely to offer superior value than a CPU based system in terms of performance per cost of hardware when all cores are fully utilised on each device.

Chapter 5

Conclusions and Further Work

The FDTD method contains inherent opportunities for parallelism due to the large number of independent calculations required at each time-step. Using a simple sequential Fortran implementation as a starting point, this project has explored the application of SIMD hardware in the form of SSE instructions on x86 architecture processors and general purpose computing on GPU hardware using NVIDIA's CUDA technology to accelerate the execution time of the FDTD method.

The introduction of SIMD hardware to the x86 architecture in the form of SSE instructions means that processors based on this architecture can execute the same operation concurrently on multiple data items. In this project, SSE instructions were used with the aim of doubling the throughput of the FDTD method when using double-precision arithmetic. Each calculation for each matrix in the FDTD algorithm was applied to two adjacent elements simultaneously, exploiting the ability of each 16 byte SSE register to store and operate on two double-precision values at once. Two SSE implementations were developed, one using hand written assembly language sequences and one using intrinsic functions which give the programmer access to SSE streaming at a higher level. It was found that the hand written assembly language implementation was not portable between different processors (even though each processor supported the SSE2 instruction set), and that it offered no significant performance improvement over the intrinsic function implementation. It is concluded therefore that intrinsic functions are the better approach for programming with SSE streaming, since they offer portability, greater ease of implementation, and do not incur any significant performance penalty.

When executed on 4 different processors supporting the SSE2 instruction set, the performance improvement offered by the SSE implementation differed noticeably on each processor. On a 32-bit Intel Core 2 Duo, the speedup offered by the SSE streaming implementation for the critical portion of the algorithm was between 1.6 and 1.75 compared to the best sequential implementation. The 64-bit Intel Xeon machine and 64-bit AMD Opteron machine offered a speedup of 2 for the critical portion of the algorithm, the ideal speedup expected when doubling the throughput. In contrast, a 64-bit AMD Athlon machine offered no significant speedup compared to the sequential implementation. The investigation into this difference in performance concluded that the most likely cause was that the SSE implementation on the AMD Athlon processor does not execute a packed SSE instruction on two sets of operands any faster than it performs separate instructions for each set of operands. That is, while the AMD Athlon processor is able to execute packed SSE instructions, it does not offer any additional throughput when doing so. This conclusion could be investigated further by executing other algorithms based on the SSE2 instruction set on the four architectures used in this Section 4.1. If the same pattern of performance is observed for other algorithms, this would support the conclusion that the AMD Athlon processor does not provide improved throughput for SSE instructions. If however other SSE2 based algorithms achieved improved performance on the AMD Athlon processor, this would suggest that there is a problem with the way SSE has been used in the implementation of this project, and the implementation could be revisited to see if it can be tuned to provide additional performance on the AMD Athlon processor without impacting the existing performance improvements observed on the other processors. The results observed in this project suggest that any implementation using SSE instructions should be executed on multiple architectures in order to determine whether the expected performance improvements are limited to particular hardware.

Both AMD processors exhibited significant dips in performance at particular problem sizes. This was investigated and ultimately attributed to a problem of the matrices used in the algorithm aligning in memory in such a way that severe cache contention occurs causing an increase in cache misses. It was shown that while programming techniques can be used to detect when alignment will occur and work around it by assigning redundant memory to force misalignment, these techniques are complex and specific to particular processors. It was concluded that while an awareness of those problem sizes causing cache contention is important to avoid drops in performance, it is more practical to simply adjust the problem size slightly rather than pursue complex, processor-specific workarounds.

Overall the results from the SSE implementation show that a Fortran based FDTD implementation can be extended with SSE streaming instructions by replacing the performance critical sections with calls to optimised procedures written in C. This approach can double performance for implementations using double-precision arithmetic, but these improvements are processor-specific and the performance of SSE streaming instructions varies greatly from processor to processor.

It is common to accelerate the FDTD method on hardware with multiple processor cores using Single Program Multiple Data (SPMD) techniques to divide the matrices into large sections and allocating each section to separate thread or process running on a different core. This is a coarse-grained division of the problem space, whereas the SIMD techniques presented in this project using SSE instructions take a fine-grained approach of processing two adjacent elements in parallel. The techniques in this project are considered to be compatible with the SPMD approach, in circumstances where each core of the multi-processor machine is capable of executing SIMD instructions. Any of the processors used in this project and detailed in Table 4.1 fit the criteria of being both multi-core and capable of executing SSE instructions. Indeed, any modern x86 architecture processor is likely to support both multi-core and SSE. A useful follow up to this project would be to apply the SIMD techniques presented to an existing parallel FDTD implementation. It is expected that the coarse-grained division of the matrices among multiple cores would not interfere with the fine-grained streaming of multiple calculations at once using SIMD on each core, and that the performance improvements observed should be similar to those seen here. That is, depending on the SSE performance of the processor, applying SSE instructions to an existing SPMD, parallel, double-precision FDTD implementation may double its performance.

The increasing programmability of GPU hardware has led to increased interest in executing scientific computations on the GPU. This project has explored the suitability of GPUs based on NVIDIA's Tesla architecture for accelerating the FDTD method. Using CUDA, the programming language created by NVIDIA for developing implementations on Tesla based GPUs, this project re-implemented the same FDTD algorithm used in the SSE experiments. It was found that in order to exploit parallelism on a GPU, sections of the program containing a large number of independent calculations which can be executed in parallel need to be identified. These sections were implemented as CUDA kernels, which are passed to the GPU for processing at run-time. In order to achieve high performance when executing a kernel it was necessary to exploit both block-level and thread-level parallelism within the implementation. The implementation which split kernels into parallel blocks but had each block executing only a single thread failed to match the performance of the CPU based implementations.

When executed on Tesla hardware with compute capability 1.3, the best double-precision GPU implementation performed between 7 and 8.5 times faster than the best CPU implementation at each problem size. The memory coalescing technique described in the literature as a key performance improvement was found to improve performance significantly when the number of threads per block was low, but that the difference between coalesced and uncoalesced memory reduced as the number of threads per block was increased. When using single-precision arithmetic, it was found that the GPU performed around 2.5 times faster than when using double-precision arithmetic. This was less than the expected difference, with the literature stating that GPUs of this architecture perform 8 times faster at single-precision rather than double precision. This difference is most likely due to the implementation being memory bound rather than compute bound. As stated in the literature, the performance of GPU implementations is highly dependent on the ratio of computing operations to memory request operations. Tesla architecture GPUs have no cache between the processing units and the global memory so are much more sensitive than CPUs to global memory bandwidth. Although GPUs typically have higher memory bandwidth than CPUs, this does not compensate for the lack of cache. The likely cause of the smaller than expected differential in performance between double-precision and single-precision arithmetic on the Tesla based GPU is that memory access time is the dominant factor in limiting performance and therefore the high speed single-precision abilities are being throttled waiting for memory requests to be satisfied. The FDTD method implemented here does not have a high enough Compute to Global Memory Access (CGMA) ratio to allow for maximum performance from the GPU. The CUDA implementations in this project do not attempt to make use of shared memory or constant memory to improve the CGMA ratio. As described in Section 4.3, it is not anticipated that these techniques would significantly improve performance in this case. There may be some performance improvement to be gained from holding matrix elements in local register variables where they are used more than once by the same thread. Extending the GPU implementation by exploring these techniques to improve the performance of the FDTD method when executed on a GPU would be useful further work.

More recent NVIDIA GPUs implement the Fermi architecture, also known as compute capability 2.0. This compute capability features improved double precision support and two levels of cache between the processing units and the global memory. The GPU experiments were repeated using a Fermi GPU on a machine rented from

Amazon's cloud computing infrastructure. It was found that the double-precision implementations executed significantly quicker on the Fermi GPU. The performance improvement increased as the problem size grew, with a performance improvement of 2.5 times at a problem size of 256 relative to the Tesla GPU. In addition, the performance on the Fermi GPU at this problem size was around 20 times faster than the performance of the best CPU implementation.

The performance improvement using single-precision arithmetic was not as significant using the Fermi hardware. At a problem size of 256, the Fermi GPU executed around 1.75 times faster than the Tesla GPU, and around 10 times faster than the best CPU implementation. These results indicated that the performance increase observed with double-precision arithmetic was a combination of improved double-precision performance, and improved memory performance due to increased global memory bandwidth and the presence of two levels of cache. The single-precision improvement was less since only the memory performance improvements apply in that case.

A major limitation of the GPU implementations is that the size of the global memory on the GPU hardware places an upper limit on the maximum problem size that can be executed. With the implementations presented here, the matrices used in the algorithm need to fit completely into the global memory of the GPU in order to execute correctly. Working around this by moving the matrices in and out of memory in segments would be likely to introduce significant overhead, reducing the performance.

Although the implementation of the FDTD method on the two generations of CUDA capable GPU hardware failed to achieve the theoretical maximum performance that the hardware was capable of, most likely due to problems of memory bandwidth, the GPU implementations still offered significant speedup compared to the CPU implementations. However, if the SSE streaming techniques were successfully combined with SPMD techniques as described above to spread the workload over multiple CPU cores, it is likely that, given enough processing cores, the resulting parallel CPU based implementation would exceed the performance on the GPU implementation. The GPU implementation could also be parallelised across multiple devices as an extension to this work. Both the University of Kyushu machine containing the Tesla architecture GPU and the rented Amazon cloud computing machine containing the Fermi architecture GPU contain 2 GPU devices. The matrices that are used in the algorithm of this project could be divided between 2 or more GPU devices, and a kernel could execute on each GPU simultaneously. Since each calculation for a particular element relies on elements from other matrices displaced in all three dimensions, it would be necessary to copy data from the boundaries of the division of the matrices in between each kernel invocation. Provided that the time to copy the boundary data did not exceed the time saved by using multiple GPUs to share the workload of the algorithm, the performance of GPU implementation could be increased in this way. In addition, spreading the execution across multiple GPUs would help with the problem of the global memory capacity of a single GPU limiting the maximum problem size. For example, if two GPUs were responsible for half the problem space each, the maximum problem size would be based on the combined memory capacity of both GPUs. Exploring a multiple GPU implementation of the FDTD method would be a natural next step to the implementation presented in this project.

The algorithm used in this project was intentionally selected as a simple case of the FDTD method in order to manage the scope of the project. Using a simple FDTD algorithm allowed the focus to be on the use of SSE and GPU hardware to accelerate the performance critical sections of the algorithm. The results from this project could be taken further by applying the techniques presented to more complex cases of the FDTD method. It would be useful to demonstrate whether the performance improvements observed when applying SSE streaming and GPGPU techniques to a simple implementation of the FDTD method are maintained when implementations of greater complexity are used.

Bibliography

[1] A. Taflove and S. C. Hagness. *Computational Electrodynamics: The Finite-Difference Time-Domain Method*. Artech House Publishers, New York, 3 edition, 2005.

[2] Kane Yee. Numerical solution of initial boundary value problems involving Maxwell's equations in isotropic media. *Antennas and Propagation, IEEE Transactions*, May 1966.

[3] F. Costen. *High Speed Computational Modeling in the Applicaton of UWB Signals*. PhD thesis, University of Manchester, 2005.

[4] Hasan Khaled Rouf. *Unconditionally stable finite difference time domain methods for frequency independent media*. PhD thesis, University of Manchester, 2010.

[5] Allen Taflove. *Advances in Computational Electrodynamics: The Finite-Difference Time-Domain Method (Artech House Antenna Library)*. Artech House Inc, 1998.

[6] C. Guiffaut and K. Mahdjoubi. A parallel FDTD algorithm using the MPI library. *Antennas and Propagation Magazine, IEEE*, 43(2):94 –103, April 2001.

[7] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Third Edition: The Hardware/Software Interface, Third Edition (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann, 2004.

[8] David B. Kirk and Wen mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach (Applications of GPU Computing Series)*. Morgan Kaufmann, 2010.

[9] S. Thakkur and T. Huff. Internet Streaming SIMD Extensions. *Computer*, 32(12):26 –34, December 1999.

[10] Jack Huynh. The AMD Athlon MP processor with 512KB L2 cache. Technical report, AMD, May 2003.

[11] Advanced Micro Devices. 3DNow! Technology. Technical report, March 2011. `http://www.amd.com/us/products/technologies/3dnow/Pages/3dnow.aspx`.

[12] Dave Sager, Desktop Platforms Group, and Intel Corp. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, 1:2001, 2001.

[13] A. Klimovitski. Using SSE and SSE2: Misconceptions and reality. *Intel Developer UPDATE Magazine*, 26:1–8, March 2001.

[14] Intel Corporation. Intel 64 and IA-32 architectures software developer's manual volume 1: Basic architecture, January 2011.

[15] David Monniaux. The pitfalls of verifying floating-point computations. *ACM Trans. Program. Lang. Syst.*, 30:12:1–12:41, May 2008.

[16] Wenhua Yu, Wenhua Yu, Xiaoling Yang, Yongjun Liu, , Raj Mittra, Dau-Chyrh Chang, Chao-Hsiang Liao, Muto Akira, Wenxing Li, and Lei Zhao. New development of parallel conformal FDTD method in computational electromagnetic engineering. To be published in IEEE Antennas and Propagation Magazine, June, 2011.

[17] Chris Lomont. Introduction to Intel Advanced Vector Extensions. Website accessed on 15/08/2011 `http://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions/`.

[18] T. Fischer, S. Arekapudi, E. Busta, C. Dietz, M. Golden, S. Hilker, A. Horiuchi, K.A. Hurd, D. Johnson, H. McIntyre, S. Naffziger, J. Vinh, J. White, and K. Wilcox. Design solutions for the Bulldozer 32nm SOI 2-core processor module in an 8-core CPU. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2011 IEEE International*, pages 78 –80, feb. 2011.

[19] Damon Poeter. Exclusive: AMD's First Bulldozer Chips Arriving Sept. 26. Website accessed on 15/08/2011 `http://www.pcmag.com/article2/0,2817,2390410,00.asp`.

[20] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 2010.

[21] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *Micro, IEEE*, 28(2):39 –55, March-April 2008.

[22] NVIDIA Corporation. NVIDIA's next generation CUDA compute architecture: Fermi. Whitepaper, 2009.

[23] R. Amorim, G. Haase, M. Liebmann, and R. Weber dos Santos. Comparing CUDA and OpenGL implementations for a Jacobi iteration. *High Performance Computing & Simulation, 2009. HPCS '09. International Conference*, pages 22–32, June 2009. 10.1109/HPCSIM.2009.5192847.

[24] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, and Kevin Skadron. A performance study of general-purpose applications on graphics processors using CUDA. *J. Parallel Distrib. Comput.*, 68:1370–1380, October 2008.

[25] S. Adams, J. Payne, and R. Boppana. Finite difference time domain (FDTD) simulations using graphics processors. In *Proceedings of the 2007 DoD High Performance Computing Modernization Program Users Group Conference*, HPCMP-UGC '07, pages 334–338, Washington, DC, USA, 2007. IEEE Computer Society.

[26] F. Costen, J.-P. Berenger, and A.K. Brown. Comparison of FDTD hard source with FDTD soft source and accuracy assessment in Debye media. *Antennas and Propagation, IEEE Transactions on*, 57(7):2014 –2022, July 2009.

[27] Sandeep.S. Gcc-inline-assembly-howto. Website accessed on 25/05/2011 `http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html`.

[28] Howto: Inline assembly & sse: Vector normalization done fast! Website accessed on 25/05/2011 `http://www.3dbuzz.com/vbforum/`

showthread.php?104753-HowTo-Inline-Assembly-amp-SSE\-Vector-normalization-done-fast!

[29] Microsoft. Streaming SIMD Extensions 2 Instructions. Website accessed 27/05/2011 http://msdn.microsoft.com/en-US/library/kcwz153a%28v=VS.80%29.aspx.

[30] GPU Ocelot project. gpuocelot - A dynamic compilation framework for PTX. Website accessed on 02/05/2011 http://code.google.com/p/gpuocelot/, May 2011.

[31] Dorit Naishlos. Autovectorization in GCC. In *Proceedings of the 2004 GCC Developers Summit*, pages 105–118. IEEE Computer Society, 2004.

[32] The GCC team. GCC 4.1 Release Series - Changes, New Features, and Fixes. Website accessed on 18/08/2011 http://gcc.gnu.org/gcc-4.1/changes.html.

[33] NVIDIA. NVIDIA Tesla Products. Technical report, 2009. http://www.lunarc.lu.se/Documents/nvidia-workshop/files/presentation/20_Tesla_Products.pdf.

[34] NVIDIA. NVIDIA CUDA C Programming Guide 3.1.1. Technical report, March 2011. http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_CUDA_C_ProgrammingGuide_3.1.pdf.

[35] Tim Lanfear. HPC Computing with CUDA and Tesla Hardware. Technical report, 2008. http://www.many-core.group.cam.ac.uk/archive/27_02_09/NVIDIA_lanfear_27_02_09.pdf.

[36] NVIDIA Corporation. TESLA M-Class GPU Computing Modules. Technical report, 2011. http://www.nvidia.com/docs/IO/43395/DS_Tesla-M2090_LR.pdf.

Appendix A

Material provided in electronic form

Included with this dissertation is a compact disc containing various electronic material as detailed below.

- An electronic copy of this document in PDF format.
- The full code, compilation scripts and batch execution scripts used to implement and execute the solutions presented in this project.
- The raw data and charts from the performance experiments performed during this project in Microsoft Excel format.

This material may be of interest to the reader of this document either to enhance the understanding or to allow for further work to be performed.