A comparative analysis of the performance and deployment overhead of parallelized Finite Difference Time Domain (FDTD) algorithms on a selection of high performance multiprocessor computing systems

by

RG Ilgner

Dissertation presented in fulfilment of the requirements for the degree
Doctor of Philosophy in the Faculty of Engineering
at Stellenbosch University

Promoter: Prof DB Davidson
Department of Electrical & Electronic Engineering

December 2013

Stellenbosch University http://scholar.sun.ac.za

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

December 2013

Copyright © 2013 Stellenbosch University
All rights reserved

Stellenbosch University http://scholar.sun.ac.za

Abstract

The parallel FDTD method as used in computational electromagnetics is implemented on a variety of different high performance computing platforms. These parallel FDTD implementations have regularly been compared in terms of performance or purchase cost, but very little systematic consideration has been given to how much effort has been used to create the parallel FDTD for a specific computing architecture. The deployment effort for these platforms has changed dramatically with time, the deployment time span used to create FDTD implementations in 1980 ranging from months, to the contemporary scenario where parallel FDTD methods can be implemented on a supercomputer in a matter of hours. This thesis compares the effort required to deploy the parallel FDTD on selected computing platforms from the constituents that make up the deployment effort, such as coding complexity and time of coding. It uses the deployment and performance of the serial FDTD method on a single personal computer as a benchmark and examines the deployments of the parallel FDTD using different parallelisation techniques. These FDTD deployments are then analysed and compared against one another in order to determine the common characteristics between the FDTD implementations on various computing platforms with differing parallelisation techniques. Although subjective in some instances, these characteristics are quantified and compared in tabular form, by using the research information created by the parallel FDTD implementations. The deployment effort is of interest to scientists and engineers considering the creation or purchase of an FDTD-like solution on a high performance computing platform. Although the FDTD method has been considered to be a brute force approach to solving computational electromagnetic problems in the past, this was very probably a factor of the relatively weak computing platforms which took very long periods to process small model sizes. This thesis will describe the current implementations of the parallel FDTD method, made up of a combination of several techniques. These techniques can be easily deployed in a relatively quick time frame on computing architectures ranging from IBM's Bluegene/P to the amalgamation of multicore processor and graphics processing unit, known as an accelerated processing unit.

Stellenbosch University http://scholar.sun.ac.za

Opsomming

Die parallel Eindige Verskil Tyd Domein (Eng: FDTD) metode word gebruik in numeriese elektromagnetika en kan op verskeie hoë werkverrigting rekenaars geïmplementeer word. Hierdie parallele FDTD implementasies word gereeld in terme van werkverrigting of aankoop koste vergelyk, maar word bitter min sistematies oorweeg in terme van die hoeveelheid moeite wat dit geverg het om die parallele FDTD vir 'n spesifieke rekenaar argitektuur te skep. Mettertyd het die moeite om die platforms te ontplooi dramaties verander, in the 1980's het die ontplooings tyd tipies maande beloop waarteenoor dit vandag binne 'n kwessie van ure gedoen kan word. Hierdie tesis vergelyk die inspanning wat nodig is om die parallelle FDTD op geselekteerde rekenaar platforms te ontplooi deur te kyk na faktore soos die kompleksiteit van kodering en die tyd wat dit vat om 'n kode te implementeer. Die werkverrigting van die serie FDTD metode, geïmplementeer op 'n enkele persoonlike rekenaar word gebruik as 'n maatstaf om die ontplooing van die parallel FDTD met verskeie parallelisasie tegnieke te evalueer. Deur hierdie FDTD ontplooiings met verskillende parallelisasie tegnieke te ontleed en te vergelyk word die gemeenskaplike eienskappe bepaal vir verskeie rekenaar platforms. Alhoewel sommige gevalle subjektief is, is hierdie eienskappe gekwantifiseer en vergelyk in tabelvorm deur gebruik te maak van die navorsings inligting geskep deur die parallel FDTD implementasies. Die ontplooiings moeite is belangrik vir wetenskaplikes en ingenieurs wat moet besluit tussen die ontwikkeling of aankoop van 'n FDTD tipe oplossing op 'n höe werkverrigting rekenaar. Hoewel die FDTD metode in die verlede beskou was as 'n brute krag benadering tot die oplossing van elektromagnetiese probleme was dit waarskynlik weens die relatiewe swak rekenaar platforms wat lank gevat het om klein modelle te verwerk. Hierdie tesis beskryf die moderne implementering van die parallele FDTD metode, bestaande uit 'n kombinasie van verskeie tegnieke. Hierdie tegnieke kan maklik in 'n relatiewe kort tydsbestek ontplooi word op rekenaar argitekture wat wissel van IBM se BlueGene / P tot die samesmelting van multikern verwerkers en grafiese verwerkings eenhede, beter bekend as 'n versnelde verwerkings eenheid.

Stellenbosch University http://scholar.sun.ac.za

Acknowledgments

I would like to thank supervisor Professor David Davidson for his assistance and motivation in the writing of this of dissertation and the lengthy path of enlightenment that has led me here. Thank you to Neilen Marais, Evan Lezar, and Kevin Colville, for their valuable reviews or input. Very many thanks to the helpful technical support crew from the Centre for High Performance Computing for their technical assistance and interesting discussions about High Performance Computing.

My gratitude to my family for giving me the time to do the thesis; in particular my thanks to Marie-Claire for her constant support and helping with the formatting and checking of the thesis.

There are many others that I am grateful to, but am avoiding writing a lengthy list in case I omit someone.

My final acknowledgment is to all of the people who have assisted via articles and discussions in internet forums, blogs and newsgroups. I have been both encouraged by these original interactions, and humbled by the selfless contributions and experience by others. I have found this a source of motivation in completing several part of this thesis.

Stellenbosch University http://scholar.sun.ac.za

Glossary & Acronyms

The reader is reminded that this thesis contains many domain specific acronyms and that these are written out in full at the first mention in every chapter. The Glossary compiled below should serve as a fall-back to identify all the acronyms, some of which are used in the chapter description below.

ABC	Absorbing Boundary Conditions
ALU	Arithmetic Logic Unit
AMD	Advanced Micro Devices; Manufacturer of computer processors
API	Application Programming Interface
APU	Accelerated Processing Unit
ARM	Advanced RISC Machines; Manufacturer of RISC based computer processors
ASP	Associative String Processor
ASIC	Application Specific Integrated Circuit
ATI	Array Technologies Industry; GPU division for AMD processors
AUTOCAD	Commercial Computer Aided Design software
AVX	Advanced Vector eXtensions
Blade	A high performance process board consisting of closely coupled processors
C	Higher level computing language
C++	Higher level computing language using class structures
CAD	Computer Aided Design
CHPC	Centre for High Performance Computing in Cape Town
CISC	Complex Instruction Set Computing
CMP	Chip-Multiprocessors
CMU	Computer Memory Unit: A high performance board containing several processors and associated memory; Used in the SUN M9000
Core	A core is considered to be a physical component of a processor performing at least one program thread
CPU	Computer Processing Unit
CUDA	Compute Unified Device Architecture
DirectCompute	Application programming interface that supports General Purpose computing on graphical processors
DirectX11	Application programming interface for graphics and games programming
DRAM	Dynamic Random Access Memory
DSMP	Distributed Shared Memory Processor
Dxf	Drawing eXchange Format; an AUTOCAD data exchange file format
FDTD	Finite Difference Time Domain
FEKO	Electromagnetic modelling software based on MOM
FEM	Finite Element Method
FMA	Fused Multiply Add, a hardware based acceleration combining the multiplication and the addition instruction

Stellenbosch University http://scholar.sun.ac.za

FORTRAN	FORmula TRANslation; a higher level computing language
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
GB	Giga Byte
GEMS	Electromagnetic modelling software based on FDTD
GFLOP	Giga Floating point Operations
GPGPU	General Purpose GPU
GPS	Global Positioning System; a system of satellites used for global navigation
GPU	Graphical Processing Unit
GPUDirect	A technology from Nvidia that will allow the exchange of data between dedicated GPU memories without having to access the Host computer
Hadoop	High performance data storage system
HPC	High Performance Computing
IC	Integrated Circuit
IEEE	Institute of Electrical and Electronics Engineers
Intel	Manufacturer of computer processors and other computing hardware and software
LISP	Locator/Identifier Separation Protocol; one of the original higher level programming languages
LTE	Long Term Evolution; a fourth generation mobile phone communications technology
Matlab	A third generation computer language for numerical modelling
MCPs	Million Cells Per second
MIC	Many In Core
MIMD	Multiple Instruction Multiple Data
Moab	A job scheduling mechanism used by supercomputing centres to schedule the running of jobs on their machines
MOM	Method of Moments
MPI	Massively Parallel Interface
MTL	Manycore Test Lab
Node	a node is a processing cluster such as a Blade processor, multicore processor or even a single CPU
NUMA	Non Uniform Memory Architecture
numPy	Library of numerical functions for the Python Language
Nvidia	Manufacturing of a popular brand of GPUs
Octave	A third generation computer language for numerical modelling
OSI	Open Systems Interconnection
OTS	Off-The-Shelf
PCI	Peripheral Component Interconnect
PCIe	Peripheral Component Interconnect express
PEC	Perfect Electric Conductor
PMC	Perfect Magnetic Conductor
PML	Perfectly Matched Layer
POSIX	Portable Operating System Interface, a set of standards from the IEEE for maintaining compatibility between operating systems
PVM	Parallel Virtual Machine
QPI	Quick Path Interconnect

Stellenbosch University http://scholar.sun.ac.za

RISC	Reduced Instruction Set Computing
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Thread
SM	Streaming Multiprocessor
SMP	Shared Memory Processor or Platform
SP	Stream Processor
SSE	Streaming SIMD Extensions
TE	Transverse Electric field
Thread	A process performing instructions on a physical processor
TM	Transverse Magnetic field
UMA	Uniform Memory Architecture

Stellenbosch University http://scholar.sun.ac.za

Contents

1 Introduction . . . 1

1.1 The objective . . . 1
1.2 The FDTD for electromagnetics in high performance computing . . . 2
1.3 High performance computing . . . 3
1.4 Research approach . . . 3
1.5 Chapter overview . . . 4

2 Literature Review of the FDTD on HPC platforms . . . 5

2.1 Overview . . . 5
2.2 History of the FDTD in the context of computational electromagnetics 5
2.3 FDTD software development . . . 7
2.4 Previous examples of FDTD deployments on a variety of HPC platforms . . . 9
2.4.1 NUMA (Non-uniform memory architectures) . . . 9
2.4.2 UMA (Uniform Memory Architecture) . . . 11
2.4.3 Acceleration devices . . . 13
2.5 Summary . . . 15

3 The FDTD algorithm . . . 16

3.1 Overview . . . 16
3.2 Maxwell's equations . . . 17
3.3 The FDTD modelling scenarios . . . 20
3.4 Cell sizing, algorithmic stability . . . 21
3.5 Boundary conditions . . . 21
3.6 EM sources and reflectors . . . 22
3.7 Validation of the FDTD code . . . 23
3.8 Computing Languages and APIs . . . 23
3.9 Performance profiling the serial FDTD . . . 26
3.10 Estimate of the Parallel FDTD coding effort . . . 27
3.11 Summary . . . 29

4 Architectures of HPC system . . . 30

4.1 Overview . . . 30
4.2 Hardware architectural considerations . . . 31
4.2.1 The Processing engine/ the Compute engine . . . 33
4.2.2 Memory Systems . . . 34
4.2.3 Communication channels . . . 36
4.2.4 Hyper-threading (HT)/Symmetric Multithreading (SMT) . . . 39
4.3 Examples of HPC systems . . . 40
4.3.1 Discrete GPUs . . . 40
4.3.2 Accelerated programming units . . . 41
4.3.3 SUN M9000 Shared Memory Processor (SMP) . . . 43
4.3.4 Intel nulticore at the Manycore Test Lab (MTL) . . . 44
4.3.5 Nehalem cluster computer and Westmere cluster computer . . . 44

Stellenbosch University http://scholar.sun.ac.za

4.3.6 Bluegene/P (IBM's massively parallel processing system). 46
4.3.7 Intel's Many In Core (MIC) processor 48
4.4 Other considerations. 49
4.4.1 Power consumption of high performance computer systems. 49
4.5 Summary. 51

5 Implementation of parallelisation methods for the FDTD. 52

5.1 Overview. 52
5.2 Amdahl's law, Koomey's law. 53
5.3 Parallel FDTD processing dependencies. 56
5.4 Parallel FDTD with openMP. 59
5.5 Parallel FDTD with MPI. 62
5.6 Acceleration of the FDTD with Vector ALU/Streaming SIMD Extensions and cache 66
prefetching. 71
5.7 Parallel FDTD for the GPU using CUDA (Compute Unified Device Architecture). 81
5.8 Parallel FDTD with low level threading. 81
5.9 Parallel FDTD on a cluster of two GPUs. 84
5.10 Parallel FDTD on the GPU using the DirectCompute interface. 86
5.11 Accelerated Processing Units and Heterogeneous Systems 87
5.12 Summary. 87

6 A comparison of the FDTD method on different HPC platforms. 89

6.1 Overview. 89
6.2 Comparison of FDTD implementations. 90
6.2.1 GPGPU and SMP. 90
6.2.2 GPGPU and multicore SIMD. 95
6.2.3 SMP and Bluegene/P. 99
6.2.4 Cluster and SMP. 100
6.2.5 Integrated GPGPU and discrete GPGPU. 104
6.2.6 Other systems. 105
6.3 General classification table. 106
6.3.1 Scalability of process. 106
6.3.2 Scalability of hardware. 107
6.3.3 Memory Access Efficiency. 107
6.3.4 Openness. 109
6.3.5 Algorithm encoding. 109
6.3.6 Performance in Speed. 110
6.3.7 Performance in terms of model size. 110
6.3.8 Cost of purchase. 111
6.3.9 Power consumption. 112
6.3.10 Implementation time. 112
6.3.11 Future proofing from aspect of coding. 113
6.4 Summary. 113

Stellenbosch University http://scholar.sun.ac.za

7 Conclusions. 114

7.1 Contributions. 114
7.2 Publications. 115
7.3 Trends in FDTD and HPC 116
7.4 Possible directions for future work 116
7.5 Conclusions. 117

8 Bibliography. 119

9 Appendix - Verification of the FDTD implementations. 131

9.1 Objectives of the verification 131
9.2 Far Field modelling with the FDTD 131
9.3 The FDTD model 135
9.4 Modelling with FEKO 136
9.5 A comparison of the FDTD and FEKO derived MOM results 136

Stellenbosch University http://scholar.sun.ac.za

Chapter 1

1 Introduction

1.1 The objective
1.2 The FDTD for electromagnetics in high performance computing
1.3 High performance computing
1.4 Research approach
1.5 Chapter overview

1.1 The objective

This thesis compares the deployment effort of the parallel FDTD, as used for Electromagnetic modelling, on to a variety of High Performance Computing (HPC) platforms. It is a maxim of hardware and software development that the two will always evolve together and this thesis will show that the evolution of the FDTD corresponds with developments in HPC. The parallelisation of the FDTD usually provides two advantages over the serial FDTD, one being the reduction of the processing time of the FDTD and the other the ability to process extremely large data sets.

The method of converting from the serial operation of the FDTD to the parallel FDTD differs from one platform to the next. A comparison is made of the deployment mechanics and the resultant performance achieved. An indication is given of the complexity involved in converting the FDTD to a particular HPC platform in the form of a tabulated coding effort. Other considerations that will be made are issues relating to accessibility to the system and dedicated usage.

The examination of the deployment effort involves a comparison of:

1) A description of the architecture and features of HPC platforms, and the suitability of that platform for the FDTD.
2) The mechanics of communication between parallel FDTD components.
3) How the serial FDTD code is decomposed and reconstituted in parallel form for a specific HPC platform, i.e. the implementation strategy.
4) The relative performance achieved by the parallel FDTD on a particular HPC platform.
5) Implementation effort, skills required, future proofing.
6) Accessibility to the development system, development tools, etc.

The comparison of the performance of one computing platform against another is a subjective issue in the sense that the range of hardware defining a platform varies greatly. While it may be obvious from the detail shown in this thesis that the Graphical Processing Unit (GPU) can process FDTD cells at a much greater rate than a single core CPU, it may not be immediately evident that the FDTD processing power of different types of GPUs varies greatly. This aspect needs to be borne in mind

Stellenbosch University http://scholar.sun.ac.za

when one makes a comparison of the parallel FDTD implemented on a GPU against a larger platform such as the SUN M9000 Shared Memory Processor (SMP).

One very notable of about this work is that it is written during a period of intense development in the field of HPC. Hardware and software platforms are being revised and improved at an ever faster rate, as will be described in more detail (chapter 2). One personal experience is a good example of the rapid evolution of HPC platforms. A microprocessor development platform was purchased to investigate the FDTD speedup using the Streaming SIMD Extension (SSE) of a multicore processor, only to find that an enhanced form of this functionality, the Advanced Vector eXtensions (AVX), was being released sooner than expected and would require the acquisition of a new microprocessor as a trial development platform. Unfortunately this new microprocessor also required a new mother board as the microprocessor pin configuration was not the same as that of the previous generation. The new microprocessor used a motherboard slot configuration with 1155 pins, as opposed to the older microprocessor which uses a motherboard slot with 1156 pins. The moral of this little example is that unless one has unlimited resources to continuously purchase new hardware, one needs to "Futureproof" the FDTD deployment. This concept is one of the contributing factors to the deployment effort and is addressed in the thesis.

The GPU is another example of rapid evolution of the FDTD and hardware deployment. The hardware efficiency and computing strength of GPUs has grown rapidly over the last decade, and the introduction of relatively friendly application interfaces, such as Nvidia's CUDA and the Khronos' group openCL, has made the GPU accessible to the Computational Electromagnetics (CEM) market.

1.2 The FDTD for electromagnetics in high performance computing

The FDTD is a method of approximating Maxwell's equations for electromagnetics using finite differences equations. In the practical sense the FDTD algorithm is used to model diverse electromagnetic scenarios such as:

1) The effect of radiation on human tissue from a variety of electromagnetic sources.
2) Modelling of the earth's geological sub surface in the oil and gas industry.
3) Modelling of electromagnetic waves for antenna and transmitter design.

Electromagnetic scenarios such as the modelling of complex antenna systems or the reflection of laser beams from the surface of DVDs can be modelled in detail at ever finer resolution and this in turn increases demand for computing power when modelled using the FDTD method. It will be shown that the research and development in the FDTD method and HPC is very closely related.

The basic computational mechanics of the FDTD can be traced back to seminal papers by Kane Yee [3] and a series of publications by Taflove and Hagness [2, 3] in the early 1980s. The FDTD algorithm is computationally intense and in the emerging years of the method, the FDTD modelling space was limited to the meagre restrictions in memory available at that time. Longer computations have been known to run for a matter of days and HPC hardware was not readily accessible during that period.

Stellenbosch University http://scholar.sun.ac.za

The improvement in HPC capacity since the end of the 20th century has promoted the development of the FDTD as an electromagnetic modelling method.

1.3 High performance computing

The FDTD has been implemented for this thesis on computers that represents a cross section of the high performance computers in the world. Supercomputing hardware has been made available from the Centre for HPC. These platforms include supercomputers such as IBM's Bluegene/P, SUN M9000, and several types of large Cluster computers based on multicore processors. Intel has made available a group of tightly coupled multicore processors as an online facility known as the Manycore Test Lab (MTL). An assortment of other multicore processors and GPUs has been purchased specifically for the deployment of the parallel FDTD.

1.4 Research approach

The research approach was focused on converting the serial form of the FDTD method into a parallel FDTD implementation for a specific platform. The intention of the research is to repeat this implementation exercise for a wide variety of different high performance hardware platforms and to compare the deployment effort for each of these. For each of the implementations the method of deployment is similar:

1) Perform a background search on previous implementations of the FDTD or similar algorithms for the chosen platform.
2) Convert the serial FDTD into parallel form by simulating the target architecture on the PC. For some architectures, such as the discrete GPU, this simulation is not necessary as the hardware is available throughout the deployment.
3) Implement the parallel FDTD on the chosen platform and verify correct operation by comparing the results from a serial version of the FDTD.
4) Benchmark the performance of FDTD models with the results from the parallel implementation.
5) Analyse the features of the deployment effort such as coding time and usefulness of documentation.
6) Compare the parallel implementation against other implementations undertaken for this thesis and those from other publications.

Stellenbosch University http://scholar.sun.ac.za

1.5 Chapter overview

Chapter 1 Introductory chapter describing the content of the thesis. It introduces the FDTD as an algorithm, and describes some aspects of HPC.

Chapter 2 Literature Review. This chapter is a review of some of the current and historical literature about the parallelisation of the FDTD method and the high performance hardware that is available for the processing of the FDTD. It examines the emergence of the HPC platforms and the use of the FDTD. The history of the FDTD and the influence on its deployment with parallel programming is analysed using data from IEEE publications.

Chapter 3 The FDTD algorithm. Chapter 3 is an examination of the mathematical foundation of the FDTD method and its conversion into a programmable algorithm. It describes the creation of an electromagnetic modelling space using Yee's formulation and the features used in the model such as electromagnetic sources and reflectors. The formulation of the far-field is described and the electric far field of an electric dipole is compared against that computed with FEKO's MOM program. A performance profile of the serial FDTD is created as a pre-cursor to the parallelisation of the FDTD. A tabular comparison is made of the coding effort required to produce various parallel FDTD programs.

Chapter 4 The architecture of HPC systems. The various types of HPC hardware used to parallelise the FDTD for this thesis are reviewed in some detail. Some of the components that make up a HPC system, such as the processors, memory and communication subsystems are described and compared. A description is made of how each of these component groups affect the parallelisation of the FDTD. The HPC systems used to parallelise the FDTD for this thesis are described in the context of the aforementioned component groups and their related implementation by the FDTD.

Chapter 5 Parallelism strategies for the FDTD. This chapter deals with the different methods used to parallelise the FDTD method. It examines the application of a variety of threading techniques and forms of data parallelism used for the parallelisation process. A description is made of how the parallelised FDTD is applied to a variety of HPC platforms with these threading and data manipulation techniques.

Chapter 6 A comparison of the parallel FDTD on various HPC systems. The parallelised FDTD on various HPC platforms are compared so as to analyse some of the deployment features for these systems. A series of characteristics representing the parallel FDTD deployment effort are compared in tabular form.

Chapter 7 Conclusion and Future Work. This chapter provides summary of the content of the thesis and the conclusions drawn from this. The contributions derived from the research, are listed here.

Stellenbosch University http://scholar.sun.ac.za

Chapter 2

2 Literature review of the FDTD on HPC platforms

2.1 Overview
2.2 History of the FDTD in the context of computational electromagnetics
2.3 FDTD software development
2.4 Previous examples of FDTD deployments on a variety of HPC platforms
    2.4.1 NUMA (Non-uniform memory architectures)
    2.4.2 UMA (Uniform Memory Architecture)
    2.4.3 Acceleration devices
2.5 Summary

2.1 Overview

This literature review provides an overview of the history of the Finite Difference Time Domain (FDTD) method and its relationship to High Performance Computing (HPC). The review is a summary of some of the publications and technology that have been created concerning the deployment of the FDTD on high performance computers and is categorised according to the hardware architecture that the FDTD is deployed on. Most of the topics discussed in this literature review are discussed in more detail in the chapters that follow. The discussion below introduces technologies that will be described in this thesis. The deployment of the FDTD on the Graphical Processing Unit (GPU), for example, requires substantial background technical detail but is in itself not the main focus of this thesis.

2.2 History of the FDTD in the context of computational electromagnetics

It is necessary to review existing implementations of the FDTD on HPC platforms in order to determine and understand the factors used in the comparison of these deployments.

The FDTD method is used in a wide variety of applications. It is used in electromagnetic modelling of antennas [38], medical diagnosis of cancers [61], geophysical earth models [62], seismics [28], and military simulations of tanks engaging each other on the battlefield [29]. The FDTD method for electromagnetic has resulted from a culmination of several key formulations:

1) Maxwell's equations [7];
   These are a set of difference equations that describe the behaviour of electromagnetic fields, how a time varying electric field generates a magnetic field, and the other way round.
2) Determination of a suitable algorithmic framework by Yee [1];

Stellenbosch University http://scholar.sun.ac.za

The electric field vector and magnetic field vector components of the electromagnetic field are solved as a finite difference equation in a leap-frog manner. The electric vectors computed an instant in time and the magnetic field vectors an instant later. This process is repeated until some conclusion is reached. This formulation was devised by Yee in 1966.

3) Computational power provided by HPC platforms ;
   The FDTD computation requires large resources in terms of memory and processing capacity it is only made practical when implemented on HPC platforms.
4) Sophisticated Graphical User Interfaces (GUI)s and output graphics for the manipulation of large and complex data models;
   To avoid hard coding input modelling geometries, or the use of complex configuration languages, all commercial electromagnetic modelling software has a sophisticated graphical data entry mechanism. The volume of data generated by the FDTD may be very difficult to analyse in raw numerical form and it is more practical and intuitive to display the results in some form of graphical representation.

Owing to the large demand made on computing power and capacity by the FDTD process, the use of the FDTD and High performance Computing are very closely related, as is shown in figure 2.1. As will be illustrated by some of the results in later chapters, the FDTD can consume extremely large computing resources. Although the FDTD method for electromagnetics was formulated in the 1960s, it has required the processing capability and capacity of High Performance computing to make this simulation technique a viable proposition.

The relationship in the number of annual Institute of Electrical and Electronics Engineers (IEEE) publications shown in figure 2.1 below, is an indication of the relationship between the FDTD, HPC and the keywords “parallel program”. The graphic shows the steady increase in the FDTD and parallel programming publications from the IEEE for a particular year. The depression in “parallel program” publications between the years 1998 to 2003 could be as a result of the global economic downturn in that period. The general upward trend in time of these curves indicates that there may be some positive relationship between all of these topics, i.e. it is very likely that the development of HPC and parallelisation techniques stimulated the development of the FDTD method.

Stellenbosch University http://scholar.sun.ac.za

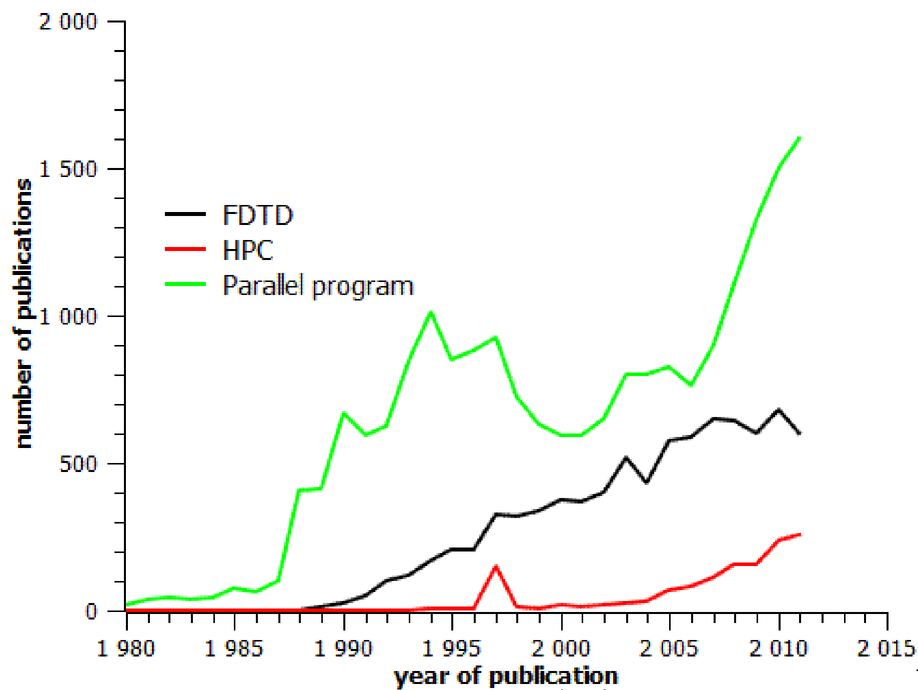

Figure 2.1: The number of IEEE publications found with specific keywords.

The FDTD has been implemented on many different computing platforms [4, 5, 6] and owing to its demand for computing capacity, it is a suitable algorithm for a comparison that examines the deployment of this type of software on High Performance Computing platforms. The FDTD has been implemented on diverse multicore processing systems such as the Bluegene/P and the GPU [2, 3, 4]. The parallel FDTD has been described as being an “explicitly parallel” implementation and also as “embarrassingly parallel”. An embarrassingly parallel implementation is one where the concurrent processing of the algorithm requires no communication between compute nodes, and can be scaled nearly linearly. An explicitly parallel implementation is one where sections of the algorithm have been devised to perform independently in parallel, yet require constant communication and process synchronisation between all instances of the deployment. Embarrassingly parallel implementations, such as those algorithms using the Monte Carlo technique, are used by hardware manufacturers to benchmark the peak performance of their products. One cannot equate the performance achieved by an embarrassingly parallel process with an explicitly parallel process on one computing platform, i.e. an embarrassingly parallel implementation has the potential to achieve the peak performance stated for a specific piece of hardware, whereas the explicitly parallel does not. The implication of this is that the performance of the FDTD on HPC systems cannot be extrapolated from benchmarking software such as the LINPACK standard alone, as the communication between the processing elements of the FDTD play a major part in the computation.

2.3 FDTD software development

The implementation of the FDTD on any high performance hardware platform requires three essential functional components to ensure the practical use of the FDTD. These are:

Stellenbosch University http://scholar.sun.ac.za

1) A graphical input facility that will allow the creation of sophisticated models to be processed or simulated. For very large models it is not practical to set up the model data manually and this has to be performed within the controlled and ordered environment of a computer aided design environment. A detailed record of a sophisticated GUI for the FDTD has been produced by Yu and Mittra [135]. It describes the creation of an input facility for the physical model, mesh creation, and other electromagnetic parameters. An intermediate store of the FDTD model is made in the dxf standard, a popular graphic data format used by Computer Aided Design products such as AutoCad. This intermediate FDTD model is then used as input to an FDTD solver mechanism, as discussed in the next point. It is also of interest that a dxf formatted input model can also be used as the input for other electromagnetic solvers such as Finite Element Method (FEM) or MOM [136, 137].
2) The FDTD processing engine which will process the simulation itself. The processing engine will need to be coupled with a high performance data storage mechanism that will record the output of the simulation. High performance data storage devices such as Hadoop [138] or the Message Passing Interface (MPI) are available to record the output from the electromagnetic solver.
3) A data display system that will allow the rendering of the simulated data. This simple requirement cannot be understated as the result of a large simulation in plain numerical form is not easy to visualise without substantial post processing. The GPU is a hardware processing board dedicated to the display of high resolution graphical information. Primarily deployed for computer gaming, it can also be used to display the results of electromagnetic computations. These results are normally displayed as a part of the post processing of the electromagnetic modelling process. One publication that describes the creation of a single program for both the processing of the FDTD and the graphical display of the resultant information is described in [91]. Also of interest in this publication [91] is the use of the GPU graphics as a low level program debugging tool, an integrated feature that is usually only found in high level languages such as Matlab or Python.

The conversion of the serial FDTD into a parallel form is dependent on the architecture of the HPC hardware. Some authors [9] have categorised the processing of the parallel FDTD using a characteristic of the number of data sources and instruction streams on Single Instruction Multiple Data (SIMD), Single Program Multiple Data (SPMD) and Multiple Instruction Multiple Data (MIMD) architectures. The implementation of the FDTD on some of these architectures can extend valuable implementation lessons for other hardware platforms. Data manipulation techniques used in the SIMD processing of the FDTD with SSE [132] for example, can be applied to the Single Instruction Multiple Thread (SIMT) processing of the FDTD on the GPU in that these both use a data parallel approach.

The capabilities of the SIMD, SPMD and MIMD processing schemes are not restricted to the FDTD processing method; they will also apply to other modelling techniques based on principles such as the MOM or the FEM. Products such as FEKO and GEMS are commercial examples of where the model creation, processing engine and result visualisation have been bundled into one integrated electromagnetic modelling system.

Higher level software languages such as Matlab, Octave or Python provide convenient higher level functionality to implement the FDTD processing engines, graphical input method and the

Stellenbosch University http://scholar.sun.ac.za

visualisation mechanism within one computational system. These software products have an advantage over conventional lower level language tools such as C or Fortran in that all of the functionality to experiment and produce results with an FDTD engine is available within one user environment. A disadvantage of such high level languages is that high performance processing tools need to be made available to the user but do not have ready access to low level interfaces such as CUDA or DirectX11. As with all computing development systems, the convenience of platform abstraction offered by scripting environments such as Matlab are offset by the lack of programming flexibility for dedicated high performance platforms such as GPU or Field Programmable Gate Array (FPGA) accelerators.

The focus of this work lies in the comparison of the FDTD engine as deployed on a variety of High performance processing platforms. To this end the graphical input and rendering components have been hard coded independently for most deployments and are not considered to be part of this evaluation.

Special mention must be made of the development of the first parallel FDTD implementation by Taflove et al [130] on a CM1 computer (which initially only had a LISP compiler). When a Fortran compiler was made available for this platform, it took approximately a year to develop the first parallel FDTD algorithm, six months of which was taken up waiting for bug fixes in the Fortran compiler. In the twenty years since these first efforts at the parallelisation of the FDTD, the high performance hardware, software techniques and FDTD knowledge base, has matured to the extent that contemporary attempts at parallelisation of the FDTD take only a fraction of this time.

2.4 Previous examples of FDTD deployments on a variety of high performance computing platforms

The FDTD has been implemented in parallel on several types of HPC architecture. These can be classified as:

1) NUMA (Non Uniform Memory Architecture) or distributed architecture.
2) UMA (Uniform Memory Architecture).
3) Use of accelerators or co-processors.

2.4.1 NUMA (Non-uniform memory architectures)

Examples: Bluegene/P, Nehalem Cluster, IBM e1350 cluster, Beowulf Cluster

NUMA architectures include systems such as Beowulf Clusters and other systems of distributed computers. These systems are generally comprised of a distributed “cluster” of Computer Processing Units (CPU)s which are afforded inter-process communication by Local Area networked connections [42]. A simplistic view of the NUMA architecture is shown in figure 2.1 which illustrates the distributed relationship of memory and processors.

The Non-Uniform Memory Architecture can also be referred to as distributed computing platform, encompassing a loosely networked system of individual computers ranging from systems such as the

Stellenbosch University http://scholar.sun.ac.za

Simple Network of Workstations [22, 49] to the formidable arrangement of individual processors and network interconnect comprising the Bluegene/P supercomputer [17, 19].

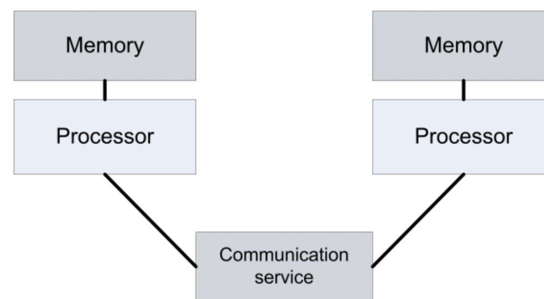

Figure 2.1: Simplistic view of the NUMA architecture

A benchmark implementation of the FDTD with Message Passing Interface (MPI) [9] serves well as a basic implementation framework for the NUMA architecture. Although published as early as 2001 the task parallel implementation of the FDTD using a Message Passing Interface (MPI) by Guiffaut and Mahdjoubi [9] can still be implemented on most multicore clusters with very little change to the parallelisation method.

As will also become apparent from subsequent chapters, the parallel FDTD for distributed systems requires regular communication of data between the parallel FDTD components or chunks [9]. This communication enforces processing synchronisation of the parallel FDTD components between each iteration cycle.

Cluster computing platforms such as the “Beowulf cluster” originated from a need to provide a low cost HPC platform for the parallel FDTD method. The first Beowulf clusters consisted of several off-the-shelf servers connected together via high speed ethernet connection [5] and could provide performance equivalent to a shared memory computer costing 10 to 100 times as much [93]. It was found that this cluster arrangement worked optimally for the creation of parallel FDTD implementations of up to 30 servers [19]. The limiting effect of inter-process communication [119] on the speedup of the FDTD was identified and formulated in 1994, with the use of a cluster of workstations. A typical processing speed of a 3D FDTD model on 8 workstations in 1994 was approximately 0.12 million cells per second (MCPs). It is noteworthy that although the performance of the FDTD on the cluster architecture has improved radically since 1994, the programming paradigm, and hence the deployment effort of the FDTD on a cluster using the Parallel Virtual Machine (PVM) or (MPI) have not. From the aspect of investing money to build computing algorithms, this would be highly desirable.

Larger implementations required better inter-processor communication and this was only possible with the implementation of network interconnects such as the Infiniband network inter-connect [75]. Although the introduction of ten GigaByte (GB) Ethernet not only improved the bandwidth between processing cores compared to the one GB standard, it also reduced the average communication latency by a factor of ten over the one GB Ethernet.

Stellenbosch University http://scholar.sun.ac.za

Where the regular Gigabit Ethernet network connection is enabled via a regular OSI network stack, the Infiniband has a dedicated communication protocol with a reduce protocol stack or structure and therefore provides much lower inter-processor communication latency.

An implementation of the parallel FDTD on a cluster system using the MPI libraries [50] has shown very close agreement between the results obtained from the serial form of the FDTD and the parallel FDTD. This comparison is of note as the parallel FDTD implemented for this work uses a similar deployment strategy.

The NUMA approach has been implemented on 7000 cores of the Tainhe-1A supercomputer, a computing platform that consumes 4MW of power and has a staggering 229,376 GB storage [131]. The FDTD with MPI on the Tianhe-1A has achieved an efficiency of 80% on these 7000 cores [131], although theoretical rating of FDTD computing speed in Millions of Cells per second is not available. In terms of scaling it is also interesting to note that the FDTD was only implemented on a fraction (7000 cores of the 112000) of the Xeon cores available. In addition to this the Tianhe-1A has 7000 GPUs, none of which were deployed for the FDTD simulation.

A good entry point to deploying an application on the Bluegene/P supercomputer is by Morozov [83], despite this publication not being specifically directed at the FDTD. It makes the programmer aware of all the features available for the FDTD application and gives an insight into FDTD specific concerns such as the memory latency between nodes.

The FDTD has also been implemented on GPU clusters [28, 129], a configuration of several GPUs being controlled by a host processor. The nature of the FDTD deployment is also different from the task parallel oriented FDTD parallelisation on multicore clusters in that the parallelisation is done on a data parallel basis.

2.4.2 UMA (Uniform Memory Architecture)

Example: SUN M9000, Intel Nehalem with Quick Path Interconnect(QPI).

Commercial High performance Shared Memory systems from SUN Microsystems have been around since the early 1980s and have been extensively used in industries such as Seismic Data Processing [28, 29]. As all the processes in a UMA device use the same program address space, the inter-process communication required by the FDTD can be reduced to the movement of data pointers in memory [10, 12]. The schematic in figure 2.2 illustrates the relationship the processors have with the memory in a UMA, or shared memory system. This greatly reduces the inter-process communication latency and allows many processing cores to be included in the one address space. Although this architecture allows the component processors to communicate so effectively via the memory it is unfortunately not possible to continuously add compute cores, owing to the physical restrictions such as pin availability on memory chips. To this end Shared Memory Processors are tied together by a functionally transparent memory fabric called the memory interconnect.

 Stellenbosch University http://scholar.sun.ac.za

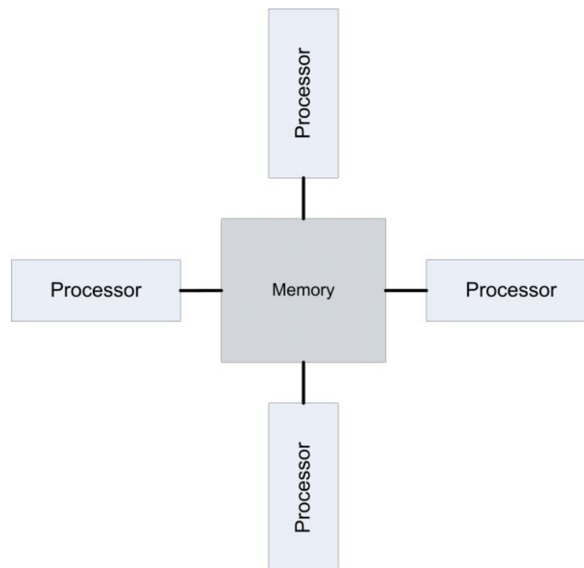

Figure 2.2: Simplistic view of the Uniform Memory (or Shared Memory) Architecture

In order to facilitate the rapid development of programming of parallel application on shared memory platforms, the openMP application programming interface (API) was devised by a collection of hardware and software vendors. The openMP API works on the basis of compiler directives that will allow the work performed by programming loops to be shared (or spawned) between several programming threads [140].

The MPI based parallel FDTD implementation, created for the NUMA environment [9], performs equally well on (Unified Memory Archtecture) UMA platforms. The MPI definition has created a special operating mode where inter thread communication happens only in memory, known as Nemesis mode [10].

The FDTD has been deployed on UMAs using the openMP threading library, the PVM [48, 119] and the MPI by several institutions. This indicates that the parallel implementation on the UMA platform is a mature technology as is illustrated by the speedup of the FDTD on an Shared Memory Processor (SMP) platform dating back to the year 2000. An example of FDTD performance [41] on an earlier SMP platform using eight Silicon Graphics processors achieved a peak FDTD throughput of approximately 2.6 million cells per second.

The inter-processor communication latency of clustered NUMA systems has been reduced to values that will allow a distributed system architecture to behave as if it were a UMA. The Quick Path Interconnect (QPI) technology used by Intel will allow a collection of discrete processing nodes to use physically distributed memory integrated circuits to be accessed as one unified address space. Although each multicore processor is attached to its own physical memory via the input-output handler, an interconnect fabric is provided by the multiple links between the processing cores and the memory by several QPI links.

Although highly successful, earlier implementations of the FDTD on Distributed Shared Memory Processing platforms [41] were limited in the performance they could achieve by the inter process communication latency and memory access bottle-necking. The latency problem can be greatly

Stellenbosch University http://scholar.sun.ac.za

reduced with the use of multicore technology using the QPI fabric as an interconnect, as will be shown later in this work.

2.4.3 Acceleration devices

Example: GPU, GPGPU, FPGA (custom hardware), Intel Phi

Two of the main reasons why the parallel FDTD has not been implemented more frequently on supercomputers or other HPC platforms are that these devices were originally very expensive and very inaccessible to the average academic. Accelerator devices include Intel's 8087 maths co-processor, which is one of the earlier examples of an acceleration device used in conjunction with algorithms running on an associated processing system, in this case Intel's 8086 line of processors. The functionality performed by the 8087 co-processor was eventually migrated onto the main processing circuit and this process appears to be repeating itself in that the functionality provided by accelerator cards such as the GPU is being migrated onto the die of the microprocessors IC. The introduction of the APU, or Fusion technology (an AMD trademark term), integrated circuit bears witness to this. Figure 2.4 shows the relationship between the host processor and the accelerator. Note that the accelerator may have its own dedicated memory.

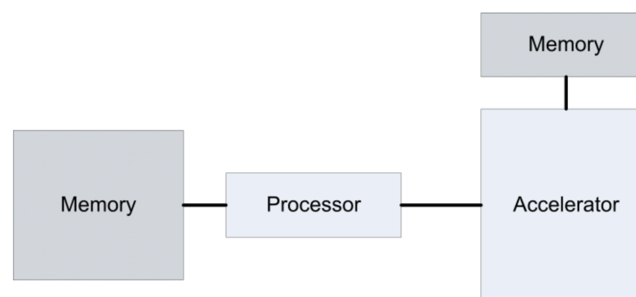

Figure 2.4: Accelerator (example: GPU or secondary processor) attached to host processor.

GPUs are breaking through this cost and availability barrier by providing off-the shelf High Performance Computing solutions for less than the price of a conventional computer workstation. Similar experiments were conducted using small clusters of Transputers [22, 23] which, although being as cheap as GPUs are today, were limited in memory allocation per Transputer, and were prone to long communication latency between the Host and the Transputer array.

One of the first mentions in the academic literature of a GPU like device used for the processing of the FDTD [42] is the Associative String Processor (ASP), an SIMD device used for graphics programming incorporated in the early DEC VAX of 1993. The ASP was configured as co-processor to a host machine and was therefore used as an accelerator board in the same way as an off-the-shelf GPU. Even the description of the SIMD processing performed on the ASP is very similar to what is performed on the GPU [42].

There are two main approaches to implementing the FDTD for the GPU. Although both of these approaches use a data parallel approach as a basis for the parallelisation, they are entirely different programming paradigms [116, 25]. The original FDTD implementations for the GPU involved the coding onto the graphical hardware directly, also referred to as the graphical pipeline model [91].

Stellenbosch University http://scholar.sun.ac.za

GPU programming is a different programming paradigm when compared to the paradigm used for the conventional CPU architecture. Programming on the GPU [121] in this manner allows direct interaction with the graphical pipeline functionality provided by the GPU.

The alternative method has been termed GPGPU by the computing industry as it uses a more conventional computing model incorporating compute cores, registers, and shared memory (cache) to parallelise the data. The GPGPU approach to parallelise the FDTD has been used for this work.

Since starting this work the volume of detail available for FDTD implementations on GPUs has increased at an enormous rate. The term GPGPU has emerged to describe GPU applications that are directed at processing numerical algorithms more efficiently and are not always concerned with the graphical output used in the visualisation of these algorithms, as the use of the term GPU would suggest. The website www.gpgpu.org has been established as a central resource for web user access to GPU related information concerning the programming of numerical algorithms.

The deployment of the FDTD algorithm on the GPU has been described in several publications [116, 117, 27, 44] and involves the decomposition of the multidimensional FDTD grid into a format suitable for processing as a Single Instruction Multiple Thread (SIMT) based algorithm on the GPU. Many FDTD deployments for the GPU are implemented on an NVIDIA platform using the CUDA interface and this has special significance regarding the objectives of this thesis. The question needs to be asked as to why these FDTD implementations have been built predominantly with CUDA as opposed to some other interface. Intel Corporation is currently the world leading GPU supplier and AMD also produces High Performance GPUs, but the implementation of the FDTD on the platforms is minimal in the academic literature. How can the lack of academic literature on the FDTD deployment on an Intel or AMD GPU be meaningful in the comparison of the FDTD deployment on other HPC platforms? The answer to this is directly related to the availability of CUDA GPU implementation examples, CUDA development forums and easy to understand documentation, something not available for the other GPU development platforms at the onset of the GPGPU development revolution.

Although a description of GPU based FDTD implementations has been presented in various publications, the publication by [27] is of particular note as it addresses the implementation of various PML by the GPU. The implementation of the boundary conditions becomes particularly complex when one mixes arrays and this publication illustrates a solution to the complex coding by using boundary calculations that are independent of the calculations of the main grid.

The Field Programmable Gate Array (FPGA), an integrated circuit designed to be configured by the programmer, has also been considered as an accelerator for the FDTD method [114, 57, 58, 59]. The FPGA is a Field Programmable Gate array, a hardware board that allows the software configuration of hardware directed at performing mathematical algorithms. The FPGA are configured as accelerator boards to the host computer, usually via a PCI bus. Similar to the manner in which the GPU is used as an accelerator, the FPGA implemented FDTD process experiences data communication bottleneck problems [122] when communicating with the host, when performing FDTD operations distributed across the host and the accelerator. These devices have a large cost of development and are also relatively complex to program. Although speed-ups of up to 22 and 97

Stellenbosch University http://scholar.sun.ac.za

times the speed of a conventional PC have been recorded [57], these are all for very small FDTD datasets in the order of several thousand grid points [59].

Contemporary processors such as the Haswell by Intel, or Pile Driver by AMD, have been customised for high performance numerical processing with the inclusion of customised hardware facilities such as the Fused Multiply Add (FMA) [141] and the Advanced Vector eXtensions (AVX) [99] on the processing die. In the case of the AVX, FMA, or SSE, these devices are programmed in a data parallel sense.

2.5 Summary

A review of the deployment of the FDTD method on high performance computers is not a review of the FDTD technology alone. The cost-performance advent of HPC since the start of the 21st Century allows the FDTD to be considered as a solver for large electromagnetic problems. FDTD solvers that used to run for several days a decade ago now run in just a fraction of that time. FDTD problem sizes that used to be considered phenomenally large, such as processing model sizes in billions of cells, are now being considered as mainstream. Compared to the frequency domain solvers such as FEM and the MOM method, the FDTD has been considered as a "brute force" approach [116] to modelling electromagnetic scenarios, and although this still holds true, the low cost/performance aspect of HPC will allow the FDTD to be deployed with less consideration for the computing cost and time.

In this chapter, the parallel FDTD method for electromagnetics has been reviewed according to the basic architecture that it has been deployed on. Although these were well defined architectures ten years ago, the advent of the new processing platforms such as the multicore processor, GPU, and customised processor functionality (such as the SSE) allow the architectures described in this chapter to be combined into hybrid processing platforms, such as the Accelerated Processing Unit (APU) and the Cluster designated Blade processors. Although it may be convenient to express the different parallel deployment styles as task parallel or data parallel, contemporary parallel FDTD implementations are a mix of both data and task parallelism.

 Stellenbosch University http://scholar.sun.ac.za

Chapter 3

3 The FDTD algorithm

3.1 Overview
3.2 Maxwell‘s equations
3.3 The FDTD modelling scenarios
3.4 Cell sizing, algorithmic stability
3.5 Boundary conditions
3.6 EM sources and reflectors
3.7 Validation of the FDTD code
3.8 Computing languages and APIs
3.9 Performance profiling the serial FDTD
3.10 Estimate of the parallel FDTD coding effort
3.11 Summary

3.1 Overview

There are many different techniques and methods to calculate or model the electromagnetic (EM) field. For the full-wave modelling of electromagnetic problems, there are three primary techniques, all of which use Maxwell’s equations to determine the behaviour of the electromagnetic field [7, 108]. These are:

1) The FDTD method, based on a finite differences approximation of Maxwell’s equations. The computation is performed on a regular lattice of electric grid values offset by one half a grid spacing from the magnetic grid values. The FDTD is a time domain based CEM solver.
2) The Finite Element Method (FEM) is a computational method used to find approximate solutions to differential equations. Unlike the FDTD which requires a rigorous rectangular computational cell structure, the FEM method can accommodate a variety of differently shaped gridding base structures.
3) The MOM (or Boundary Element Method) is a system for solving Maxwell’s equations in integral form; usually as a boundary integral [2]. It requires the calculation of boundary values only and does not need to compute the entire modelling space as with the FDTD. The MOM method usually operates in the frequency domain.

In the context of an EM simulation, these methods are part of a solver or computational stage of the simulation as was identified in the foundation days of the FDTD method [90]; that is, the electromagnetic modelling process requires that the following three actions be performed:

1) The input to the simulation requires the establishment of a modelling scenario indicating the dimensions of the object to be modelled, and the configuration of details such as conductivity, susceptibility, establishment of boundary conditions, and electromagnetic sources.
2) The computation of the modelling scenario or solution, also known as the solver or computational engine. A major thrust of this work is the comparison of the deployment effort using FDTD as the model solver on a variety of High Performance platforms.

 Stellenbosch University http://scholar.sun.ac.za

3) Visualisation of the output from the solver. A vitally important aspect from the functional sense in that it is required to visualise the massive numerical output from the solver and also because it can generate a large amount of computing load.

3.2 Maxwell's equations

The two Maxwell curl equations describe the physical behaviour of electromagnetic waves passing through free space and their interaction with physical matter [7]. Although the principle equations used in this work have been formulated by Maxwell, it should be noted that other contributors of that period, such as Faraday, Ampere, Gauss, Coulomb, and Lorentz, [145] established the foundations for the formulation of Maxwell's equations.

Maxwell's fundamental curl equations [3] describing the behaviour of electric and magnetic waves in an isotropic, nondispersive, and lossy material are:

$$\frac{\partial H}{\partial t} = -\frac{1}{\mu}\nabla \times E - \frac{1}{\mu}(M_{source} + \sigma^{*}H) \qquad 3.1$$

$$\frac{\partial E}{\partial t} = \frac{1}{\varepsilon}\nabla \times H - \frac{1}{\varepsilon}(J_{source} + \sigma E) \qquad 3.2$$

where:

J_{source} and M_{source} are independent sources of electric and magnetic energy

σ is the electrical conductivity

σ* is the equivalent magnetic loss

μ is the magnetic permeability

ε is the electrical permittivity

t is time

For the FDTD implementations undertaken in this thesis the coefficients σ, σ*, μ, and ε, are assumed to have constant values. Lossy dielectrics may be described over a band of frequencies [156, 157] and the representation of these coefficients by a complex term may contribute additional computational load.

To devise a solution for these equations using the FDTD method, a model structure is discretised over a three dimensional computational volume [1, 47]. The continuous partial differential equations of 3.1 and 3.2 are calculated using a centred finite differences approximation for the time and space variation [3]. A spatial convention for the formulation of these approximations is made according to the schematic cell structure shown in figure 3.1. The entire computational volume is comprised of these cells.

 Stellenbosch University http://scholar.sun.ac.za

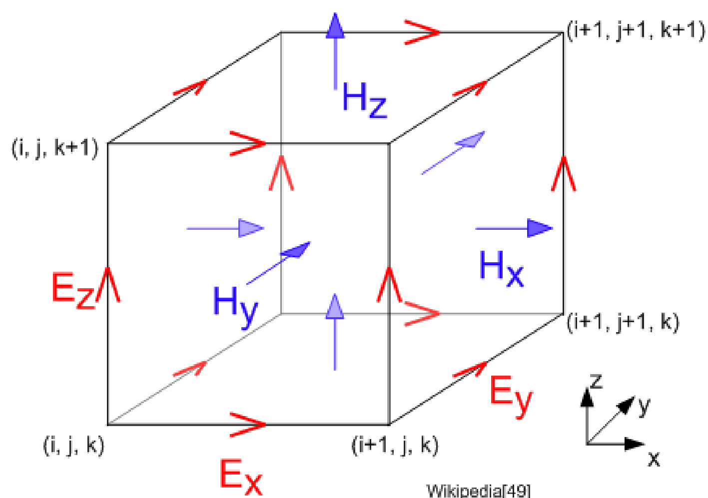

Figure 3.1: Magnetic and electric cell conventions in the Yee cell [Reproduced from Wikipedia].

It is by convention that the electric and magnetic field components are offset by one half of a cell spacing as shown in the two dimensional arrangement of electric and magnetic vectors of the TE field in figure 3.2 below. The electric field values are in the plane of the page and the magnetic field vectors are pointing into the page:

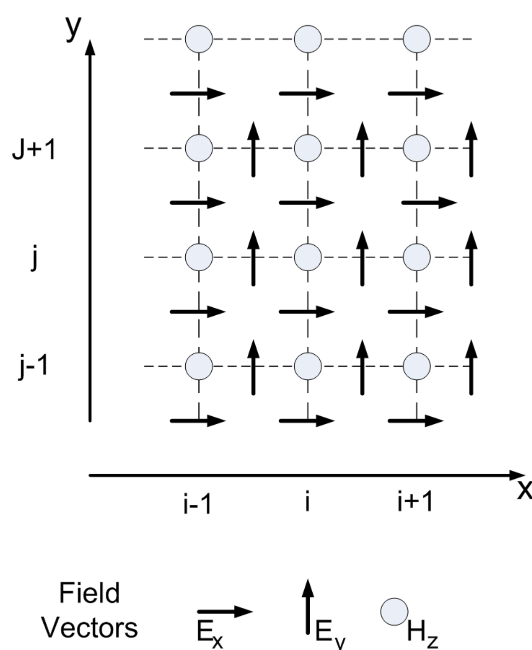

Figure 3.2: The Half-cell offset of magnetic and electric cells in the Yee lattice.

Using the coordinate convention as shown in the figures 3.1 and 3.2, the electric and magnetic field equations 3.1 and 3.2 can be expressed as a system of three dimensional finite difference equations [3] as:

 Stellenbosch University http://scholar.sun.ac.za

$$H_{x(i,j,k)}^{n+\frac{1}{2}} = D_{ax(i,j,k)} H_{x(i,j,k)}^{n-\frac{1}{2}} + D_{bx(i,j,k)} \left(E_{y(i,j,k)}^{n} - E_{y(i,j,k-1)}^{n} \right) - D_{bx(i,j,k)} \left(E_{z(i,j,k)}^{n} - E_{z(i,j-1,k)}^{n} \right) \quad 3.3a$$

$$H_{y(i,j,k)}^{n+\frac{1}{2}} = D_{ay(i,j,k)} H_{y(i,j,k)}^{n-\frac{1}{2}} + D_{by(i,j,k)} \left(E_{z(i,j,k)}^{n} - E_{y(i-1,j,k)}^{n} \right) - D_{by(i,j,k)} \left(E_{x(i,j,k)}^{n} - E_{x(i,j,k-1)}^{n} \right) \quad 3.3b$$

$$H_{z(i,j,k)}^{n+\frac{1}{2}} = D_{az(i,j,k)} H_{z(i,j,k)}^{n-\frac{1}{2}} + D_{bz(i,j,k)} \left(E_{x(i,j,k)}^{n} - E_{x(i,j-1,k)}^{n} \right) - D_{bz(i,j,k)} \left(E_{y(i,j,k)}^{n} - E_{y(i-1,j,k)}^{n} \right) \quad 3.3c$$

$$E_{x(i,j,k)}^{n+1} = C_{ax(i,j,k)} E_{x(i,j,k)}^{n} + C_{bx(i,j,k)} \left(H_{z(i,j+1,k)}^{n+\frac{1}{2}} - H_{z(i,j,k)}^{n+\frac{1}{2}} \right) - C_{bx(i,j,k)} \left(H_{y(i,j,k+1)}^{n+\frac{1}{2}} - H_{y(i,j,k)}^{n+\frac{1}{2}} - J_{source(x)}^{n} \right) \quad 3.3d$$

$$E_{y(i,j,k)}^{n+1} = C_{ay(i,j,k)} E_{y(i,j,k)}^{n} + C_{by(i,j,k)} \left(H_{x(i,j,k+1)}^{n+\frac{1}{2}} - H_{x(i,j,k)}^{n+\frac{1}{2}} \right) - C_{by(i,j,k)} \left(H_{z(i+1,j,k)}^{n+\frac{1}{2}} - H_{z(i,j,k)}^{n+\frac{1}{2}} - J_{source(y)}^{n} \right) \quad 3.3e$$

$$E_{z(i,j,k)}^{n+1} = C_{az(i,j,k)} E_{z(i,j,k)}^{n} + C_{bz(i,j,k)} \left(H_{y(i+1,j,k)}^{n+\frac{1}{2}} - H_{y(i,j,k)}^{n+\frac{1}{2}} \right) - C_{bz(i,j,k)} \left(H_{x(i,j+1,k)}^{n+\frac{1}{2}} - H_{x(i,j,k)}^{n+\frac{1}{2}} - J_{source(z)}^{n} \right) \quad 3.3f$$

where the coefficients C and D are:

$$C_{a(i,j,k)} = \left(1 - \frac{\sigma_{(i,j,k)} \Delta t}{2\varepsilon_{(i,j,k)}} \right) / \left(1 + \frac{\sigma_{(i,j,k)} \Delta t}{2\varepsilon_{(i,j,k)}} \right) \quad 3.4a$$

$$C_{b(i,j,k)} = \left(\frac{\Delta t}{\varepsilon_{(i,j,k)} \Delta} \right) / \left(1 + \frac{\sigma_{(i,j,k)} \Delta t}{2\varepsilon_{(i,j,k)}} \right) \quad 3.4b$$

$$D_{a(i,j,k)} = \left(1 - \frac{\sigma_{\{i,j,k)}^{*} \Delta t}{2\mu_{(i,j,k)}} \right) / \left(1 + \frac{\sigma_{\{i,j,k)}^{*} \Delta t}{2\mu_{(i,j,k)}} \right) \quad 3.4c$$

$$D_{b(i,j,k)} = \left(\frac{\Delta t}{\mu_{(i,j,k)} \Delta} \right) / \left(1 + \frac{\sigma_{\{i,j,k)}^{*} \Delta t}{2\mu_{(i,j,k)}} \right) \quad 3.4d$$

 Stellenbosch University http://scholar.sun.ac.za

The formulations comprising equation 3.3 can then be easily encoded into higher level computing languages such as C, Python, or Fortran.

3.3 The FDTD modelling scenarios

The serial FDTD has been converted into parallel form on several different high performance platforms for this work. The following two electromagnetic scenarios were used to compare the deployment effort between the different platforms. This was done not only to keep the amount of coding effort used in converting from one platform to another approximately the same, but also to validate the correctness of the serial and parallel FDTD implementations.

3.3.1 Scenario 1

The two dimensional model shown in figure 3.3 below is of a rectangular area with a wire as a source, shown as a small circle on the left hand side, passing through the area perpendicular to the page. A conductive solid circular cylinder, also perpendicular to the page, has been modelled on the right hand side of the rectangular cavity. The rectangular shaped area is surrounded by an absorbing boundary, which is not shown. Various shaped current waveforms such as a Gaussian pulse, or a continuous sine wave, are transmitted by the source wire. The rectangular cross is a fictitious template indicating which data values are involved in the calculation of a single Yee cell, also referred to as a computational stencil. Each dot in the area shown in figure 3.3 is the centre of an FDTD cell.

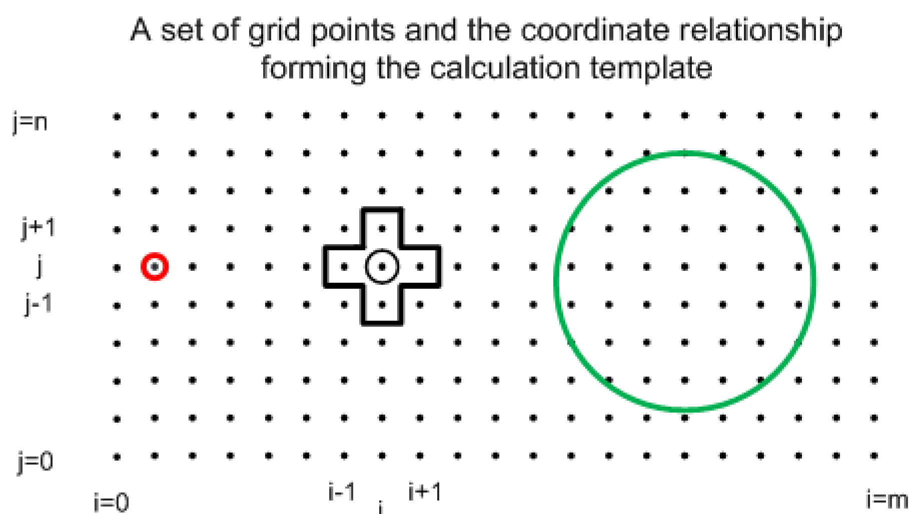

Figure 3.3: A schematic representation of the model components used for the 2D FDTD modelling.

3.3.2 Scenario 2

For the three dimensional form of the FDTD, the 3D modelling space consisted of a cubic free space cavity with a point source or dipole transmitter at the centre of the cube. The cube is surrounded by an absorbing boundary area. The source is modelled using a continuous sine wave or Gaussian pulse.

 Stellenbosch University http://scholar.sun.ac.za

3.4 Cell sizing, algorithmic stability

The theoretical speed of an electromagnetic wave in the FDTD domain should not exceed the speed of light in order to maintain computational stability when using the FDTD method. If the EM wave is propagated in computational space faster than the speed of light, or conversely, the computed spatial advance of the EM wave exceeds that of the grid spacing, then the algorithm will become unstable. In programmatic terms this will result in nonsensical data being created, or a program crash. Some authors have proposed formulations permitting the Courant limit to be exceeded [105] although this does come with restrictions such as being able to calculate solutions for only one frequency.

Instability is avoided by adhering to the Courant condition [2, 3], which can be expressed as equation 3.6; assuming $\Delta x = \Delta y = \Delta z$:

$$\Delta \mathrm{t} \le \frac{\Delta x}{\sqrt{n}.u} \quad 3.6$$

where n is the number of the dimension and u is the velocity of propagation of the EM wave. In practical terms it is best to operate the cell spacing at just underneath the Courant limit in order to guarantee computational stability but limit dispersion. Although operating at a small time increment may ensure stability, it may also lead to an increase in numerical dispersion [2]. Even though it may be possible to operate exactly on the Courant limit, a minute increase over the Courant limit will lead to numerical instability. There is also a risk of numerical instability introduced by floating point truncation or rounding errors when operating exactly on the Courant limit.

The Courant limit may also be calculated using the formula 3.7 below, where c is the speed of light in free space, and the Δx, Δy, and Δz terms are the dimension of the Yee cell used for the modelling.

$$\Delta \mathrm{t} \le \frac{1}{c} \sqrt{\left[\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2} + \frac{1}{\Delta z^2}\right]} \quad 3.7$$

3.5 Boundary conditions

To simulate an infinite or extended computational space while still limiting the computational domain, boundary conditions need to be used to absorb the electromagnetic radiation.

The creation and the computational management of the boundary conditions are a critical part of the serial and parallel FDTD simulation, owing to the physical significance and computational complexity of what happens when the FDTD computation reaches the edge of the calculation grid and boundary space. The computation of the FDTD cannot be simply terminated at the edges as this will create computational artefacts owing to the incorporation of invalid values in the FDTD computational stencil.

 Stellenbosch University http://scholar.sun.ac.za

If the requirement should be to simulate a space larger than that defined by the Yee lattice, then the boundary conditions are required to absorb the electromagnetic radiation. The Absorbing Boundary Conditions (ABC) conditions, initially defined by Mur [106], perform well with absorbing radiation perpendicular to the boundary [2], but are less efficient when the radiation is oblique to the boundary. The creation of boundary conditions by Berenger have resulted in the formulation of the Perfectly Matched Layer (PML) and have culminated in a mathematical mechanism that is better suited to absorb radiation oblique to the intersection between the main computational lattice and the boundary lattice [107]. For this thesis the FDTD boundary conditions were implemented using the original split-field Berenger PML [107]. Each electromagnetic vector field is split into two orthogonal components. By choosing loss parameters consistent with a dispersionless medium, a perfectly matched interface is created. The FDTD update equations in the PML boundary area need to be modified to accommodate the new formulation and associated parameter values [2, 3].

From a computational aspect, the boundary condition can influence the parallel FDTD computation in the following way:

1) Compared to the processing load of the main computational lattice the PML boundaries require minimal processing load, as shown by the performance profile in table 3.2. The PML boundaries require regular data synchronisation with the main grid for each iteration of FDTD processing.
2) The coding of the PML boundary conditions using data parallelisation techniques is not simple. The PML boundaries are calculations that are made in their own computational lattice and share data from the main computational lattice. The flattening of the computational arrays, as required for the GPGPU processing make this boundary and main lattice alignment particularly prone to coding errors. It is critical to have a good data visualisation tool to assist with the debugging when developing this form of software.

From the aspect of coding effort, the implementation of the split field PML was undertaken on all of the computing platforms to gauge the relative effort required to implement the same algorithm on every high performance platform. The coding effort for the implementation of the split field PML in the task parallel sense with threading methods such as MPI

3.6 EM Sources and reflectors

The FDTD modelling process requires that the EM source is defined in terms of the following attributes:

1) A physical size and shape approximation in space: in a manner very similar to the approximation of the reflectors, or other artefacts in the modelling space, the physical dimensions of the source are required to approximate the actual physical dimensions and properties as closely as possible. The physical size of the FDTD cell must be conditioned for the size of the source or reflector model so that the model can be represented accurately. The approximation of a cylindrical source with a circular cross section by a rectangular FDTD cell structures will lead to stair-casing, or sizing, discrepancies in the model surface, which can in turn lead to a wide variation in modelled responses [20, 12].

 Stellenbosch University http://scholar.sun.ac.za

2) Maintaining control of the physical location of the source is important to some forms of the parallel FDTD, such as those implemented in individual memory spaces using MPI. The source must be available to a continuous modelling space even when the modelling space has been divided into a series of individual chunks. Using the MPI threading will require the correct positioning of the source structure in the appropriate program chunk or thread. The openMP model in contrast does not experience this problem as the excitation of the source occurs in a grid that is held in a single unified memory space and hence uses a single set of coordinate indices to identify grid points.
3) The nature of the excitation of the source. This can be either by using a Gaussian pulse or a sine wave as the input waveform shape, which in turn is dependent on the objectives of the simulation.

3.7 Validation of the FDTD code

In order to ensure that the FDTD algorithm as used in this thesis is functioning correctly, the electromagnetic far field of a vertical dipole antenna was calculated and the results compared to those from a commercial electromagnetic modelling product. The commercial product used for the modelling is based on the Method of Moments and is called Feko [137]. Results of the FDTD modelling of the far field and the comparison to the commercial results are available in Appendix A.

3.8 Computing Languages and APIs

When comparing the parallelised form of the FDTD on different computing platforms one needs to examine the suitability of the chosen language for the purpose of parallelisation and the portability of that language to other computing platforms. It is common practice for software developers to reuse computational frameworks and templates to create new computational solutions from existing ones. The implication of this is that the experience gained by a large group of users of that language can be transferred from other numerical discipline to the coding of the FDTD. A specific example of this is the performance improvement achieved for the encoding of video using SSE when used to enhance the run-time of the FDTD. Code fragments used to enhance the performance of video encoding were adapted to the FDTD algorithm to speed up processing of the FDTD, as will be described in section 5.6.

Although the serial form of the FDTD in Computational Electromagnetics (CEM) was originally coded in Fortran [67], some early mention is made of implementations in C, such as a description on a Transputer array [22], an early form of RISC based High Performance Architecture. The parallel FDTD has been implemented on several different hardware platforms with a variety of computing languages. However, applications written in C and C++ are portable across most architecture, allow direct access to low level device architectures such as GPUs, and provide ample coding frameworks from other disciplines. Python, a higher level scripting language allows parallel FDTD functionality [68] on several platforms and provides easy access to high level functionality such as visualisation products. Although customised language elements need to be created for Python to access to specific hardware devices, such as GPUs, the language definition provides advantages over C++ by

 Stellenbosch University http://scholar.sun.ac.za

providing much simpler access to mathematical definitions such as complex numbers and arrays. Python also provides integrated access to high end visualisation methods required to view the resultant output from the FDTD.

The FDTD has been implemented on the Compute Unified Device Architecture (CUDA) supported GPGPU quite commonly but recent publications listing FDTD deployments with openCL are becoming more frequent. For this work an FDTD deployment was created with the Direct Compute API from Microsoft, as will be described in section 5.9.

A summary of these different GPGPU interfaces and their applicability to the FDTD is given below:

1) CUDA is the innovator in the development of numerical algorithms for GPGPUs, and as such, has a competitive advantage over the number of systems deployed over other interfaces such as open CL or Direct Compute.

   The advantages are:

   - Fairly simple API
   - Mature numerical library set such as Cublas, CUFFT
   - First mover deployment in numerical algorithm market
   - Bindings to many languages
   - Excellent documentation
   - Large user community
   - Good language toolset such as profilers
   - Regular performance and software updates

   Disadvantages of CUDA are:

   - For Nvidia GPUs only
   - Long learning curve

2) openCL also supports the implementation of FDTD code on GPGPU hardware [111, 112] provided by different manufacturers, yet is much later in arriving on the scene than CUDA. Although the fundamentals of the CUDA interface and openCL are quite similar, the openCL is driven by C++.

   The advantages of openCL are:

   - Wide range of hardware supported. Not only Nvidia
   - Same code can operate on both GPU or CPU
   - Can share resources with openGL
   - Can produce a single code set for the APU

 Stellenbosch University http://scholar.sun.ac.za

Disadvantages of openCL are:

- Not well supported by all manufacturers
- Lacks mature libraries and code examples/templates
- Very few language bindings
- Limited documentation

3) Direct Compute is an interface provided by Microsoft and works in conjunction with the DirectX11 product suite on Windows platforms [113]. The only known FDTD implementation using the Direct Compute Interface has been created for this work, although there are earlier implementations using the DirectX graphics pipeline.

   The advantages of Direct Compute are:

   - Works on GPUs from most vendors using the Windows platform. Well supported by manufacturers
   - Direct access to graphics of the DirectX APIs
   - Allows integration of single code set for both cores and GPU on APUs

   Disadvantages of Direct Compute are:

   - Very few examples for GPGPU
   - Complex to code
   - Few language bindings

Visual Basic is a Rapid Application Development (RAD) language designed by Microsoft for the rapid prototyping of computing models. Although not a very efficient language when considering numerical computations it is a versatile language that will allow the conversion of resultant FDTD output into graphical formats and assist with the creation of FDTD models [66]. Visual Basic is very useful to create numerical utilities, such as those that can compare three dimensional grids or images. For this work the Visual Basic language was used to create utilities to compare data sets during debugging exercises and also at times to do a quick visualisation of the FDTD results. In particular it provided a tool to stitch together segments of images output from the parallel FDTD deployments. The image composites were then numerically compared to the image results from the serial FDTD. Visual Basic is also used as a scripting tool in some commercial electromagnetic modelling products such as CST Microwave Studio.

Matlab is a higher level language that provides an integrated development, processing and visualisation environment under the control of a customised operating framework. It is useful for the development of FDTD-like numerical algorithms that can only be conveniently developed and debugged with the use of some form of visual aid. Matlab makes use of a command interpreter that has language elements specifically created for parallel or high performance functionality.

For most of the work described in this document algorithms have been coded in C or C++ as this allows the portability of code between different platforms. It was also very convenient to implement

 Stellenbosch University http://scholar.sun.ac.za

compiler intrinsics in plain C as the documentation was geared towards this. A good example of this is the implementation of the FDTD with SSE. Early implementations of the with SSE in x86 assembly language embedded in the FDTD code were inefficient compared to implementing the FDTD with SSE using compiler intrinsics. It is also simpler and quicker to code with compiler intrinsics than with assembly language.

3.9 Performance profiling the serial FDTD

In order to convert the serial FDTD into a parallel FDTD it is necessary understand precisely which part of the FDTD code was consuming the largest amount of processing time.

The serial FDTD process is made up of three discrete computational stages:

1) A serial processing stage consisting of a model preparation stage and other serial functionality, such as process control and status messages. The preparation stage calculates the initial FDTD modelling parameters and assigns the configuration of the calculation lattice to memory.
2) The computation stage consists of the repeated iterations of the FDTD stencil over the calculation lattice.
3) The output functions which save the resultant FDTD computations to disc for subsequent visual post processing.

Although a visual inspection of the existing FDTD programs indicated that the bulk of the processing effort would be taken up by the processing of the finite difference equations (equations 3.3 a-f) this was confirmed by using a code profiling tool and also by manually profiling the FDTD code using timing markers. The results of the profiling of the 3D FDTD code with a profiling tool are shown in table 3.1 below. The function “Compute”, shown in table 3.1, is a function that will calculate the FDTD finite difference equations and the FDTD PML boundary conditions and the profiler confirms initial expectations that these would take up the dominant part of the processing time. The processing time of the other components of the FDTD program, although small, are still relevant when considered in the context of Amdahl’s law in described section 5.2. The consideration that needs to be made with regard to Amdahl’s law is that even if all of “compute” functionality in table 3.1 below could be so highly parallelised that it took up zero seconds, further speedup of the FDTD would be limited by the timings shown in the serial “non-parallelised” functions.

Name of 3D FDTD program function	Time – micro S	% Total Time
Compute – FDTD computations	126894.384	99.80%
Dipole – create source model	0.021	0.00%
setup – setup initial values	0.673	0.00%
Initialise – reserve memory	192.594	0.20%
main	0.006	0.00%

Table 3.1: Results of time profiling the constituent functions of the 3D FDTD with a profiling tool.

The results of manually profiling the 2D FDTD code correspond very much with the 3D FDTD, results for which are shown in table 3.2 below.

 Stellenbosch University http://scholar.sun.ac.za

Total run time of FDTD program	1520s	100%
Time consumed by FDTD difference equations	1499s	98.62%
Time consumed by calculating FDTD boundaries	20s	1.32%
Start-up time before FDTD	1s	0.06%

Table 3.2 The profiling of the 2D FDTD using manual timing marks.

3.10 Estimate of the parallel FDTD coding effort

The effort required to convert the serial FDTD program into parallel FDTD program on a specific platform is subject to factors such as the experience base of the personnel doing the deployment and the technical information available for that specific platform. These deployment factors listed below are specific to the platforms evaluated for this thesis and are summarised in table 3.3. The deployment factor has been ranked in the range from 1 to 10, in the sense that a lower score contributes towards less coding effort.

The coding effort is subject to the following:

1) The size of the user community for the target platform

   A larger community is seen as a positive. The size of a user community for a specific coding platform creates its own source of self-help and discussion. Good examples of these are the user forums for Intel, AMD and NVidia processors.
   User Rank: 1 Large community, 10 Small community.

2) Maturity of the technology

   Later versions of one type of technology have a development precedent and require less work to optimise. An example of this is the development of the SSE standard on the Intel and AMD chipset which was previously only accessible using inline assembly language. The subsequent release of SSE compiler intrinsics makes the task of building FDTD applications with SSE relatively simple. The number of releases a platform has evolved through is also an indication of maturity.
   Maturity Rank: 1 Mature, 10 Immature.

3) Availability of Computing Platform

   One of the aspects that makes the discrete GPU so appealing as an HPC platform is not only its price-performance aspect but also that it can be accessed at will from the convenience of one's PC. Sharing a resource such as a GPU cluster can be a difficult task as usage by others may make the development of programs a long process.
   Avail Rank: 1 High availability, 10 Low availability.

4) Coding precedent in the form of coding templates from other disciplines

 Stellenbosch University http://scholar.sun.ac.za

Coding examples from one computing discipline can be reused in another. An example of this is the use of the SSE to accelerate the FDTD process, as described in chapter 5. The SSE is quite commonly used in video compression algorithms and closer examination of how it is achieved yields some good examples of SSE usage that can be implemented for the FDTD.
Precedent Rank: 1 Good precedent, 10 No coding precedent.

5) Documentation

With regard to the information available for the deployment two examples are given that illustrate the effect this has when optimising the FDTD formulation. An example of good documentation is given by Nvidia's arsenal of clear examples and easy-to-read manuals, all geared towards the programmer being able to achieve his objectives on the GPGPU platform. The other end of the spectrum is the documentation for the Bluegene/P for which the availability of information may best be expressed by an example. This example is the Wikipedia definition of the IBM PowerPC 450 used in the Bluegene/P. The quote reads:
"The processing core of the Bluegene/P supercomputer designed and manufactured by IBM. It is very similar to the PowerPC 440 but few details are disclosed."
Doc Rank: 1 Good documentation, 10 Lack of documentation.

6) Orthodoxy

The parallel FDTD code threads implemented with MPI and openMP threading libraries are very similar in coding structure to the serial form of the FDTD. The code implemented for the GPU is a vector based data parallel code that can be difficult to follow when compared to the serial form of the FDTD, i.e. it is not structured in a conventional or orthodox coding style.
Ortho Rank: 1 Orthodox, 10 Not Orthodox.

Platform	User	Maturity	Available	Precedent	Doc	Ortho	Total
Bluegene/P	5	2	2	3	7	3	22
M9000	5	3	2	3	6	3	22
Cluster	1	2	4	1	5	3	16
GPGPU	3	6	1	4	5	8	27
Multicore	2	1	2	1	3	2	11

Table 3.3: Factors influencing the coding effort of the FDTD

Based on the deployment effort summarised in table 3.3 an FDTD like algorithm is most readily deployed on a multicore chip and its derivations, such as the cluster computer. A coding deployment of the FDTD on a GPU is much more involved than on a multicore architecture, and this is reflected as the GPU being ranked as the fifth, and most onerous deployment.

 Stellenbosch University http://scholar.sun.ac.za

3.11 Summary

The FDTD method is a finite differences approximation of Maxwell's equations and its highly repetitive algorithmic structure make it ideal for deployment on a High Performance Computing platform. Careful management of the stability and boundary conditions of the modelling domain allow the FDTD method to be used in one, two or three dimensional form.

The present FDTD implementation of the electromagnetic far field emanating from a dipole antenna corresponds well with an equivalent formulation using FEKO, a commercial product using a Method of Moments modelling approach.

The profiling of the serial FDTD code shows that the greatest computational load in the FDTD algorithm is being incurred by the update of the electric and magnetic field equations. This identifies the electric and magnetic field update equations as being good targets for parallelisation. The effort used to code the parallel FDTD has been estimated using factors such as maturity of technology, documentation available, and the conventional nature of the FDTD algorithm. From this it is determined that the multicore platform requires the least effort to code the parallel FDTD.

Stellenbosch University http://scholar.sun.ac.za

Chapter 4

4 Architecture of HPC systems

4.1 Overview
4.2 Hardware architectural considerations
4.2.1 The processing engine
4.2.2 Memory systems
4.2.3 Communication channels
4.2.4 Hyper-threading (HT)/Symmetric Multithreading (SMT)
4.3 Examples of HPC systems
4.3.1 Discrete GPUs
4.3.2 Accelerated programming units APUs
4.3.3 SUN M9000 shared memory processor
4.3.4 Intel multicore at the Manycore Test Lab
4.3.5 Nehalem and Westmere cluster computer
4.3.6 IBM's Bluegene/P
4.3.7 Intel's Many In Core (MIC) processor
4.4 Other considerations
4.4.1 Power consumption
4.5 Summary

4.1 Overview

This chapter describes the hardware characteristics of the High Performance Computing (HPC) systems used to deploy the parallel FDTD for this thesis. The architecture of each system is analysed from the perspective of the FDTD implementation.

Most numerical algorithms are very closely related to the hardware that they are deployed on. It is a maxim that hardware and software evolve together in iterative steps, as is reflected by the increase in computing capacity of the personal computer hardware and software since the early 1980s. This also applies specifically to the parallel FDTD which requires extremely large hardware capacity and computational power to make the operation of the FDTD viable [69].

An examination of the components of a high performance computer system is necessary to identify the elements required by the parallel FDTD and this is the objective of this chapter. To understand how these individual serial components are transformed from a traditional processing mechanism to a high performance processing device requires a decomposition of the computing devices. The reasons for this are:

1) To identify the relationship between HPC components and understand how these in turn relate to the implementation and performance of the FDTD. A typical example of this is the manner in which the FDTD performance is affected by correct memory coalescence of Graphical Processing Unit (GPU) threads [87].

 Stellenbosch University http://scholar.sun.ac.za

2) To analyse the evolution of architectures within the computing devices as this may have a profound effect on how the FDTD is programmed. An example is the adaptation of the FDTD to using the Streaming SIMD Extensions (SSE) or Advanced Vector eXtensions (AVX) in a processor and describing the implication of how the change in the AVX will affect the processing performance of the FDTD. This will be discussed in some detail in chapter 5.

4.2 Hardware architectural considerations

In the context of the parallel FDTD and High Performance computing, the computer hardware is made up of three main functional components:

1) The processing engine/compute engine.
2) The memory associated with the processing engine.
3) The communication channels connecting memory and processors elements.

An analysis of the computing hardware used for the parallel FDTD would not be complete without some description of the basic building blocks that make up the platform. The FDTD literature is strongly influenced by marketing forces, and the description of features such as GPU processing power can be misleading where the implementation of the FDTD are concerned. The provision of many cores for the processing of graphical algorithms may be very appealing as a first impression, but for the FDTD one needs to consider:

1) That these GPU processing cores have a reduced instruction set when compared to the conventional CISC instruction, and hence a reduced capability.
2) Many cores require a sufficient supply of data to process at full capacity and this requires a consideration to be made for adequate memory bandwidth and latency.

This is not meant to put the GPU in a disparaging or contentious light but serves as an example that a simple comparison of numbers describing the hardware characteristics tells only part of the processing performance story. It is also not enough to generalise about the characteristics of the HPC platform at large, such as GFLOP or other speed ratings, as such a performance rating will change from one type of algorithm to another [83]. As an example consider two different algorithms being processed on the GPU, an algorithm such as Monte-Carlo simulation requires no communication between the processing cores, or a constant supply of data from memory to the cores, whereas the FDTD requires both [97].

It is also relevant to the FDTD that HPC hardware is constantly in a state of improvement. The inclusion of Fused Multiply Add (FMA) [142] and Streaming SIMD Extensions (SSE) facilities in conventional Complex Instruction Set Computing (CISC) processors show the ability of processor manufacturers to adapt the hardware design to HPC requirements. The development of computing hardware is now in a phase where one can no longer identify simplistic architectures, such as the single serial compute cores used by older Beowulf cluster nodes, which have been replaced by multicore processors in contemporary versions. There is a similarity in the architectural structure of the larger HPC systems such as the Bluegene/P, SUN M9000 and the cluster computers in that the compute codes are made up of multiple closely coupled multicore processors, as shown in table 4.0.

Stellenbosch University http://scholar.sun.ac.za

System	Compute Node description	Number of Processors on Compute Node	Total number of physical cores in 1 Compute Node	Inter-node Connection method
M9000	CMU	4	16	Interconnect (SUN)
Bluegene/P	Node Card	32	128	Interconnect (IBM)
SUN Nehalem	Blade 6275	4	16	Infiniband

Table 4.0: The use of Compute Nodes in large HPC computers

The implication of this similarity is that in order for the FDTD to achieve good performance on an HPC system, good data bandwidth and low latency data communication is required between the multicore processing cores and the associated memory. The memory latency is the time required by the processor to access a section of data for reading or writing. It is a measure of how quickly the communication and memory medium is required to complete a data access request. Communication latency can be measured in a variety of ways, such as with a Ping-pong program [98], or according to a method described as IEEE RFC2544 [98]. Figure 4.0 shows which components of the communications stack [123] are involved in calculating the latency for an FDTD data exchange. The measurement of latency will depend on several sub-systems in the Open System Interconnect (OSI) [123] communication stack as is shown in figure 4.0.

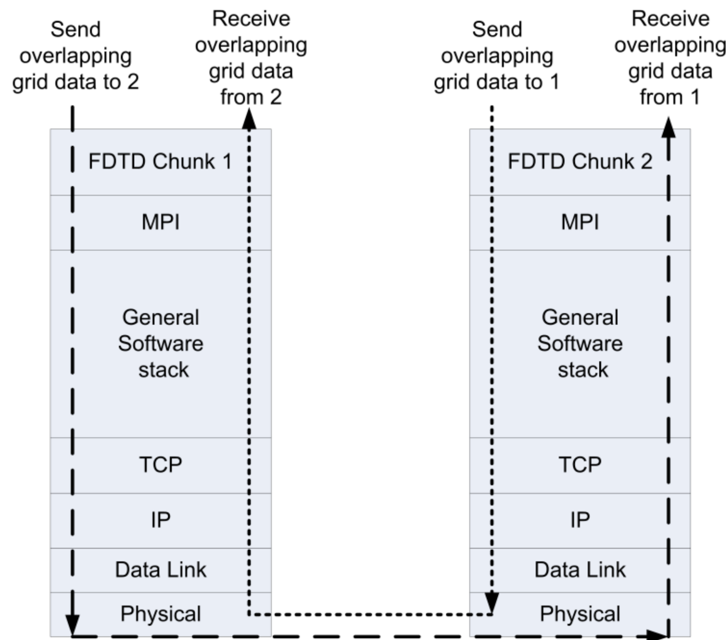

Figure 4.0 Communication latency experienced by two FDTD chunks as the data passes through the OSI stack using an ethernet connection.

Short communication and memory latency between processors needs to be one of the central requirements when considering the hardware to process the parallel FDTD. A consideration to be made when analysing the hardware is not just to examine components individually but also how they communicate. One of the objectives of this thesis is to distinguish between the different platforms that support the FDTD, and the examination of processor interconnectivity is one of these

 Stellenbosch University http://scholar.sun.ac.za

aspects. The comparison is not simple as communication functionality provided by the close collaboration of the processing components in devices such as an Accelerated Processing Unit (APU), cannot be expected to be reproduced for larger systems using discrete processor and memory interconnects.

By examining the memory to processor latency, and processor to processor communication latency, the latency can restrict the processing performance achievable by the parallel FDTD algorithm, as is schematically shown in figure 4.1. Although the length of the arrows showing the elapsed time of the computation and latency sections are not proportional to actual timings, they do serve to illustrate that for the FDTD the latency penalty is incurred during a synchronised event between the computational stages on different processing threads. Therefore, if the latency can be reduced, then the efficiency of the process can be improved. There are two main categories of latency that have an effect on FDTD performance. One is the latency inherent in inter-process communication, as is shown in the ping pong test schematic of figure 4.0 above. The other source of latency is the time required to communicate with physical devices, such as memory components.

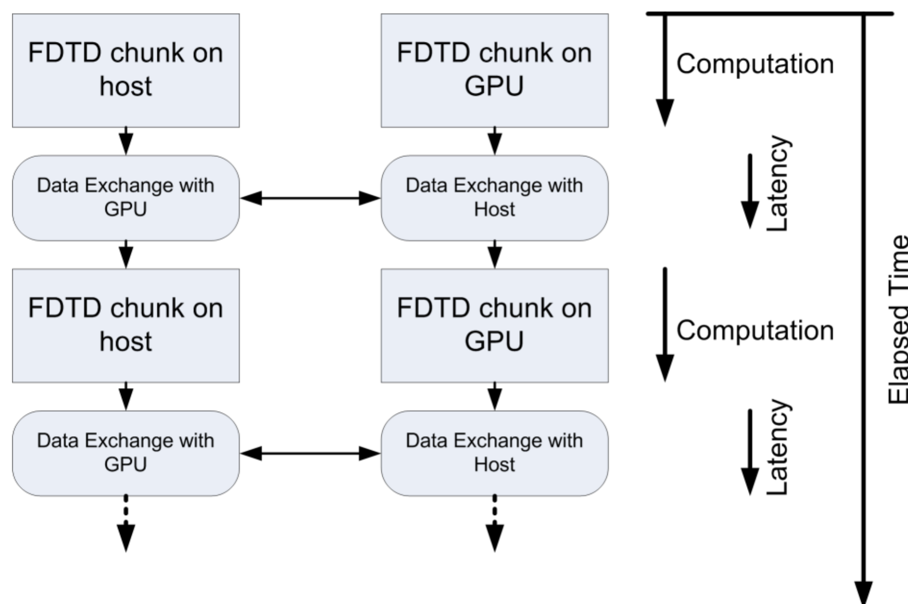

Figure 4.1: Latency in the parallel FDTD process.

4.2.1 The Processing engine/ the Compute engine

The processing engines used for the parallelisation of the FDTD use processor architectures varying from the Reduced Instruction Set Computing (RISC) based GPU core to the CISC based Nehalem Xeon 5570 multicore CPU.

There is a trend in contemporary processors to incorporate more high performance computing elements on the processor die. An example of this is the incorporation and extension of the vector processing capability provided by SSE and AVX [99] and specific arithmetic logic such as the Fused Multiply Add (FMA) in the "Ivory Bridge" series of processors by Intel. Multicore CPUs form the basic building block of most supercomputers as is shown in table 4.1.

Stellenbosch University http://scholar.sun.ac.za

Configuration	Type of Multicore IC	Number of cores	Release Year
Beowulf Cluster	Intel E2140	2	2007
Bluegene/P	PowerPC 450	4	2007
SUN M9000	Sparc 64 VII	4	2009
Nehalem Cluster	Intel Xeon 6550	8	2010
Cray XK6	AMD Interlagos	16	2012
Intel	Knights Corner/Intel Phi	50	2013

Table 4.1: The increase in the number of physical cores on a processor with time.

The table shows the number of cores on a processor die is increasing with time. If one follows the trend shown in table 4.1 then Intel's Knight's Corner would potentially be a good candidate as a building block for a supercomputer, as it has 50 processing cores. Supercomputers such as the Cray XK6 and the Chinese Tianhe A1 project are using processing engines made up of a hybrid configuration of multicore and GPU processing engines. The GPU processing engines are configured as accelerators to the multicore processing engines.

4.2.2 Memory systems

The memory available for FDTD processing can be classified according to its speed of access and capacity. Memory performance is generally defined in the following terms:

1) Latency; the delay in reading and writing the memory by the processor.
2) Bandwidth; this is the rate at which data can be written to or from a memory IC by the processor. The maximum theoretical bandwidth is calculated by multiplying the memory bus width by the frequency it can transfer data at. This maximum theoretical throughput cannot usually be sustained and is only an indication of peak bandwidth.
3) Memory channels; in order to alleviate the access of memory via one memory channel, several memory channels may be used. This has been common use in server technology for some time and since 2007 has been available on off-the-shelf motherboards. Although the memory is accessed via one memory controller, several channels of memory can be used. The Blade servers used in the CHPC's cluster computers provide three physical channels of memory to each CPU.

In this thesis we make use of the following types of memory:

4.2.2.1 RAM/DRAM/SDRAM/DDR/GDDR/SRAM

The term Random Access Memory (RAM) is used quite generically in that it describes most types of memory where the data may be accessed for reading and writing in any order. Dynamic Random Access Memory (DRAM) supplies the data to the system bus in an asynchronous sense. In physical terms the DRAM is RAM memory that stores each individual bit in a capacitor, which requires constant refreshing as the charge dissipates (leaks).

Stellenbosch University http://scholar.sun.ac.za

Synchronous Dynamic Random Access memory (SDRAM) is DRAM that is synchronised with the system bus and affords lower memory latency than DRAM. The clock synchronisation allows the more efficient use of pipelining instructions and data and hence exhibits lower memory access latency than DRAM.

The term DDR (Double Data Rate) is used in conjunction with SDRAM to identify a memory that operates at twice the data rate of the conventional SDRAM, and has twice the bandwidth. DDR3 and DDR4 are higher speed versions of the DDR memory.

The Graphic DDR (GDDR) is a type of DDR memory adapted for graphics processing in that it provides better memory bandwidth to the processing unit. The GDDR5 uses less power than DDR3 and will allow sections of the memory to be assigned to different physical memory channels, and hence specific processing cores within a multicore, Many In Core (MIC), or graphics processor.

4.2.2.2 Cache memory

This is used as an intermediate memory staging area in moving data to and from the processing cores. It reduces the average time to access the DRAM by storing frequently used copies of data in closer functional proximity to the core. Cache memory has much lower access latency than DRAM and a rule of thumb for game programmers holds that the access latency of cache memory is nearly $1/10^{th}$ of DRAM [33]. There is a sub-classification of cache memory, as data is cached in several levels of cache, i.e. Level 1, Level 2, etc. Access to the cache memory is usually controlled by the operating system.

The programming overhead and skill required to use the cache memory levels programmatically is much larger than the use of programming data access to DRAM. Cache memory can be very useful when alleviating the data bandwidth and latency bottleneck while transferring FDTD grid data to and from the processing core [100]. The use of the multilevel cache mechanism can be used to hide the longer latency incurred by accessing the DRAM.

The cache is Static RAM (SRAM), a RAM based on the switching of Integrated Circuit logic. SRAM is different from DRAM in that it does not need to be refreshed at every clock cycle. The use of SRAM is commonly associated with the use of Field Programmable Gate Arrays (FPGA) as this type of memory is used to store the FPGA's configuration data. SRAM has lower access latency than DRAM, it consumes less power, but is more expensive. A typical processor's cache memory is usually made up of SRAM. In GPUs, cache memory is also referred to as shared memory.

The cache system used by a multicore processor needs to be cache coherent in order to achieve good performance. That is, some data may be available as several copies in different caches on a multicore processor. Changes to one copy in one cache need to be propagated to other caches and this is referred to as cache coherence. The cache coherence should not disrupt the process flow.

Stellenbosch University http://scholar.sun.ac.za

4.2.3 Communication channels

The technologies that provide the communication channels between processors can operate at a variety of speeds as listed in table 4.2. The entry of the DRAM at the bottom of the table is a reference value to compare the latency and bandwidth achieved by the communication of the processor directly with the memory.

Network	Type	Latency	Bandwidth
Ethernet [76, 77]	1 Gbit	30-125 µs	1 Gbit/sec
Ethernet	10 Gbit	5-50 µs	10 Gbit/sec
Infiniband [75, 77]	QDR	3-15 µs	2.5 GByte/sec
Infiniband [Mellanox]	FDR	1 µs	2.5 GByte/sec
Interconnect	Gemini (Cray)	105-700 ns	8.3 GByte/sec [74]
Interconnect	Jupiter (SUN)	258-498 ns	737 GByte/sec
Interconnect	Torus (IBM)	3-7 µs	380 MByte/sec
Integrated Interconnect	QPI (Intel)	40-60 ns*	32 GByte/sec
Memory [5]	DRAM	3-10 ns	3-10 GByte/sec

*processor dependent

Table 4.2: A comparison of the latency values for some memory and communication systems.

The network topology is a description of how the processors in a multiprocessor system are related [75]. The topology can be described in a physical and logical sense. The manner in which processors are spatially related is the physical topology. The logical topology is the relationship of processors as defined by the parallel FDTD program. As an example, two parallel FDTD program threads communicating data between each other are considered to be neighbours according to the logical topology, but in the physical sense may be resident on cores in different nodes of a cluster computer, i.e. they are not neighbours in the physical topology.

At the Centre for High Performance Computing, which kindly provided the use its supercomputers for this project, all programs are submitted for execution as a MOAB [148] job. MOAB is a scheduling system used to optimise the use of programs run on its supercomputers. Apart from some limited configurations for the Bluegene/P, the job scheduler does not allow the assignment of a physical topology. Although the assignment of a physical network topology which mirrors the logical topology does show an improvement in the performance [54] of the parallel FDTD deployed when using MPI on a cluster computer, this could not be implemented for this thesis owing to the abstraction provided by the job scheduler.

The parallel FDTD is deployed on two types of topology in this thesis:

1) The parallelisation of the FDTD on a peer system; the parallel FDTD program is implemented on a collection of processors or cores which use one or more of the communication channels described in table 4.2. Examples are the Bluegene/P and cluster computer. Each processor in the computing system behaves as a peer in that system.
2) The parallelisation of the FDTD using accelerators; communication between the host processor and the accelerator takes place over a communication bus, such as the Peripheral Component Interconnect Express (PCIe) bus. This is also known as asymmetric processing,

Stellenbosch University http://scholar.sun.ac.za

which entails that the dominant component of the processing is to be performed on the accelerator itself. The first mention of the FDTD being deployed in parallel using an accelerator board, is from an implementation on a network of Transputers [22, 23]. The 50 core multiprocessor known as the Intel Phi, to be released in 2013, is also to be configured for use as an accelerator card.

The FDTD is implemented using one of the following types of communication mechanisms:

4.2.3.1 Infiniband

The Infiniband system is a switched communication system that can operate at a network latency of up to 100 nanoseconds and achieves a bandwidth of around 2.5 Gbit/s. The network topology is a switched fabric where compute nodes are connected via a system of high speed switches.

Infiniband operates on a proprietary protocol communication stack that offers good scalability with respect to the number of nodes connected to the switched fabric, i.e. a system of network switches as is shown in figure 4.2.

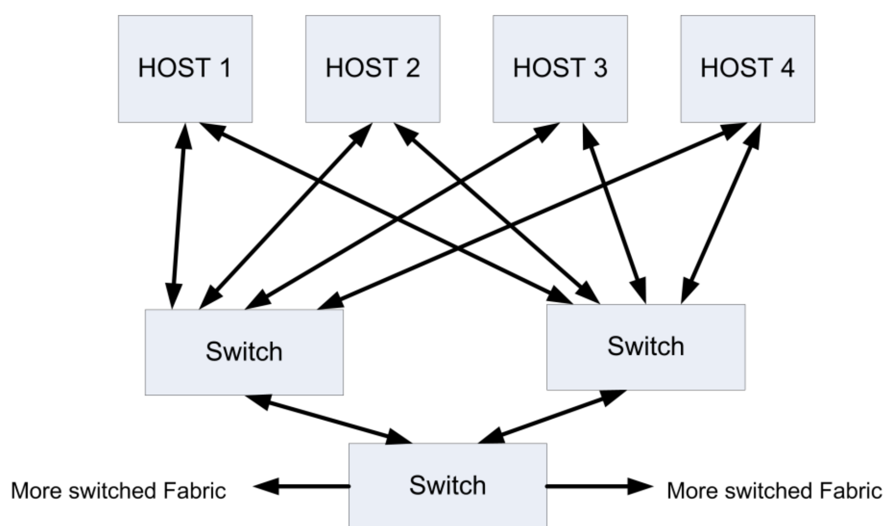

Figure 4.2: Switched Interconnect Fabric

Infiniband only communicates data packets from one compute node to another and does not function in the same manner as an interconnect does on the M9000, Bluegene/P, or Cray XK.

Compared to Gigabit Ethernet, Infiniband offers better communication performance between nodes as it has much lower communication latency [53]. Research by Yu [54] established that the network topology can have an influence on the performance of up to 45% for large clusters running the FDTD with MPI. The network topology used for the work in this thesis was fixed to be hierarchical with a single switch level. The configuration of the nodes to a different topology was not possible owing to the assignment of the processing cores by the MOAB system scheduler.

Stellenbosch University http://scholar.sun.ac.za

4.2.3.2 Interconnects

Interconnects are communication systems that provide inter-processor communication in High Performance Computers. Interconnects provide physical communication channels managed by the operating system of the supercomputer, over and above that described by a switching system as in section 4.2.3.1.

1) Interconnect used by IBM Bluegene/P

The Bluegene/P uses a 3D Torus interconnect, a network topology that will allow good communication with its nearest neighbouring compute nodes in a three dimensional sense, as is shown in the figure 4.3.

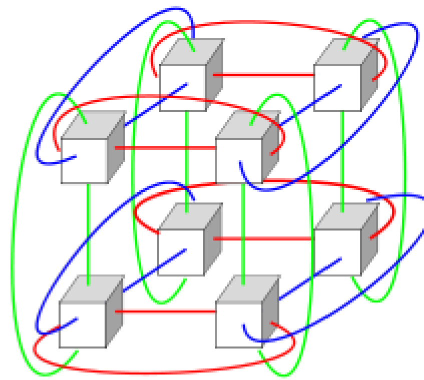

Figure 4.3: Torus interconnect [142]

2) Jupiter Interconnect used by SUN on the M9000

The SUN M9000 Shared Memory Processor uses the Jupiter Interconnect which allows node to node communication by using many data communication paths concurrently. A schematic of the physical hardware of the interconnect is shown below. Multiple system boards, or Compute Memory Units (CMU), connect into a crossbar switch. The crossbar switches are again connected via a PCIe communication bridge. The architecture of the M9000 is based on this interconnect mechanism and affords bandwidths in the order of 737 GigaBytes per second.

Stellenbosch University http://scholar.sun.ac.za

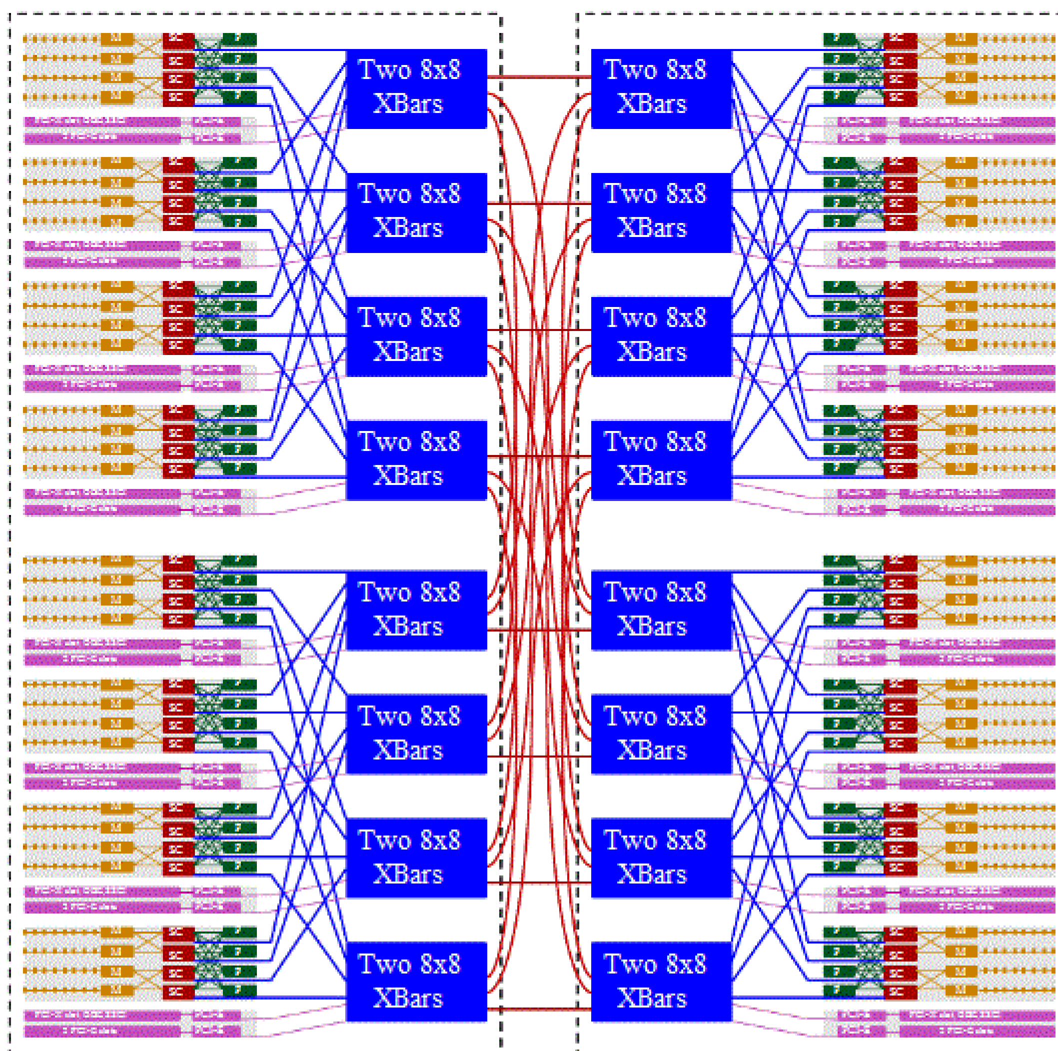

Figure 4.4: Jupiter interconnect (from SUN M9000 architecture manual) [36].

4.2.4 Hyper-threading (HT)/Symmetric Multithreading (SMT)

Hyper-threading, also known as Symmetric or Simultaneous Multithreading Technology (SMT), allows one physical processing core to be viewed as two logical ones by sharing some system resources. It allows two threads to operate in parallel and reduces the impact of memory access latency by overlapping the memory latency of one thread while the other thread is executing. The logical processor is afforded its own set of registers and program counters yet makes use of shared resources on the processor [88].

Tests performed on an Intel Pentium processor showed that processing the FDTD with two hyper threads on a single core takes slightly longer than the processing of a single thread of the FDTD on a single core. It is speculated that the contention for processing resources by the two logical threads using the single processing pipeline is the cause of this reduction in processing speed. There was no performance benefit to using the hyper threading to process the FDTD on the Pentium 4 processor.

Stellenbosch University http://scholar.sun.ac.za

4.3 Examples of HPC systems

4.3.1 Discrete GPUs

The discrete GPU is typically configured to a host computer via a Peripheral Component Interconnect express (PCIe) bus as is shown in figure 4.5 below:

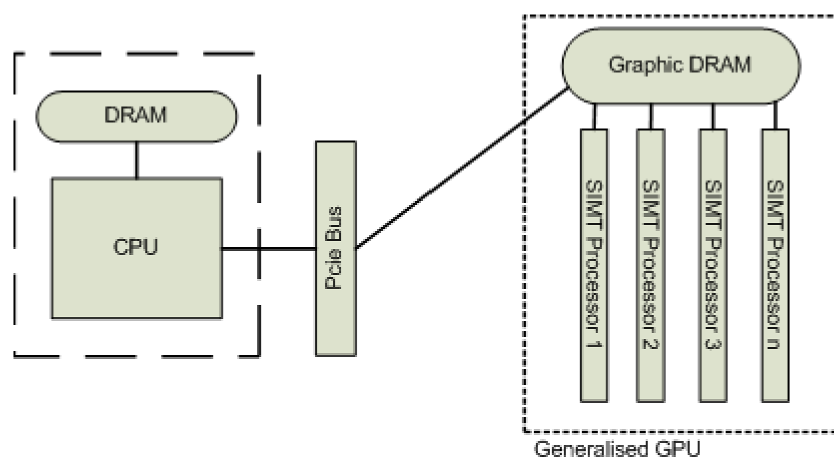

Figure 4.5: Discrete GPU attached to Host computer

The work for this thesis uses GPUs from Nvidia, which, from the aspect of processor architecture, are a collection of SIMD devices called Streaming Multiprocessors (SM) that are connected to a global memory. The SM in turn are composed of a collection of cores, or Stream Processors (SP), also referred to as shaders in some of the literature. Processes are deployed on the GPU using a Single Instruction Multiple Thread (SIMT) approach which requires the same thread to be run on every Stream Processor belonging on one Streaming Multiprocessor. In figure 4.5 the Streaming Multiprocessor has been shown as an SIMT Processor. The architectural relationship of the Streaming Multiprocessor and the Stream Processor are shown schematically in figure 4.6.

The ATI (GPU manufacturing division of AMD) equivalent concept of the SIMT is termed SPMD (Single Program Multiple Data). The difference between SIMT and SIMD is that with SIMT one can use conditional (IF) logic within the SIMT program flow. Intel also uses a graphical processing device called an EU (Execution Unit). Intel will not reveal what the architecture of a graphic EU is but it is speculated to be some form of SIMT device.

The basic structure of these GPUs is described with a comparison to a typical CPU in the simplified schematic in figure 4.6. The GPU's RISC core has available a logic controller and Arithmetic Logic Unit (ALU), whereas the Complex Instruction Set Computing (CISC), such as Intel's Xeon architecture, allows for a controller, an Arithmetic Logic Unit (ALU), and a Vector Arithmetic Logic Unit (VALU, such as SSE) [100]. A more detailed description of the architecture can be found in Nvidia's programming manual [82].

Stellenbosch University http://scholar.sun.ac.za

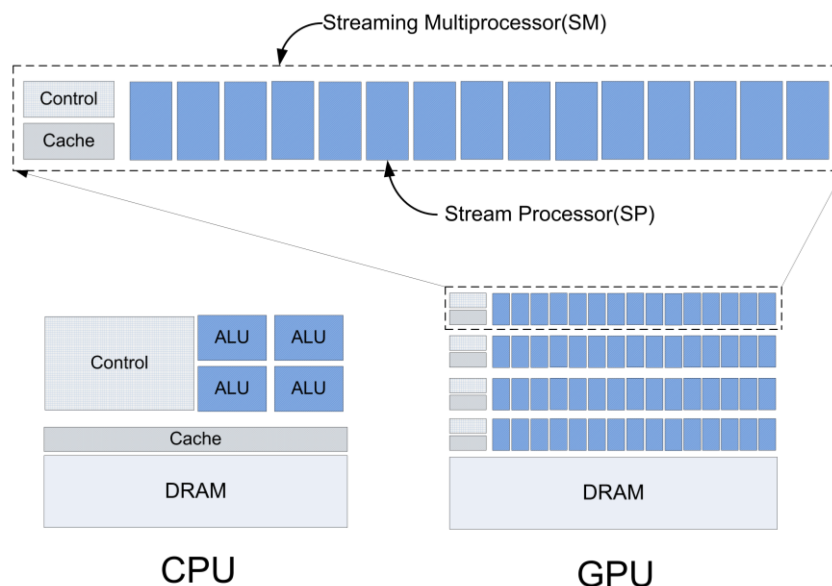

Figure 4.6: Comparison of CPU and GPU architectures [37, 82]

There is a strong analogy between the GPU-CPU relationship and the original Intel 8086 CPU operation with the 8087 maths coprocessor. The 8086 and the 8087 initially operated with the 8086 as a host processor and used the 8087 as a slave processor to perform the floating point operations. As chip die technology improved, the 8087 floating point technology was finally moved onto the host die as the Floating Point Unit as this reduced the communication latency between the two discrete chips.

The schematic of the Intel Accelerated Processing Unit (APU) in figure 4.7 shows that the GPU is now being moved on to the die of the host CPU, thereby allowing more convenient access to the main memory by both the GPU and the processor's cores.

4.3.2 Accelerated programming units

Multicore CPUs are powerful processing platforms, with the cores all having access to the same physical memory. In an attempt to reduce the power consumption and pricing of the CPU and GPU on motherboards, manufacturers have moved the GPU functionality onto the same die as a multicore processor, as is shown schematically in figure 4.7. Processor manufacturers are also moving the memory handler and larger amounts of cache memory onto the chip die itself, as is also illustrated by the schematic of the architecture of the Intel i5 four core CPU in figure 4.7. American Micro Devices (AMD) refers to this technology as Heterogeneous System Architecture Technology, and has trademarked this name.

The memory controller and the cores are now on the same die as the four cores and the GPU. The CPU cores and the GPU in effect now use the same physical memory and this has significant implications for the processing of the parallel FDTD - specifically as the GPU and the processors cores

Stellenbosch University http://scholar.sun.ac.za

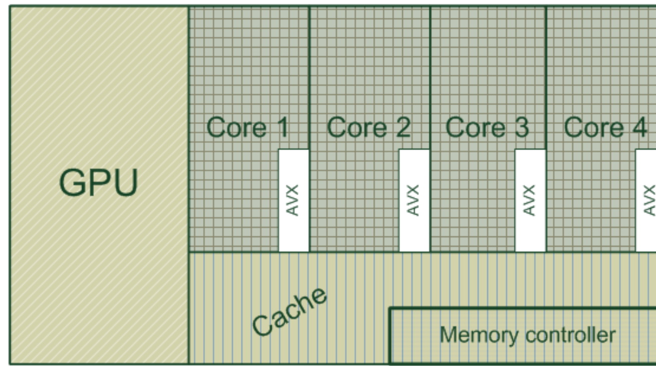

Figure 4.7: APU architecture

now have access to the same physical memory. This close association of memory suggests that the parallel processing power on the integrated GPU and the multicores can be combined.

The SSE and Advanced Vector eXtensions (AVX), compared in figure 4.8, are not independent hardware components but a collection of registers resident on the die of some microprocessor cores. These registers are highlighted here because of the similarity in programming method when compared to a multicore GPU. The SSE, AVX and GPU allow the parallelisation of the FDTD in a data parallel manner, i.e. the FDTD is processed in an SIMD or SIMT manner on both the AVX and the GPU. The SSE register length is 128 bits long, the AVX registers are 256 bits long, although the AVX specification will allow them to be extended to 1024 bits [99]. These registers have also been described as belonging to the VALU section of the processor die [99, 100].

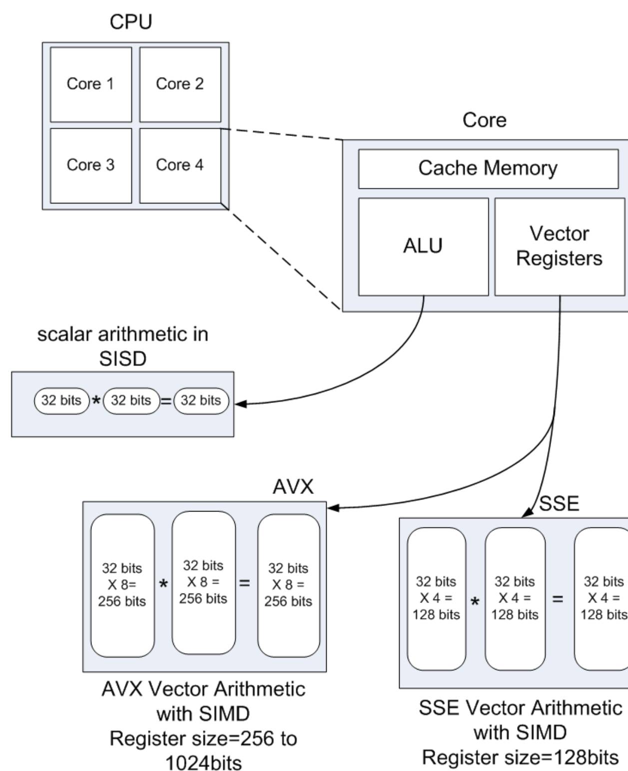

Figure 4.8: A comparison of AVX and SSE on a multicore processor.

Stellenbosch University http://scholar.sun.ac.za

4.3.3 SUN M9000 Shared Memory Processor (SMP)

The M9000 is a shared memory processor produced by SUN Microsystems. The processing power is provided by 64 Sparc VII 64 bit processors which are all able to communicate via a highly efficient memory interconnect as is shown in the schematic in figure 4.9. The Scalable Processor ARChitecture (SPARC) is a RISC processor technology developed by SUN Microsystems. This will make available 64 x 4 = 256 processing cores as each Sparc VII processor has four cores. Simultaneous Multithreading allows each core to be capable of supporting two processing threads simultaneously.

Although the Computer Memory Unit (CMU) board, an ensemble of two x SPARC CPUs, has available 128 GB of memory, the interconnection of all the CMU boards by the Jupiter interconnect allows all of the processors to share the memory space. This is in essence the foundation of the term Shared Memory Processor.

The Jupiter memory interconnect is the defining component, as is illustrated in figure 4.4. The memory communication latency remains low for communication between Computer Memory Units (CMU) when compared to the communication latency between the cores on one CMU. That is, threads running on one CMU will incur the same communication penalty as between threads located on two different CMUs. This is due to several levels of caching provided for communication between the CMU and the Jupiter Interconnect.

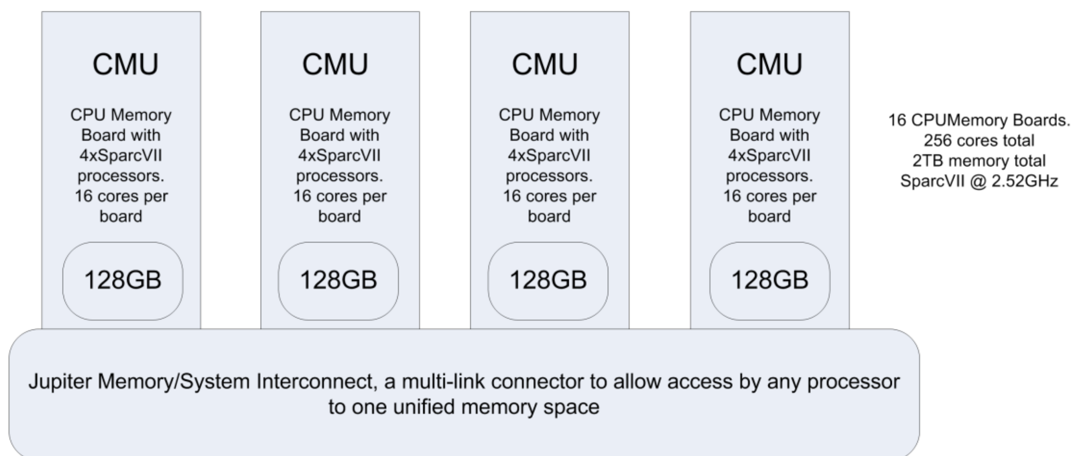

Figure 4.9: A schematic showing the components of the M9000 shared memory processor.

Another advantage provided by the Jupiter interconnect is the ability for the processors to access memory in parallel. As this is one of the key points explaining the good communication between the CMUs, and therefore enables the configuration of memory boards to be viewed as one transparent memory, this is shown schematically below in figure 4.10.

Stellenbosch University http://scholar.sun.ac.za

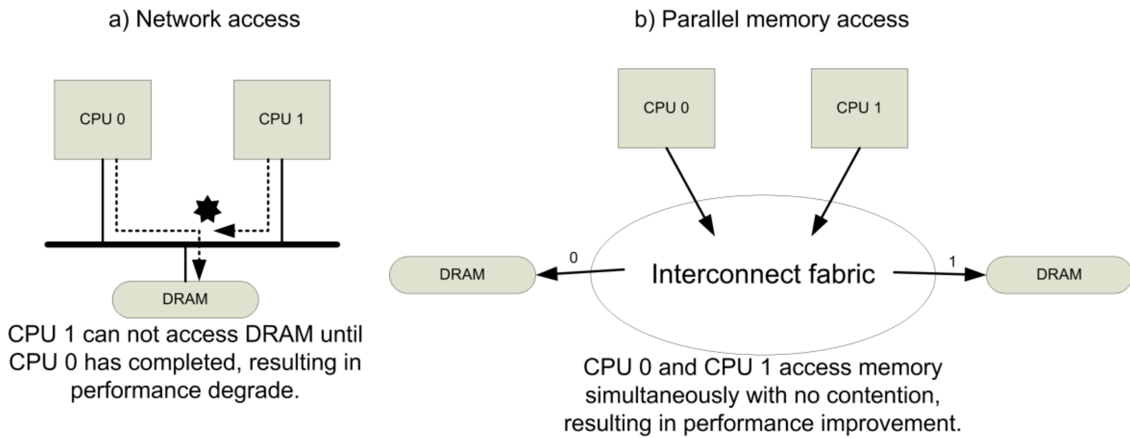

Figure 4.10: The Jupiter interconnect allows Memory Level Parallelism.

4.3.4 Intel multicore at the Manycore Test Lab (MTL)

The Intel MTL (Manycore Test Lab) is an online facility made available to academics by Intel. It consists of several Intel Xeon X 7560 processors connected using Quick Path Interconnect (QPI) technology. The QPI is a switching and communication fabric that is integrated to the Intel processors and allows the system of processors to work in a Uniform Memory space. The Xeon 7560 processor is a multicore chip with eight cores that allows hyper-treading. The processors are configured such that the memory is viewed as a unified memory space by the operating system, as is schematically shown in figure 4.11. By using QPI the system will also allow simultaneous access to the memory as described in figure 4.10 above, i.e. it has Memory Level Parallelism (MLP).

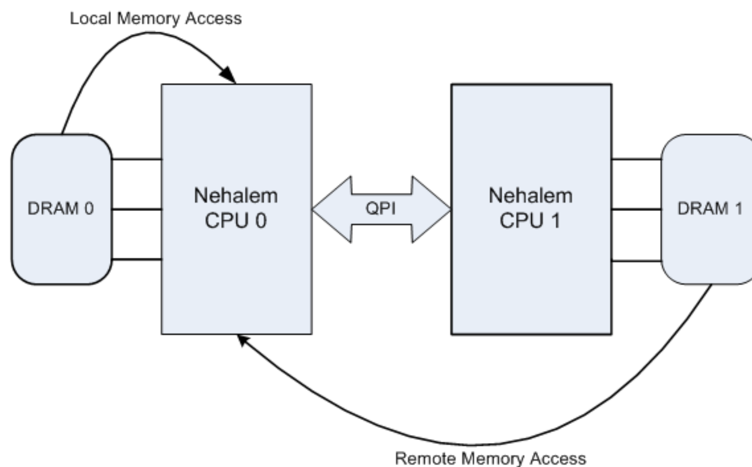

Figure 4.11: Distributed Shared Memory Processor

4.3.5 Nehalem cluster computer and Westmere cluster computer

The cluster computer has the simplest architecture of all the HPC platforms used for this work. The cluster is a collection of multicore processors in a blade configuration, shown in figure 4.12, that

Stellenbosch University http://scholar.sun.ac.za

communicate via a bus system provided by a Local Area Network. At the Center for High Peformance Computing (CHPC), this network structure is made up of a system of Infiniband switches, as is schematically shown in figure 4.13. The Blade configurations are comprised of multicore chips, which are in themselves tightly coupled using the QPI fabric as is shown in figure 4.10. This arrangement of CPUs, memory and the QPI is shown in figure 4.13 below. It is worthwhile noting that the 6275 Blade configuration allows the memory to be accessed via three physical channels, a feature which greatly improves the access to memory and reduces the data bottleneck (discussed in subsequent chapters) experienced by the parallel FDTD on a multicore processors. The QPI fabric will provide cache coherency between the processing cores on all of the CPUs [32] connected by a particular QPI system.

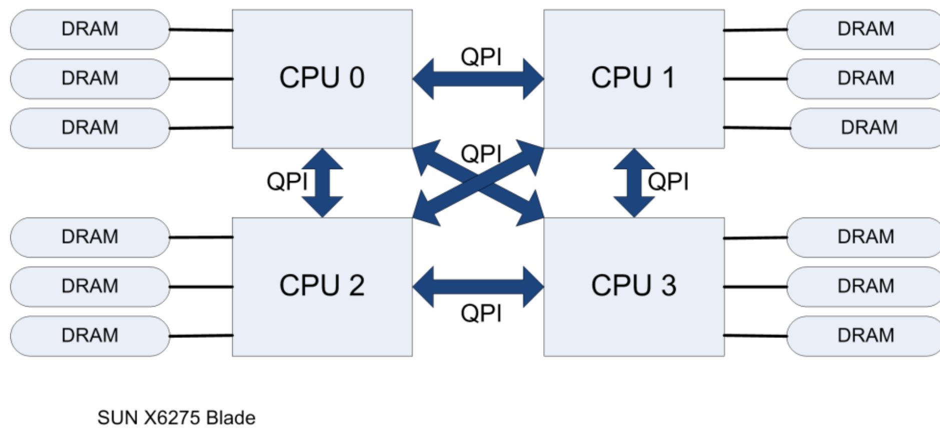

Figure 4.12: Architecture of the Blade 6275 used in the CHPC cluster computers.

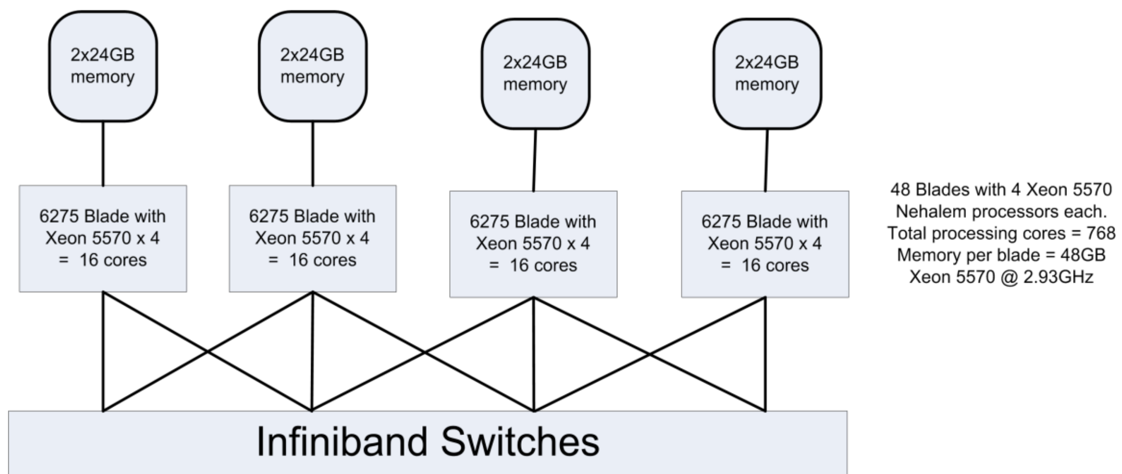

Figure 4.13: Architecture of cluster at CHPC

Two different clusters were available for this thesis:

1) Nehalem cluster

This consists of Intel Xeon 5570 processors operating at 2.9 Ghz set in a Blade configuration. The Xeon 5570 processors have four cores and are hyperthreaded, thereby making available two

Stellenbosch University http://scholar.sun.ac.za

threads per core to total eight threads per processor. The connection network is made up of switch based Quad Data Rate (QDR) Infiniband.

2) Westmere cluster

This consists of Intel Xeon 5670 processors operating at 2.9 Ghz in the 6275 Blade configuration. The Xeon 5670 processors have six cores and are hyperthreaded, thereby making available two threads per core to total 12 threads per processor. The connection network is made up of switch based Quad Data Rate (QDR) Infiniband.

All of the Blades on the network will contend for network communication bandwidth as required by the process running on the system. The Infiniband connection does not possess any switching logic to provide features such as parallel memory access to one thread, as is found in the typical interconnects.

For this thesis the FDTD is implemented on the clusters using the Message Passing Interface (MPI) threading, discussed in more detail in the subsequent chapters. The Unified Memory Architecture (UMA) of the Nehalem and Westmere processors has also been exploited to produce hybrid implementations of the FDTD using both the openMP and MPI threading methods [41].

4.3.6 Bluegene/P (IBM's massively parallel processing system)

The Bluegene/P is a collective of compute nodes that are connected by a sophisticated and configurable interconnect system. The communication interconnect between the compute nodes can be configured to reflect different processing strategies and topologies. The 1024 PowerPC 450 processors available on this machine each have access to two GB of memory. This makes a total memory of 1024 x 2 Gb = 2 TeraBytes of memory space available to the system. The machine consists of 32 compute nodes, each of which in turn has 32 processors on it. Each processor consists of an IBM PowerPC 450 which has four cores. From this we can calculate that the machine has 32 x 32 x 4 cores available for processing. A schematic of this assembly, taken from the IBM programming manual [104], is shown in figure 4.14.

Stellenbosch University http://scholar.sun.ac.za

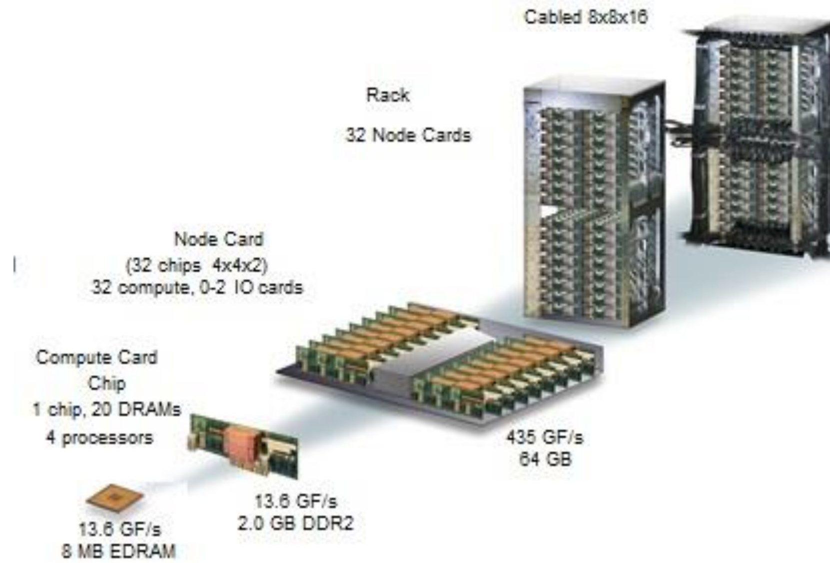

Figure 4.14: Bluegene/P components [104]

The Bluegene/P is a collection of low power IBM Power 450 processors connected by highly configurable interconnect. Each processor node has two GB of memory and four cores, the arrangement of which is shown in figure 4.15. The Bluegene/P allows the interconnect to be configured to map the topology of the hardware to the requirements of the FDTD. The advantage of mapping the compute node topology in this way is that it will allow more low latency local communication than the slightly longer remote latency.

The Torus memory interconnect is particularly suitable for the processing of the 3D FDTD as the physical topology reflects the structure of the FDTD in a 3D cubic mesh [17].

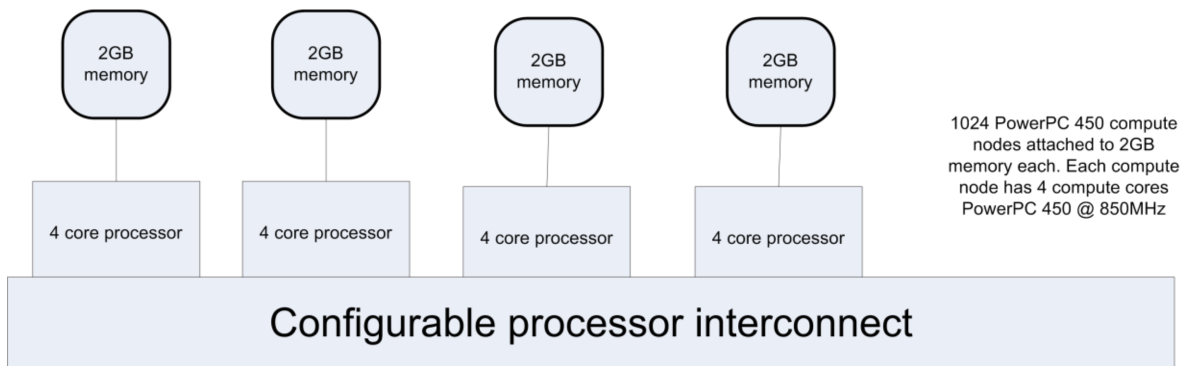

Figure 4.15: Blugene/P architecture as seen by the FDTD application.

 Stellenbosch University http://scholar.sun.ac.za

4.3.7 Intel's Many In Core (MIC) processor

Intel's Many In Core architecture is an accelerator card that can be used for numerical processing. Although it is not available for experimentation in this thesis it addresses several features relevant to the performance of the FDTD on this platform and these are addressed in chapter 5. Intel's MIC processor is configured to the host processor as an accelerator using the PCIe form factor in the same manner as the discrete GPU in section 4.3.1, and as is shown below in figure 4.16. The MIC processor has up to 61 processing cores on the die as opposed to the typical multicore processor which typically has four to ten cores. The cores are supplied data via a ring interconnect which has access to GDDR5 memory via 16 memory channels, as is shown in the architectural schematic of the MIC in figure 4.17.

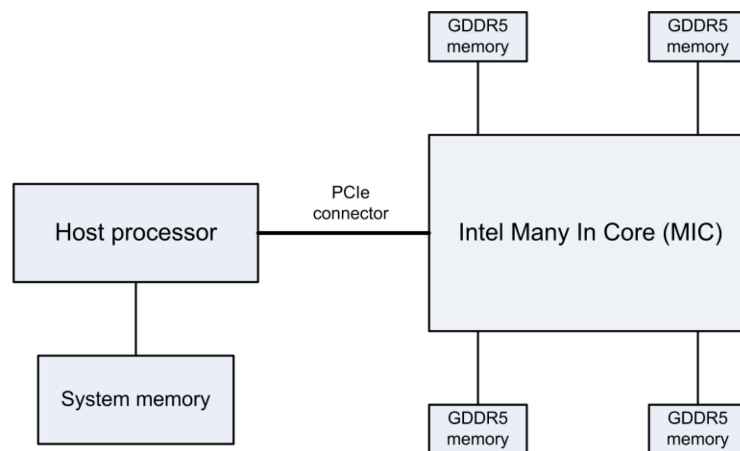

Figure 4.16: The Intel MIC configured to the host processor via a PCIe connection

The length of the vector processing registers has been extended up to 512 bits, which is double the length of the vector processing registers in the Intel i5 series of processors, as is described in section 4.3.2. This gives each core on the MIC the potential to process eight single precision floating points simultaneously. For a MIC processing with 50 cores this would give the potential to process 50 x 8 = 400 single precision arithmetic operations concurrently. The Tag directories are a feature of the distributed infrastructure used to manage the cache coherence of the many cores. Considering the large number of cores accessing one unified memory space, the tag directories are a vital construct required to achieve good processing performance for applications that require inter process communication, such as the parallel FDTD.

Stellenbosch University http://scholar.sun.ac.za

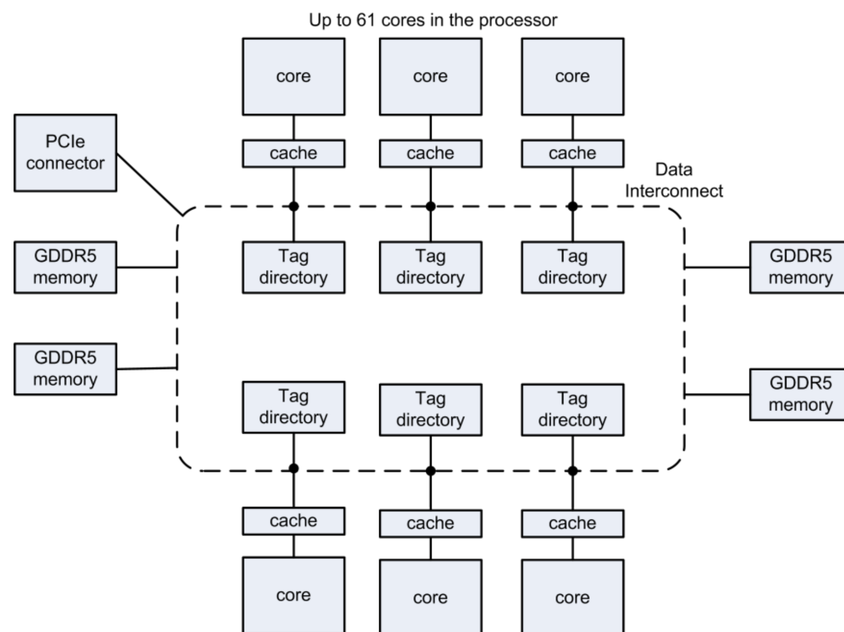

Figure 4.17: A schematic of the architecture of the Intel MIC processor.

The Intel datasheet states that the power consumption at peak performance for this device will be 300 watt.

4.4 Other considerations

4.4.1 Power consumption of high performance computer systems

Owing to the large number of processors, usually operating at very high clock rates, HPC platforms are generally assumed to consume a lot of power. The table 4.3 below lists the power consumption of typical HPC configurations. The power consumption is highly subjective as the HPC architectures and system sizes are so very different and because different algorithms use different components of the system.

As an example, a comparison of the Bluegene/P with its 4096 compute cores and the SUN M9000 with 256 cores in table 4.3, shows that despite the M9000 having fewer compute cores, the power consumption is virtually the same. This can be partly explained owing to the Bluegene/P strategy of using many low power CPUs (approximately 600 Mhz), whereas the SUN M9000 CPUs operate at a much higher clock frequency (approximately 2.88 Ghz). As will be shown in chapter 6, despite consuming the same amount of power, the Bluegene/P is much more efficient at processing the parallel FDTD than the SUN M9000.

Stellenbosch University http://scholar.sun.ac.za

HPC system	Cores	Power consumption	Power per core
IBM Bluegene/P*	4096	40 kW per rack x 1	9.7 W
Sun M9000-64*	256	38.8 kW	151 W
Nehalem Cluster*	8000	80-100 kW per rack	10 W
Nvidia C 2070 GPU*	448	238 W	0.5 W
Intel multicore MTL*	32	170 W	5.3 W
Intel MIC	50+	300 W	6 W
Opteron Cluster[82]	2576	300 kW	116 W
Fujitsu K	705,024	12.6 MW	17.9 W
S870 GPU cluster	8192	16 kW	2.0 W

Table 4.3: Power consumption for a variety of HPC systems (* systems used in this thesis)

Table 4.4 shows the results of normalising the power consumption on an “equivalent number of cores” basis for different types of processor architectures [124] when calculating an equal time to solution, i.e. a test program was processed to completion in an equivalent time period using the relative number of cores shown in column 1 of table 4.4. Compared to table 4.3, table 4.4 gives a different perspective of the power consumption of the Bluegene/P. Although the Bluegene/P processors consume less power per processor, more processors are required to solve a problem in the same time as fewer cores on a more powerful architecture.

Equivalent # cores	HPC cystem	Power consumption(kW)
10	IBM Power-7	0.61
20	Intel Nehalem	0.72
20	Intel Westmere	0.59
30	IBM Power-6	4.79
64	Barcelona	3.52
160	Bluegene/P	1.23

Table 4.4: Power consumed to process to an equal time to solution [124].

As illustrated by the discussion above, the consumption of power by an HPC platform is a highly subjective issue. Koomey’s law [94, 95], discussed in more detail in section 5.2, shows that the amount of power consumed by a typical CPU is halved every year [95], or to paraphrase this : “at a fixed computing load, the amount of battery you need will fall by a factor of two every year and a half.”

There is a general trend to reduce the power consumed by devices ranging from CPUs to disk drives and communication switches. The power consumption of HPC systems can be roughly classified on the following basis:

1) Newer HPC platforms are more energy efficient than their predecessors. This assertion is supported by Koomey’s law [95].
2) Larger HPC systems consume more energy than smaller ones.
3) HPC that have been designed for low power consumption are probably more energy efficient than systems of the same generation that have been built on an ad-hoc basis.

Another attempt to classify the power consumption of HPC systems is the Green Index [143], a method of combining various high performance computing benchmarks into a single number. This

Stellenbosch University http://scholar.sun.ac.za

Green Index recognises the fact that benchmarks such as the Linpack rating [84] loads predominantly the CPU and other components of the High Performance processing platform, such as the communication system. An investigation by Hackenberg et al. [83] shows the FDTD method implemented with MPI on a cluster computer will only consume 70% of the power consumed when running the LINPACK [84] rating. This reduced power consumption is very probably a feature of the explicit nature of the FDTD method, i.e. the LINPACK rating is a very compute intensive process and requires very little inter-processor communication, whereas the FDTD is less compute intense owing to the higher inter-processor communication requirement.

4.5 Summary

The analysis in this chapter has described a variety of different high performance architectures, an important feature for the deployment of the parallel FDTD being the physical relationship of the processors and associated memory. In order for the parallel FDTD to achieve good efficiency it is critical that the memory must supply sufficient data to the processing cores, i.e. the system memory must have good bandwidth and low latency.

Stellenbosch University http://scholar.sun.ac.za

Chapter 5

5 Implementation of parallelisation methods for the FDTD

5.1 Overview
5.2 Amdahl's law, Koomey's law
5.3 FDTD and parallel processing dependencies
  5.3.1 Verification and precision of the parallel FDTD results
  5.3.2 Load Balancing
  5.3.3 Thread affinity
  5.3.4 The use of single and double precision variables
5.4 Parallel FDTD with openMP
  5.4.1 openMP on the shared memory architecture
  5.4.2 openMP on the Distributed Shared Memory Processor (DSMP)
5.5 Parallel FDTD with MPI
5.6 Acceleration of the FDTD with Vector ALU/Streaming SIMD Extensions and cache prefetching
5.7 Parallel FDTD for the GPU using CUDA
5.8 Parallel FDTD with low level threading
5.9 Parallel FDTD on multiple GPUs
5.10 Parallel FDTD on the GPU using the DirectCompute interface
5.11 Accelerated Processing Units and heterogeneous system architecture technology
5.12 Parallel FDTD on the Intel Many In Core (MIC) processor using openMP
5.13 Summary

5.1 Overview

The objective of the parallel FDTD method is to reduce the processing time of the FDTD computation. This chapter is about the different ways that the FDTD can be parallelised on a variety of hardware platforms. The intention is to decompose the parallel implementations so as to identify what they have common and what makes them unique.

The parallelisation methods involve a deployment of the parallel FDTD on a variety of hardware architectures and are influenced by the following factors:

1) The physical structure of the underlying hardware architecture.
2) Communication methods between the parallel FDTD components.
3) Efficiency of access to the memory and other architectural primitives of the devices supporting the parallel method.

A timing analysis of the serial FDTD in section 3.9 has shown that the bulk of the computational load is taken up by the repeated FDTD calculation of the electric and magnetic fields, as shown in the processing flow of the serial FDTD in figure 5.1 below. This chapter is directed at the parallelisation strategies for the calculation of the electric and magnetic fields only, as the processing load resulting from output of the resultant data and other effects has been purposefully minimised.

 Stellenbosch University http://scholar.sun.ac.za

Although several commercial brand names and products such as Nvidia and Intel have been used in this chapter, it is not meant to display any form of preference for the manufacturers or their competitors.

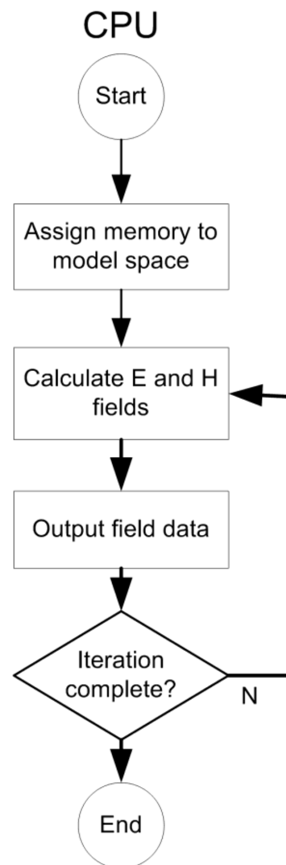

Figure 5.1: The serial form of the FDTD

The parallelisation strategies discussed below deal with how a particular parallel FDTD processing concept is adapted for a specific hardware platform, and identifies why the two are related.

5.2 Amdahl's law, Koomey's law

Amdahl's law is an approximation of the maximum theoretical speed up a parallelised program can achieve on a multiprocessor system. The speedup is still measured in the conventional sense as:

$$Speedup = \frac{Serial\ process\ time}{Parallel\ process\ time}$$

An algorithm is assumed to consist of two components, one that will operate in serial form and one that can be parallelised. The speedup is defined by the ratio of the time required for serial execution to the time required for computing the same results in parallel. Even if one were to speed up the parallel component of the algorithm such that it took close to no time at all, the program execution would still need to execute the finite serial component. In his original paper, Amdahl [39] did not

Stellenbosch University http://scholar.sun.ac.za

formulate the equation as shown below as formulation 5.1, but it serves to describe the speedup behaviour of the basic framework of a lot of the parallelised systems.

$$\text{Maximum Speedup} = 1/((1-p)+(p/n)) \qquad 5.1$$

Where p is the fraction of the code that can be parallelised and n is the number of processing cores.

Several assumptions are made in this formulation. These are:

1) The serial and parallel code will run at the same speed.
2) The serial and parallel code sections are not pipelined, i.e. partly overlapping code sections.
3) There are no time penalties from inter-process events and thread or process management.

As the number of processing cores used in the parallelisation increases to be much larger than one, the Amdahl formulation above indicates that the parallel process achieves very little speedup. This is reflected by the near horizontal component of the speedup curve shown in the theoretical processing time below:

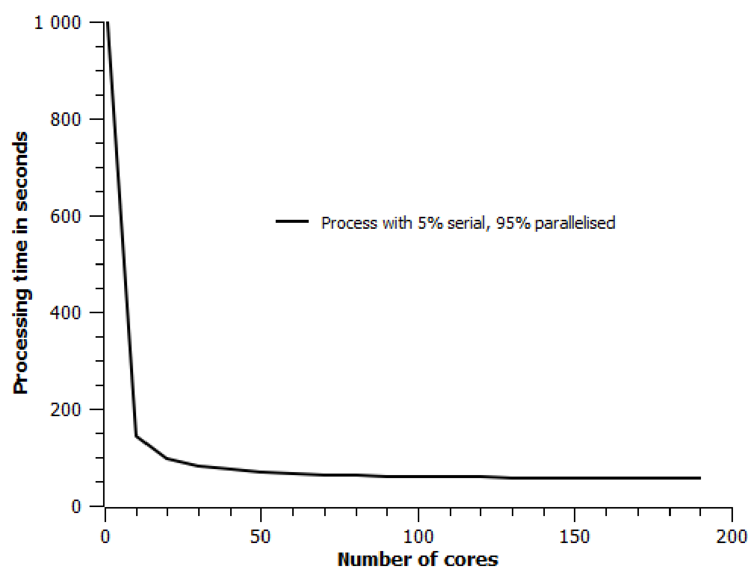

Figure 5.2: Amdahl residual for a process that is 5% serial and 95% parallelised

In practical terms, Amdahl's law is also limited by the effects of computing hardware used to parallelise the FDTD [8]. One has to consider that the operational burden on the operating system will increase as the physical size of the multicore and memory hardware increases and that this will reduce the effect of the speedup. As the size of the memory configuration increases to cope with ever larger problems, so will the ability to supply the processing cores with sufficient data [51, 89]. This exacerbates the memory access problem to the extent that the parallel process experiences a

Stellenbosch University http://scholar.sun.ac.za

gradual "slow down" as the number of processing cores increases. This is demonstrated in figure 5.3, i.e. beyond a certain number of cores the parallel code runs increasingly more slowly.

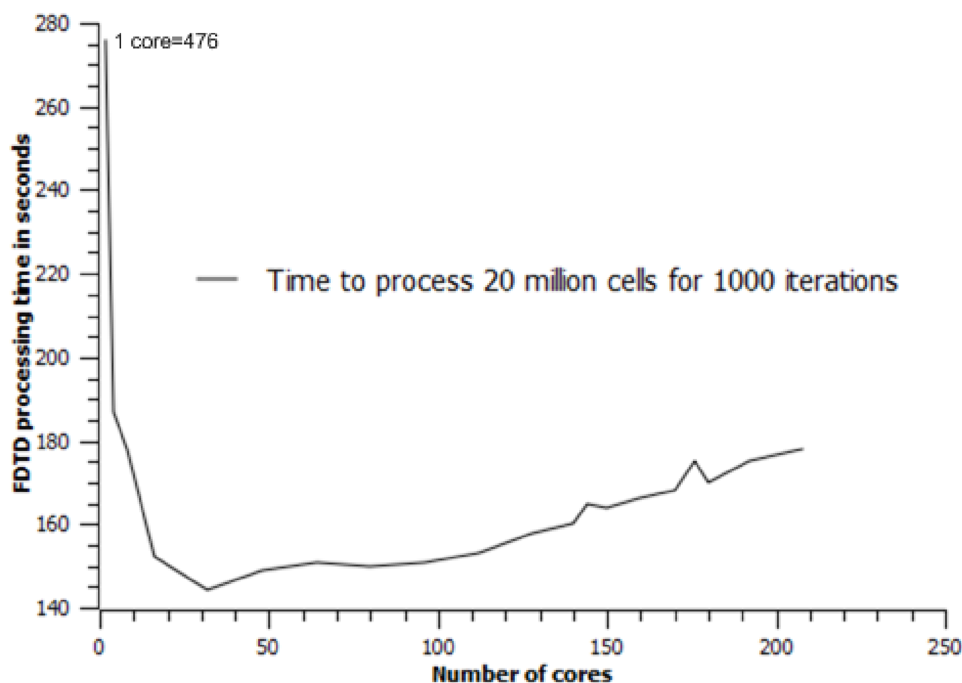

Figure 5.3: Processing the FDTD using openMP on the M9000 showing the parallel slowdown

One aspect that is readily overlooked when discussing issues related to Amdahl's law is that there is always scope for speeding up a serial process to accelerate the overall process. This logic is an indication of the requirement for a hybrid architecture to speed up the parallel processes as much as possible, i.e. the parallel component is accelerated in the conventional sense and the serial component of the algorithm is assigned to the quickest serial processor available.

There are several contributors to the residual of the Amdahl residual, such as the general process start-up, thread management overhead incurred by the operating system, and increased memory management penalties inherent in larger systems. Figure 5.2 shows the purely theoretical Amdahl residual for a computer with up to 200 cores for a serial component of 5% and a parallel component of 95%.

Koomey's law [94, 95] states that the processor energy efficiency is doubled every eight months. The results of an analysis [95] is shown in figure 5.4 below. Apart from giving an indication regarding processor performance similar to Moore's law [103], i.e. that the performance increase for processors is doubling every one and a half years, it also suggests that the more modern the processor, the more energy efficient the High Performance Computing (HPC) system. The implication for the parallel FDTD is that later generation HPC platforms will provide better performance for less power.

Stellenbosch University http://scholar.sun.ac.za

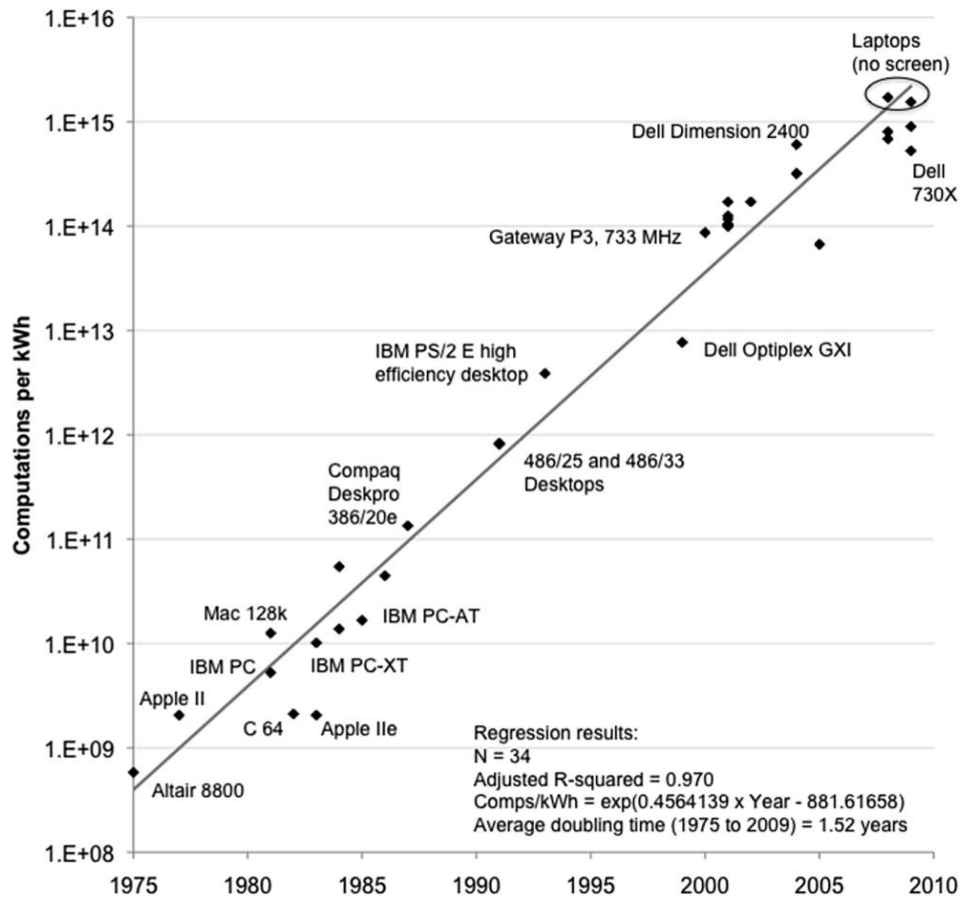

Figure 5.4: Koomey's assertion suggests that the computational performance of a processor doubles every one and a half years. (source of graph from [94])

5.3 Parallel FDTD processing dependencies

The conversion of the FDTD process from its serial form into a parallel form is achieved by dividing the computational load into discrete sections (or chunks) and processing each section as an individual process, or thread [33]. Owing to the Maxwellian dependency of the adjacent electric and magnetic field values, each processing or cell point is calculated according to a calculation template, shown as a cross in figure 5.8. This template, or a variation of this template, is also known as the FDTD computation stencil. The FDTD processing threads require synchronisation after every iteration. For some processing models this data synchronisation is quite simple as all of the threads operate within the same address space, such as with openMP on the SUN M9000. Process synchronisation is subject to load balancing on a hybrid processing platform, such as one made up by dual GPU boards and multicore CPUs, where load balancing and synchronisation of the FDTD threads becomes somewhat more complex.

Stellenbosch University http://scholar.sun.ac.za

5.3.1 Verification and precision of the parallel FDTD results

The verification of the parallel FDTD is a comparison of the resultant output of the serial FDTD and the parallel FDTD. Two different processes were used to verify the results of the parallel FDTD codes during the software implementation cycle. One method entailed the comparison of explicit electric or magnetic field values at predetermined locations in the serial and computed parallel FDTD grid, the other consisted of comparing two dimensional images slices of the resultant parallel FDTD computation against those from serial FDTD computations.

The resultant electric and magnetic fields of the parallel FDTD can be written to disk in a simple image format for every iteration cycle. The resultant images from the parallel processing are computationally subtracted from the resultant images produced by the serial form of the FDTD to determine any differences between the parallel and serial calculation of the FDTD.

In some cases, such as when MPI is used on a cluster, the resultant data is produced as an image strip for the section being processed. These image strips are then combined into one continuous image showing the complete electric or magnetic field components.

Figure 5.5 below shows a comparison of the explicit numerical results from the three dimensional FDTD on three different platforms. The serial and parallel results are shown to accord well. There are minor variations in the precision of the results, very probably owing to the differences in the implementation of the floating point definitions on these different computing platforms and because of the natural error in the precision inherent in calculations involving floating point numbers [159, 160].

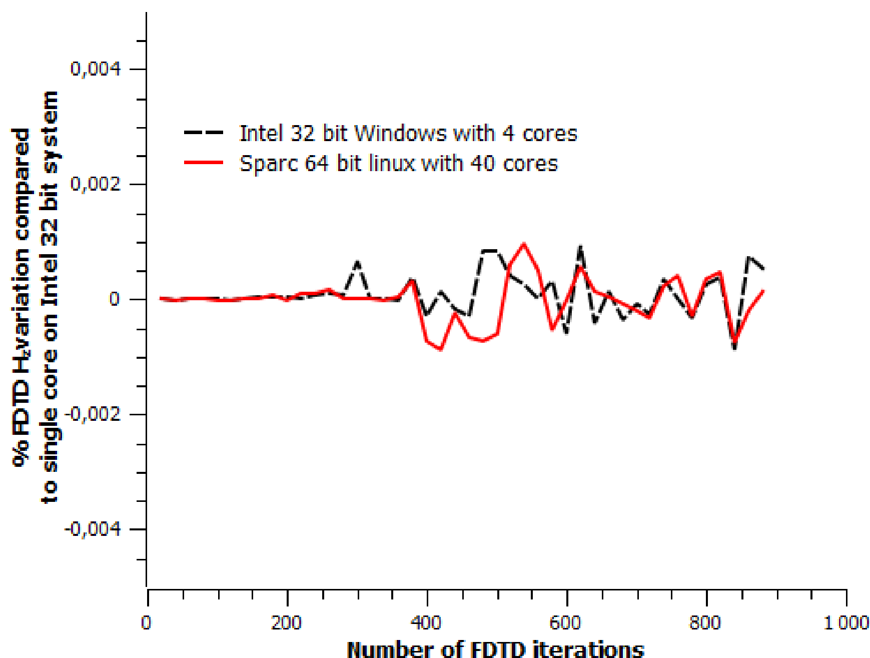

Figure 5.5: A comparison of the precision of the results from the serial and parallel FDTD on different computing platforms.

Stellenbosch University http://scholar.sun.ac.za

5.3.2 Load Balancing

It is important to synchronise all threads at the end of each iteration cycle in order to maintain the integrity of the computation. To calculate electric or magnetic values from adjacent cells that come from different iteration cycles would corrupt the FDTD calculation; this could happen if the thread loads are unbalanced or unsynchronised unless care is taken. The implication for the efficiency of the process is that the parallel FDTD would only perform as quickly as the slowest thread.

It would be an ideal condition to process the FDTD with parallel sections that all finish processing their respective computational load at the same time, as thread synchronisation would not be necessary. This is unfortunately not the case as threads with equivalent processing loads will take different times to complete owing to variables such as latencies in memory interconnects, inter thread communication latencies, and other effects [19]. Unless synchronisation controls are built into sections of the threads involved in the parallel computation, the reduction of the processing inefficiency resulting from this unbalanced processing is a manual and iterative task.

5.3.3 Thread affinity

The HPC systems used for the parallelisation of the FDTD in this work all use multicore processors, also referred to as Chip Multi Processor (CMP) [1], as the basic building block of the systems whether these are clusters, SMP, the Bluegene/P or GPUs.

These systems all have a multi-layer communication structure between the processors and the memory available to them, some processors communicating with the memory on the same chip while some processors require communication with memory that is not on that same node, in this case referring to multi-processor, Computer Memory Unit (CMU) or Blade. The performance of the communication between processors and memory on the same IC is usually much better than the communication to processors or memory off the chip [51]. With the use of threading mechanisms such as openMP, threads are continuously created and terminated and the scenario may occur where a thread is initiated on one core but the associated data is resident in the memory of some other processor or node. The probability of this happening obviously increases as the number of cores in a Shared Memory Processor (SMP) system increases, i.e. it is more likely to encounter openMP performance issues when coding with openMP on SMP systems with a large number of nodes. The Message Passing Interface (MPI) threading for the FDTD algorithm is generally coded such that a program or code thread is closely related to processor or set of processors and therefore the probability of dealing with un-coalesced data is avoided. This is illustrated in the performance of the FDTD on the M9000 SMP as per figure 5.9 below.

It is therefore preferable to map FDTD threads to corresponding physical cores so that the affinity of process to core is large. Although it is known that the processor affinity can affect performance of the parallel process as a whole, the impact of the processor affinity is not easily quantifiable [52]. A comparison of the thread duration indicators in figures 5.7 and 5.10 shows that the thread duration, and hence the thread affinity, of the MPI version of the parallel FDTD is larger than that of the openMP based FDTD. The openMP based FDTD spawns and closes new threads for every iteration, whereas the MPI based FDTD uses a fixed number of threads throughout the processing. The FDTD implemented using MPI is expected to perform better than the FDTD implemented using openMP owing to less thread management required by the MPI technology. It is very possible that this is the

Stellenbosch University http://scholar.sun.ac.za

reason why the FDTD using MPI shows better performance than the FDTD using openMP on the M9000 SMP as shown figure 5.9.

5.3.4 The use of single and double precision variables

The programming work undertaken for this thesis uses single precision floating points throughout. Contemporary Complex Instruction Set Computing (CISC) processors incorporate dedicated Floating Point Units (FPU) [164] that will allow fast computations of floating point scalars which are four, eight or ten bytes in length. The use of double precision floating point numbers on processors such as the Intel i5 range of processors, or those using the Nehalem architecture, were not considered to contribute to any performance degradation. This was confirmed by converting a FDTD program using single precision scalars to one that uses double precision values, and comparing the execution times of these programs. The execution times of both programs was indeed exactly the same.

The GPUs used for this thesis do not have the same hardware capability to process the double precision floating point operations as the contemporary CISC processors, and as such, these are performed by a software emulation process. The double precision emulation inevitably consumes more processing power than the single precision equivalent, and this will lead to longer program execution times. The newer Kepler GPU architecture by Nvidia provides hardware support for the processing of double precision values and will very probably not incur the same performance penalty when using double precision values. A GPU with Kepler architecture was not available for this thesis and hence the assumption made above could not be confirmed.

5.4 Parallel FDTD with openMP

5.4.1 openMP on the shared memory architecture

openMP consists of a set of simple compiler directives and an Application Programming Interface (API) that greatly simplifies writing parallel applications for the shared memory architecture machines, as implemented by SUN's M9000 and Intel's Nehalem architecture. It is a computing industry standard that has evolved from the programming of the vector array processors such as Cray supercomputers in the 1990s. openMP also operates on the Distributed Shared Memory Processor (DSMP) architecture described in section 4.3.4.

The execution principle of openMP is based on the fork and join model as shown in figure 5.6, whereby several threads are simply created, executed in the same address space, and then terminated. In this manner, a process can be very easily decomposed into serial and parallel sections. As all of the parallel threads operate within the same memory address space, the speed of data communication between threads is very efficient.

Stellenbosch University http://scholar.sun.ac.za

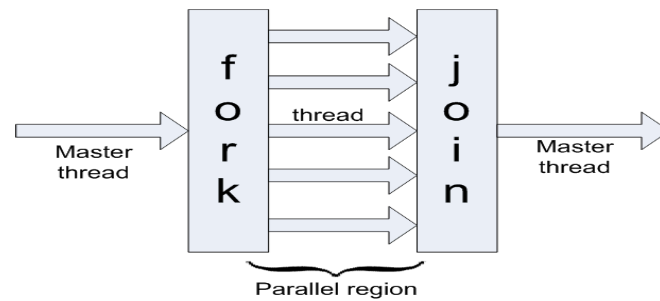

Figure 5.6: openMP principle the creation and termination of threads

The loop process involving the calculation of the electric and magnetic fields is subjected to openMP decomposition and this results in the process flow as is shown in figure 5.7. openMP is particullarly suited for the parallelisation of loops as shown in the code snippet below.

The following pseudo-code shows the calculation of one of the electric field components for the 2D FDTD:

Code segment 5.4:

```
1)       int numt=3;
2)       omp_set_num_threads(numt);

3)#pragma omp parallel for private(i,j)
4)         for (i = 0; i < ie; i++) {
5)             for (j = 1; j < je; j++) {
6)      ex[i][j] = caex[i][j]*ex[i][j]+cbex[i][j] * ( hz[i][j]-hz[i][j-1]);
7)             }
8)         }
```

The number of threads that need to be used in parallel are set up in line 2. One would ideally assign one thread per logical processing core, i.e. some physical cores are capable of Hyperthreading or Symetrical Multi Threading (SMT) and this may allow the use of two threads (two logical cores) to be efficiently deployed on one physical processing core. Using more than one thread on a physical core would not necessarily improve procesing performance [89] as the processing pipleline is bound to be resticted by the limitation in processor resources common to the multiple logical threads.

Line 3 is a compiler directive that will divide the processing loops to fork into 3 threads, but still share the same memory address space.

Stellenbosch University http://scholar.sun.ac.za

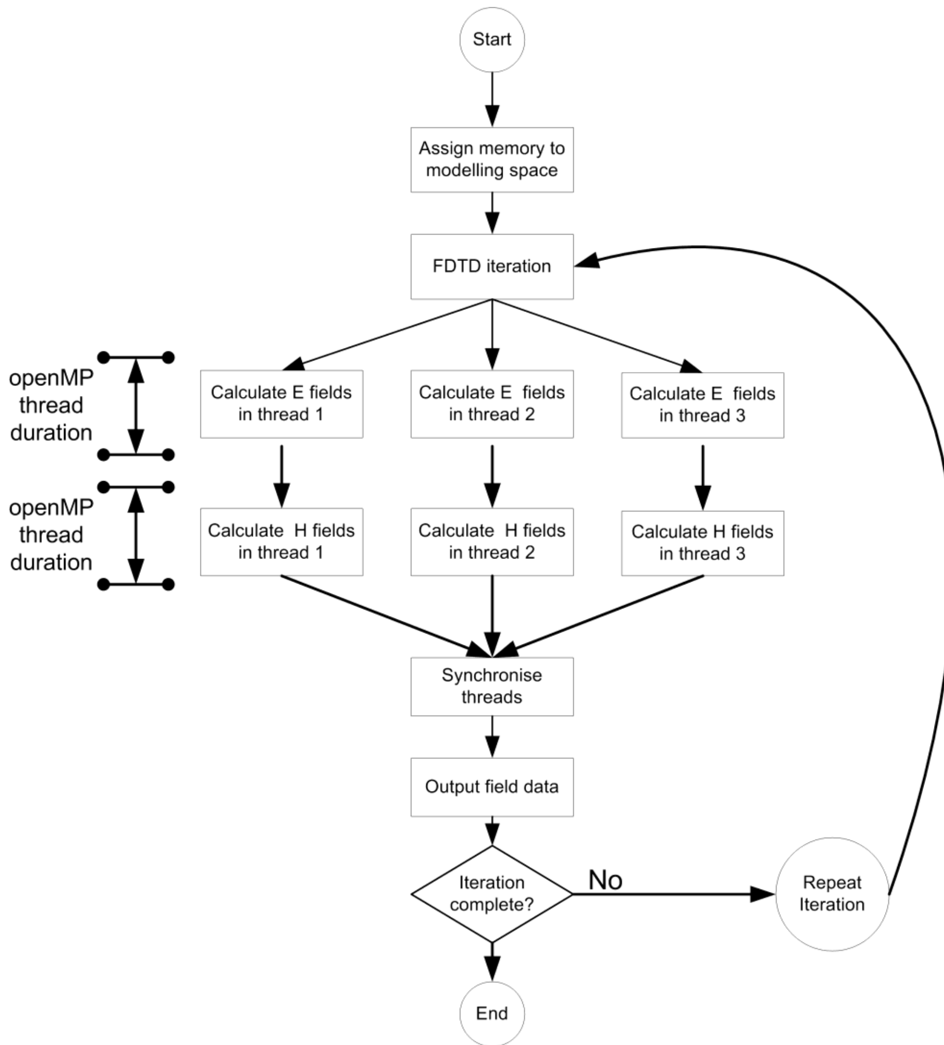

Figure 5.7: The parallel FDTD using the openMP method

Figure 5.7 above shows the allocation of threads for each of the calculations of the electric and magnetic fields for a two dimensional FDTD. The three dimesional FDTD is processed in a similar manner in that the loops that compute the largest processing load are decomposed into individual threads.

Although the FDTD computation is now spread across several threads, the electric and magnetic cell values are all calculated within the same address space as is shown in figure 5.8. Provided that the FDTD points are all in the same iteration, field points in one thread may use adjacent field values from another thread, but using the same computational stencil as is shown in Fgure 5.8. Despite this convenience, the calculation of electric and magnetic values has to be carefully synchronised within each iteration cycle so that the data values in adjacent thread segments are not calculated out of step, i.e. each iteration performs at least one electric and then one magnetic calculation as required by the FDTD stencil. If the threads performing the iteration were not synchronised then the iteration of one thread could proceed on to the calculation of the magnetic field values using electric field values from the previous (or next) iteration cycle.

 Stellenbosch University http://scholar.sun.ac.za

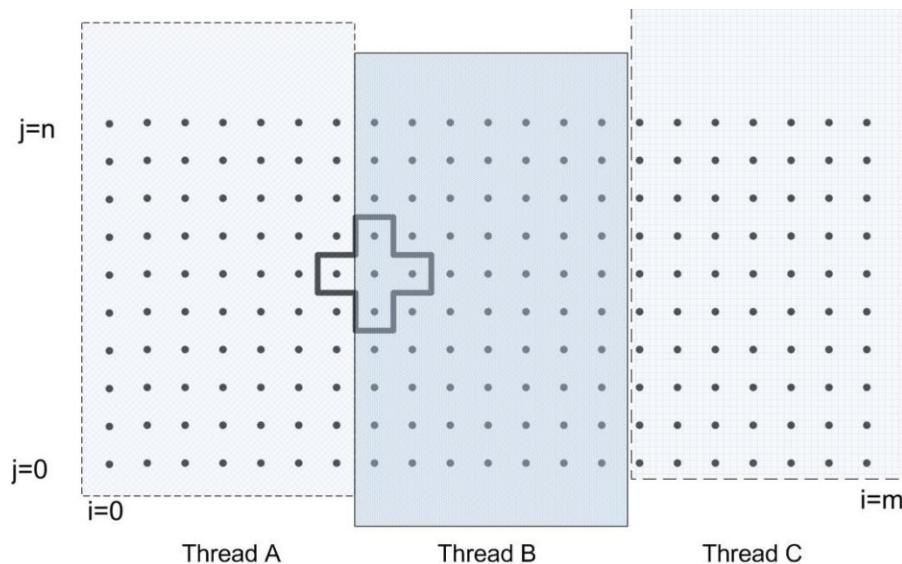

Figure 5.8: openMP threads sharing the same memory address space

The openMP threading library implementation is very simple. The conversion of the serial 3D FDTD to the parallel version using openMP took less than one hour of programming work on the M9000, and provided correct answers.

5.4.2 openMP on the Distributed Shared Memory Processor (DSMP)

The Distributed Shared Memory concept is an abstraction that supports the shared memory architecture although the hardware is physically distributed, i.e. the operating system provides the application with a single uniform memory space thereby hiding the physical distribution of memory. The access to memory from a shared memory architecture is more tightly coupled than that of the access for the distributed memory architecture. Although physically resident on remote and local processors, if the inter-processor communication and memory latency is low, the operating system can view the physically distributed memory as a uniform shared memory. The FDTD implementation as discussed for the shared memory architecture in 4.2.1 also applies to the DSMP. The identical code deployed for the implementation of the parallel 2D and 3D FDTD were run on the Intel's Multicore Test Lab DSMP test system and the results of this are compared to that achieved on the M9000 in figure 5.9.

5.5 Parallel FDTD with MPI

The Message Passing Interface (MPI) was designed to enhance the portability of parallel applications primarily for the distributed machine architectures, such as clusters of PCs, but also supports shared memory architectures. Typical hardware platforms that support MPI are high performance cluster computers, MIC, IBM's Bluegene/P and also the SUN M9000 Shared Memory Processor. All of these hardware platforms can use the exact same parallel FDTD code created with the MPI method. This standardisation of the MPI definition across different computing platforms is a great advantage from the aspect of deployment effort.

Stellenbosch University http://scholar.sun.ac.za

As with openMP, the MPI standard was first supported by the computing industry in the 1990s. MPI has its foundation in the Parallel Virtual Machine (PVM) computing standard, which is now rarely used.

The implementation strategy for the MPI is to make it programmatically simple to spawn program threads from a master program and to provide an intuitive communication protocol for data and events between the these threads. Each thread supports its own memory address space. Data values that are shared between the threads must be explicitly communicated between these threads.

The fundamental difference in the implementation of the MPI and openMP parallel FDTD is that the MPI threads are spawned in their own local address space whereas the openMP threads all operate within the same address space. MPI implementations also perform well on shared memory architectures, in an operational state known as Nemesis mode [17], as the data communicated between the threads are passed as memory address references and not actual values [17]. Typically the parallel FDTD will perform as well with openMP as with MPI on shared memory architectures with a small number of cores (less than eight), but perform better with MPI than openMP for a system with a large number of cores, as is indicated by the graph in figure 5.9. The MPI and the openMP version of the FDTD are not always equally efficient at processing the FDTD owing to the thread management and processor (or thread) affinity [51] exhibited by these two different programming strategies [as described in section 5.3.3]. The MPI strategy uses one thread per FDTD processing chunk, which associates a specific FDTD processing thread with a single core. This is opposed to the openMP strategy which spawns a new set of threads for every large loop event and closes these threads after the loop event, i.e. an FDTD openMP thread is subject to much more context switching because of this, leading to a greater performance penalty than that occurred by the more continuous MPI threads. The result of this performance penalty is manifested in the comparison of the openMP and MPI based FDTD of figure 5.9.

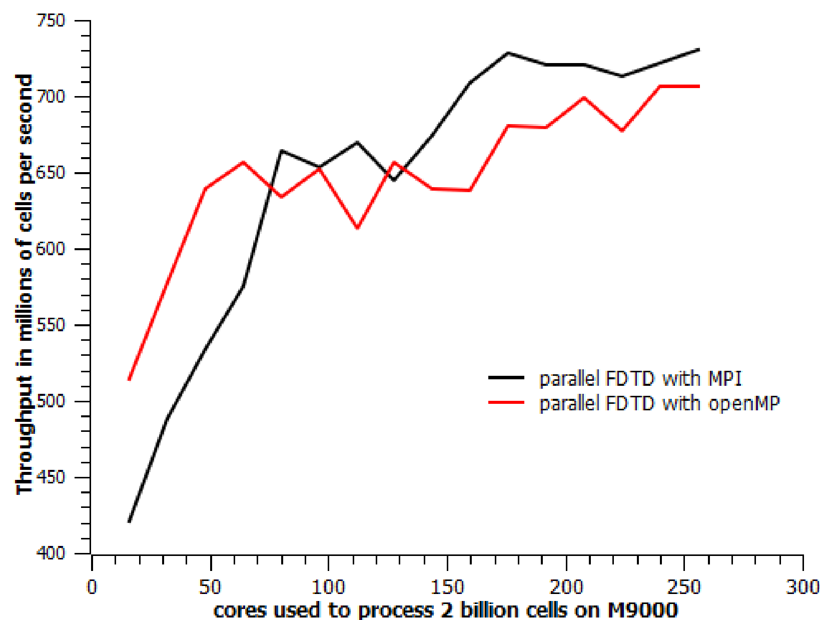

Figure 5.9: A comparison of the throughput of the parallel FDTD with openMP or MPI on the 256 core M9000 shared memory processor.

Stellenbosch University http://scholar.sun.ac.za

The MPI threads that allow the FDTD to be implemented in parallel are shown in figure 5.10 below. They consist of an MPI thread initialisation and a directive from the command line on how many threads are required to be run. To maximise performance the most efficient form of mapping the MPI threads on to the hardware would be to allocate one processing thread per physical core.

The FDTD modelling space has been divided up into proportional FDTD cell chunks for each thread and the electric and magnetic field values are calculated for that chunk.

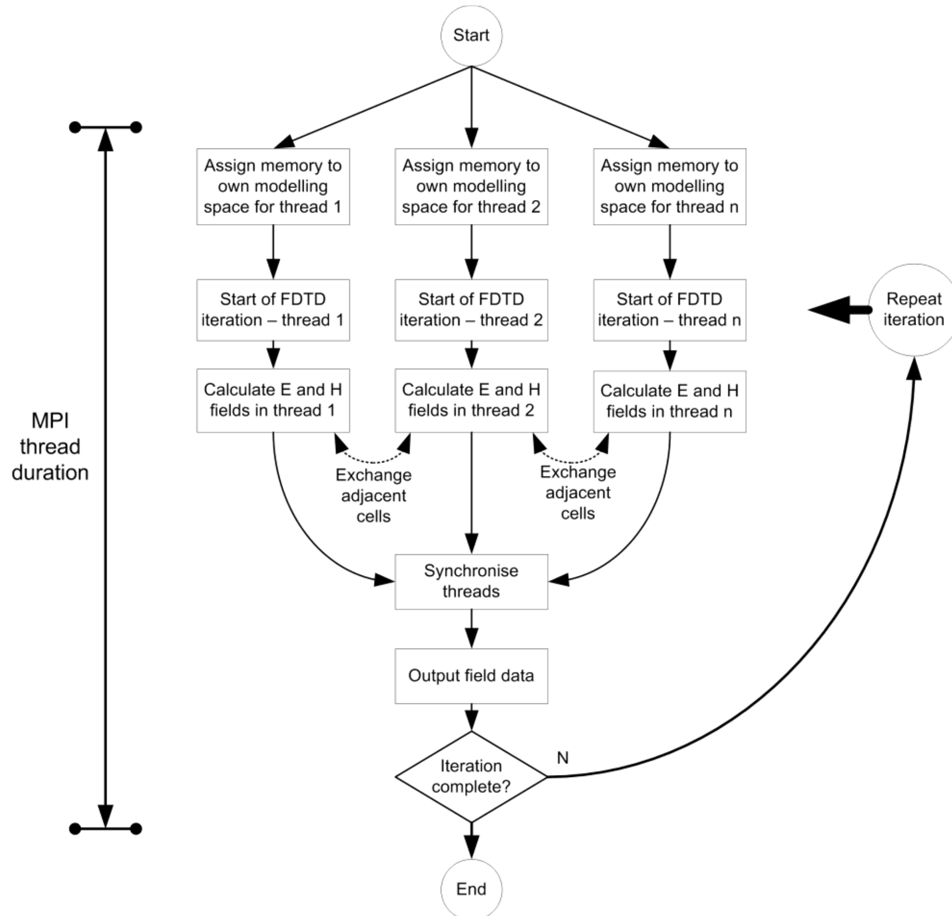

Figure 5.10: Parallel FDTD with MPI

Figure 5.11 shows how a two dimensional modelling space has been divided up between three threads, each thread having its own memory address space. Apart from synchronising the threads at the end of each iteration, some exchange of adjacent data is required by the FDTD, as described further below. This implementation of the algorithm uses very intuitive MPI interface primitives to exchange data between the threads, as is described in the pseudo-code below:

Code segment 5.5:

```
1)          Loop for number of columns
2)            Loop for number of rows
3)                 Calculate magnetic field values according to electric
                    field values in the FDTD stencil
4)                }
5)          }

              Now exchange magnetic field fringes between threads
```

 Stellenbosch University http://scholar.sun.ac.za

```
6)     MPI_Recv(Receive column of magnetic fringe values from thread B to
       replace in thread A)

7)     MPI_Send(Send column of magnetic fringe values from thread A to
       replace in thread B)
```

The schematic in figure 5.11 below shows the processing of an FDTD grid using three threads. The electromagnetic field values are calculated from column 1 to 5. Electric and magnetic field values to compute the values in column 5 are required from column 6, the fringe cells of the data component in thread A. The magnetic and electric values cannot be updated in column 6 in thread A because there are not enough electric and magnetic cell values available to satisfy the requirements of the FDTD stencil, i.e. the column 7 values are resident in thread B. Field values for column 6 are therefore calculated in thread B and are used to update the electric and magnetic values of column 6 in thread A after the iteration cycle has completed. The pseudo-code snippet above shows the calculation of the magnetic field FDTD values for a thread and the exchange of fringe data with the `MPI_Recv` and `MPI_Send` instructions. A similar exchange of fringe data occurs after the electric fields are updated.

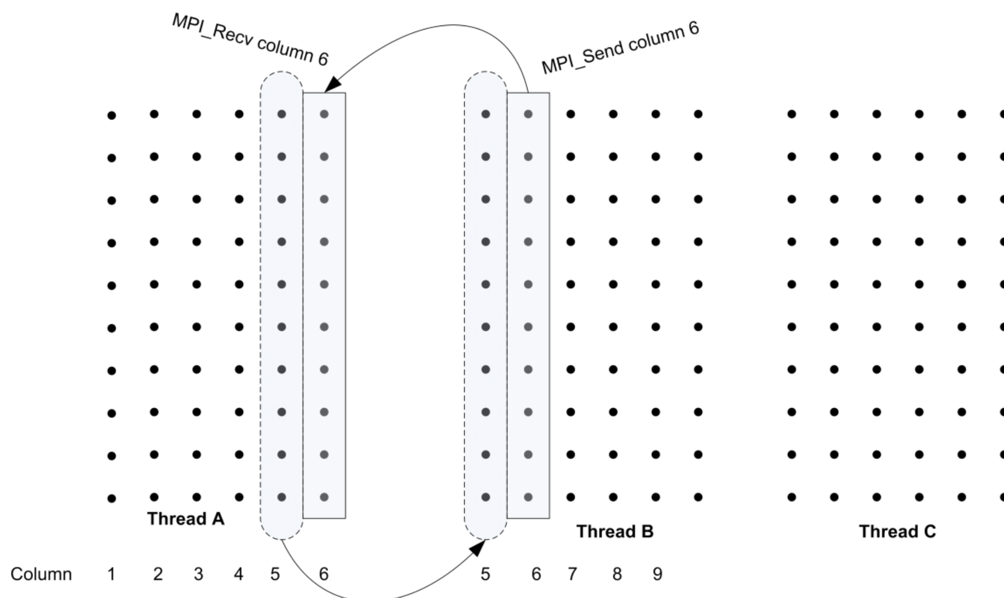

Figure 5.11: Parallel FDTD segments communication fringe data values

The performance of the parallel FDTD implementation is dependent on the speed of the data exchange described above and this is dependent on both the data communication latency of the exchange, and the degree to which the fringe data exchange has been synchronised. For the scenario where the parallel FDTD with MPI has been implemented on a shared memory platform, the fringe data transfer is an exchange of memory addresses only, as opposed to physically transmitting the column values from one thread to another. This results in a negligible data communication latency and an improvement in the performance of the parallel FDTD algorithm. For this situation MPI is said to be operating in Nemesis mode [17].

Stellenbosch University http://scholar.sun.ac.za

5.6 Acceleration of the FDTD with vector arithmetic logic units/ Streaming SIMD extensions and cache prefetching

The current range of commercial mainstream CPUs have a special set of very long instruction length registers and associated instruction sets incorporated within the CPU that can be used for the Single Instruction Multiple Data (SIMD) processing of numerical data. Although this set of registers and instructions was initially envisaged for the on-chip processing of image and video data it has also now been conveniently applied to accelerate the processing of numerical arrays. One implementation example of this on-chip feature is the Streaming SIMD Extension (SSE) which is used extensively for the decoding and encoding of video data.

The SSE are a Single Instruction Multiple Data (SIMD) capability incorporated in the current range of Intel and AMD processors. SSE provides eight 128 bit registers and several dedicated operators for the manipulation of data with these registers. Each SSE register can accommodate four 32 bit floating point numbers (single precision). The term VALU for Vector Arithmetic Logical Unit is also used to describe the SSE functionality [100]. The SSE specification has evolved into providing vector arithmetic functionality with a standard known as Advanced Vector Extensions (AVX) and uses registers 256 bits in length.

Figure 5.12 below illustrates the distinction between the computation of a scalar value of 32 bit precision and that computed by the SSE consisting of a single operand required to process the (4 x 32 = 128) bit values. Although these operations can be programmed on an assembly language level, it is much more convenient and efficient to use the intrinsic compiler functions provided for that specific CPU and compiler. It must also be noted that an "out of order" read is required by the adjacent location of electromagnetic cell points, and that this will have a slowing effect on processing the FDTD using SSE.

The FDTD code has been programmed with single precision floating point arithmetic (32 bits) in order to fit the greatest number of floating point numbers into an SSE register (128 bits). Considering that the SSE will allow SIMD calculations with four single precision floating point numbers simultaneously, we would expect a speedup of approximately times over an operation using only the conventional one scalar value. Figure 5.12 above shows a comparison of how single value instructions and vector value instructions access memory and operate. The theoretical speedup produced by making use of the SSE vector registers should therefore be in the order of four times per core. In practical terms, this speedup is very dependent on the correct alignment of data with the length of the vector processing registers (128 bits) and the speed with which data is manipulated in and out of the 128 bit SSE buffers.

Stellenbosch University http://scholar.sun.ac.za

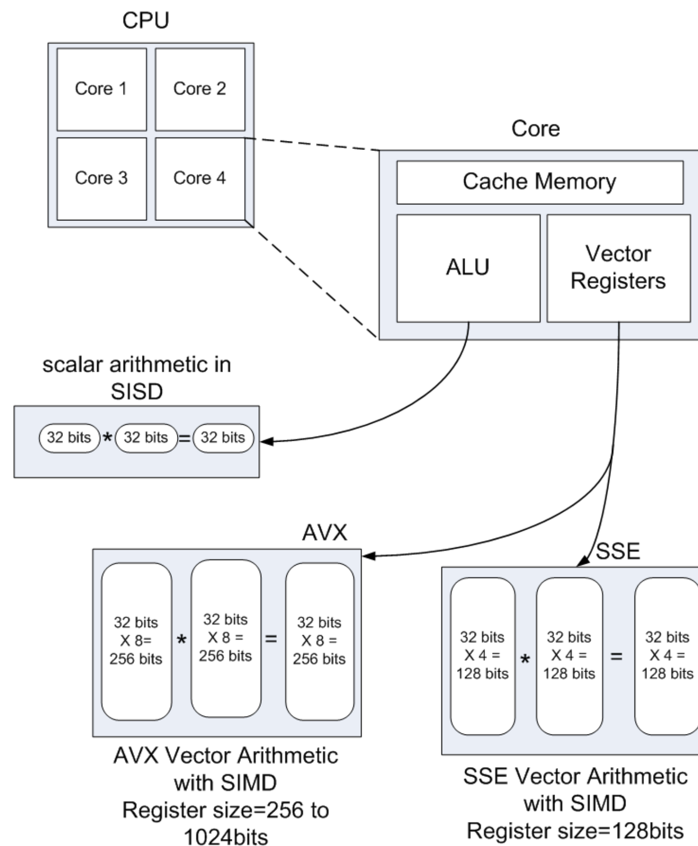

Figure 5.12: Schematic showing the two fold advantage that AVX has over SSE (adapted from Yu et al. [100])

The practical aspects of this are shown in the snippet of pseudo-code for the calculation of the electric field by the FDTD using the vector ALU mechanism:

Code segment 5.6.1:

```
1) For i=0 to end in steps of 4
2) Load 4 scalar values of electric, magnetic and coefficients into the
   VALU }                                                intrinsic SSE
3) Calculate 4 values of electric simultaneously with single SSE
   instruction                      }                    intrinsic SSE
4) Next  4 values in array
```

Instead of calculating the scalar values of an array with a simple unit incremented loop, a stepped loop is used to perform a single instruction on four values simultaneously. Figure 5.13 below shows a schematic of three threads used to implement the parallel FDTD with the openMP library. Each thread is again accelerated using the SSE functionality, the SIMD operation on four cells being shown by the rectangular bars.

 Stellenbosch University http://scholar.sun.ac.za

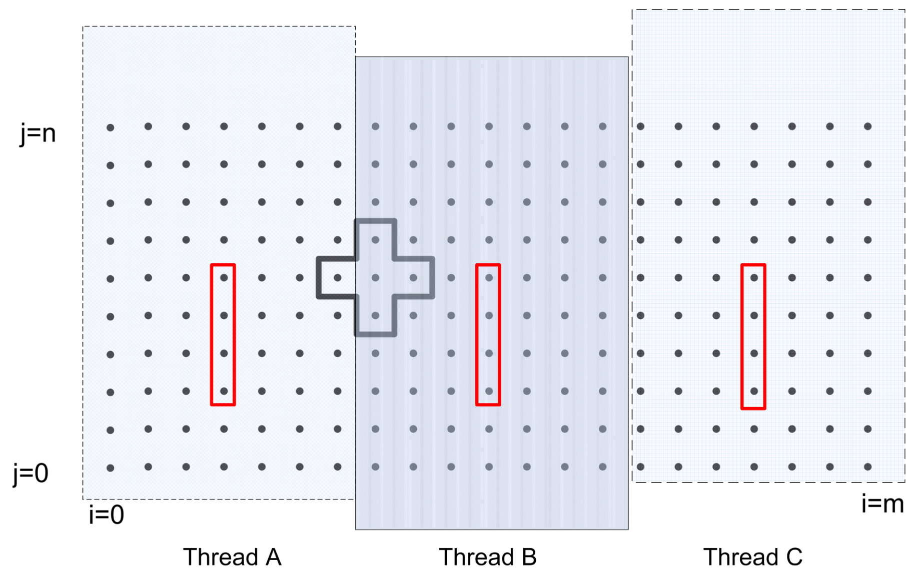

Figure 5.13: The parallel FDTD implemented with FDTD showing a snapshot of the cells subject to SIMD processing by the SSE registers within each thread.

In practice this potential fourfold speedup is not easily achieved as the movement of data in and out of the registers needs to be highly optimised to reduce the memory access latency and memory channel bottlenecking. In addition to this, the reading of cells in an “out of order” process is also a reason why the speedup of the SSE does not achieve its full potential speedup. Although the implementation of cache prefetching has been shown to reduce the throttling effect of bottlenecking and memory latency [100], it was found that the programming for this is complex, and this work could not sustain the elevated data throughput to any detectable level.

The parallel conversion of the FDTD using the SSE vector registers performed equally well with the openMP or the MPI libraries on an eight core Nehalem processor as shown below in figure 5.14. Together with the advent of multicore technology this implies that the every core on a multicore processor will have this vector processing functionality as provided by the SSE. This performance aggregation can be seen in the figure 5.14 below. A conventional parallel FDTD algorithm was converted so that the main processing loops make use of the SSE registers in each of the cores.

Stellenbosch University http://scholar.sun.ac.za

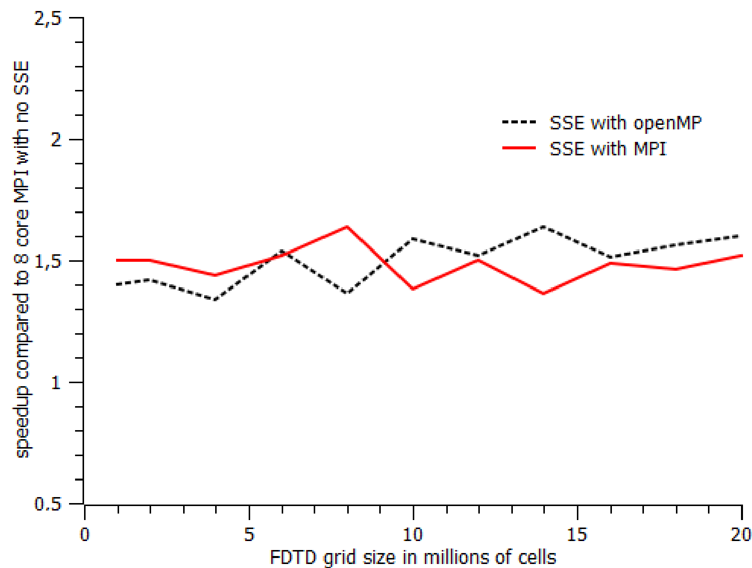

Figure 5.14: A comparison of the performance of the FDTD acceleration on an Intel Nehalem processor with eight cores using either the openMP or the MPI threading libraries.

The AVX can be implemented in a similar manner to the SSE although consideration for memory alignment has to be made for the longer register lengths. Code segment 5.6.2 shows the sections of the FDTD code used for the AVX implementation. The correct alignment of data in memory and data contiguity are key to the efficient performance of the AVX process.

Code segment 5.6.2: Lines 1 to 9 shows the definition of the 256 bit AVX variables and line 10 shows how the AVX variables are aligned on a 32 bit memory boundary.

```
1)      __m256 *aoh0;
2)      __m256 *aoh1;
3)      __m256 *aoh2;
4)      __m256 *aoh3;
5)      __m256 *aoh4;
6)      __m256 *aoh5;
7)      __m256 *aoh6;
8)      __m256 *aoh7;
9)      __m256 *aoh8;

10)       pointer = (float **)_aligned_malloc(imax * sizeof(float *),32);
```

Code segment 5.6.3: Lines 11 to 35 shows the FDTD processing loop used for the calculation of electric field values. The instructions prefixed with __mm256 are the AVX instructions used to add, multiply, or shift the FDTD coefficient and grid data, `ie` and `je` are the sizes of the FDTD grid.

```
11)       for (i = 1; i < ie; i++) {
12)       for (j = 0; j < je; j+=8) {

13)                   aoh0=(__m256 *)&ex1[i][j];
14)                   aoh1=(__m256 *)&caex[i][j];
15)                   aoh2=(__m256 *)&cbex[i][j];
16)                   aoh3=(__m256 *)&hz1[i][j];
17)                   aoh5=(__m256 *)&ey1[i][j];
18)                   aoh6=(__m256 *)&caey[i][j];
```

Stellenbosch University http://scholar.sun.ac.za

```
19)                     aoh7=(__m256 *)&cbey[i][j];
20)                     aoh8=(__m256 *)&hz1[i-1][j];

22)                     hz_shift=&hz1[i][j-1];
23)                     amDst=(__m256*)hz_shift
24)                     amTar=_mm256_loadu_ps(amDst);

26)                     am1 =  _mm256_mul_ps(*aoh0,*aoh1);
27)                     am2 =  _mm256_sub_ps(*aoh3,amTar);
28)                     am3 =  _mm256_mul_ps(*aoh2,am2);
29)                     am4 =  _mm256_add_ps(am1,am3);

30)                     am5 =  _mm256_mul_ps(*aoh5,*aoh6);
31)                     am6 =  _mm256_sub_ps(*aoh8,*aoh3);
32)                     am7 =  _mm256_mul_ps(*aoh6,am6);
33)                     am8 =  _mm256_add_ps(am5,am7);

34)                     _mm256_store_ps(&ey1[i][j],am8);
35)                     _mm256_store_ps(ex1[i]+j,am4);
```

A comparison of speedup of the parallel FDTD using the SSE or AVX functionality on the four cores of the Intel i5-2500k is shown below in figure 5.15. The reduction in performance of the FDTD with AVX on the four cores of the multicore processor is very probably owing to memory bottlenecking. The memory bottlenecking can be alleviated by making use of the multiple physical memory channels providing data to the processor [18], as is shown in figure 5.17.

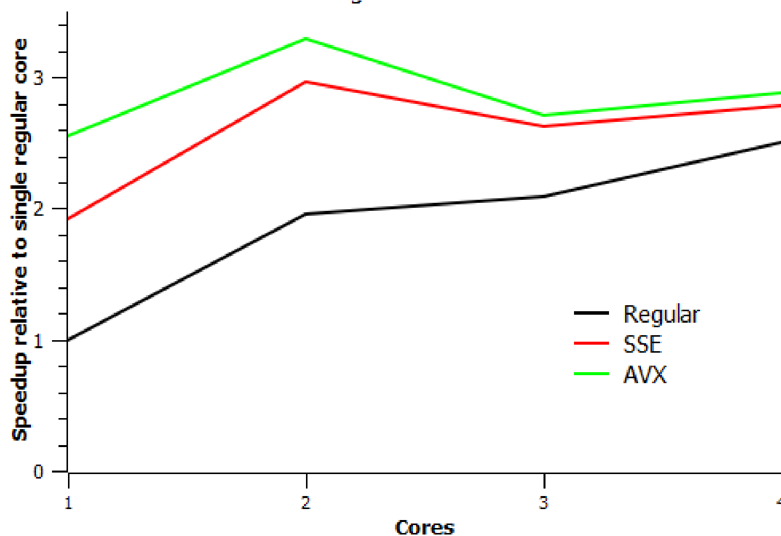

Figure 5.15: A comparison of the parallel FDTD using the MPI libraries for acceleration with both the SSE and AVX using a single physical memory channel. The FDTD model size is 20 million cells.

Although some compilers provide automated vectorisation, this optimisation provided is not yet as efficient as manually coding the VALU functionality as is shown in figure 5.16.

Stellenbosch University http://scholar.sun.ac.za

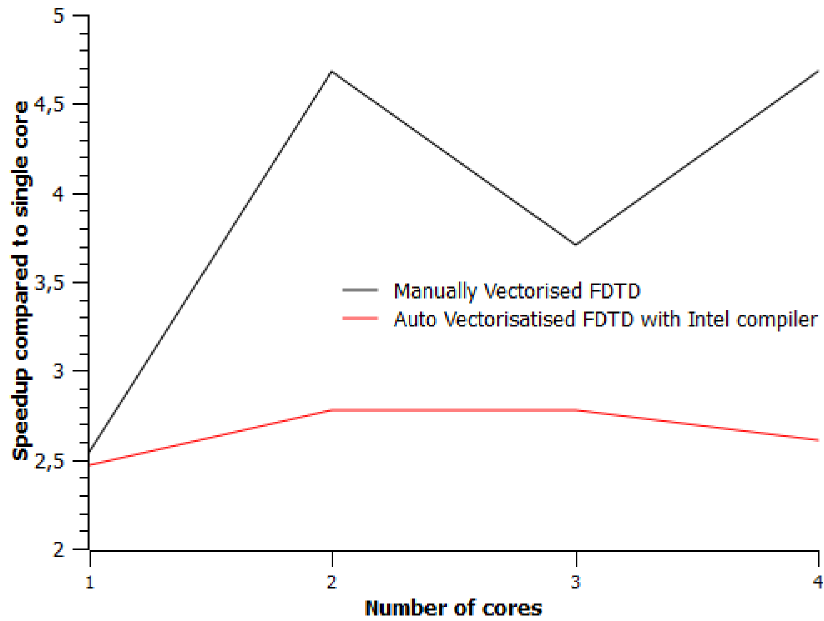

Figure 5.16: The speedup of the FDTD using AVX (256 bit registers) functionality on a four core processor for a grid size of 20 million cells. Task parallelism was achieved with the openMP libraries.

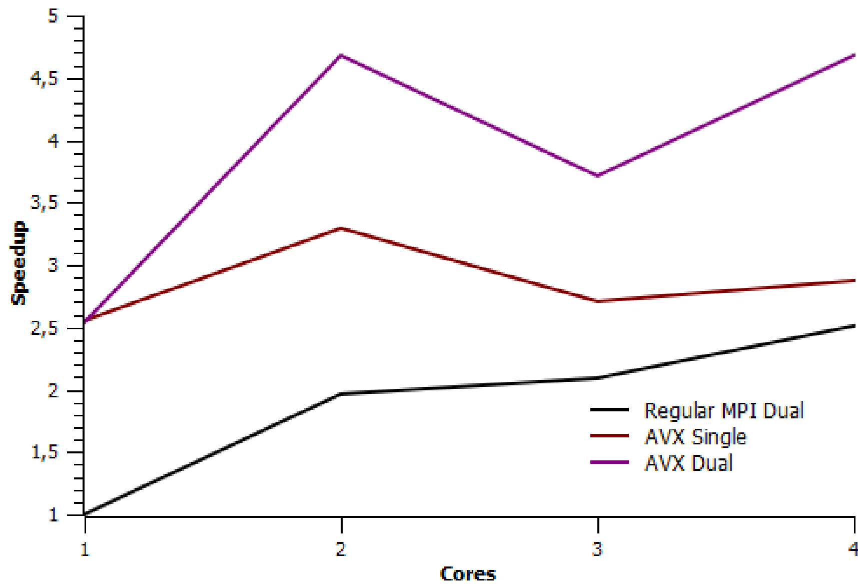

Figure 5.17: A comparison of the parallel FDTD implemented with MPI and AVX showing the improvement in performance when using dual physical memory channels. The FDTD model size is 20 million cells.

5.7 Parallel FDTD for the GPU using CUDA

CUDA is a proprietary programming interface for Nvidia GPUs. It consists of functionality that allows the manipulation of data and processes between the CPU and GPU, and on the GPU itself. Important

Stellenbosch University http://scholar.sun.ac.za

to the mapping of the parallel FDTD on to the GPUs architecture, is the hardware architecture of the GPU, shown in outline in figure 5.18 below. Figure 5.18 shows a comparison of the basic CPU and GPU structure. A major difference is that the CPU only has four processing cores but the GPU has many. Although the CPU core has an instruction set that is more diverse in logic than the GPU core, the CPU and GPU are both able to process a similar set of scalar arithmetic instructions. GPU threads are excellent for processing arithmetic calculations such as addition, multiplication, division and subtraction, but are not as efficient as CPUs for the processing of conditional instructions such as "IF" statements. This is one of the principal differences between the processing flow on the CPU and on the GPU. Although there is an aspect of SIMD processing to the implementation of the FDTD on the GPU with CUDA, Nvidia's description of the process as being Single Instruction Multiple Threads (SIMT) is more appropriate as all threads in a Streaming Multiprocessor (SM) will process the same instruction. A key concept of CUDA programming is the "warp". The warp is a group of 32 threads that process instructions in lockstep and is the smallest unit of work issued by a GPU.

As the parallel FDTD implementation strategy for the GPU described here is very dependent on the GPU architecture, a brief description of this is given below.

5.7.1 CUDA GPU architecture

In CUDA GPU terminology, the GPU is a hierarchical architecture made up of several vector processors called Streaming Multiprocessors (SM). Each SM contains an array of processing cores termed Stream Processors (SP) which are managed under the collective of the streaming multiprocessor. As a comparison to contemporary processors such as Intel's eight core Nehalem CPU, Nvidia's GTX 480 has 15 streaming multiprocessors, each containing 32 stream processors. That is, the GTX 480 has available 480 processing cores (stream processors). All of these processing cores have access to the global GPU memory via the GPU's memory interconnect. All of the cores within a streaming multiprocessor collective also have access to a type of memory with very low access latency, known as shared memory.

All of the threads in one streaming multiprocessor will perform the same instructions. Nvidia has coined the term SIMT (Single Instruction Multiple Thread) for this. The GPU supports the processing of many more logical threads than there are SPs (stream processors or cores) available and the distribution of these logical threads on to physical cores is determined by the GPU's thread scheduling mechanism. Allowing the thread scheduler to manage many more threads than there are SPs (cores) is an attempt to reduce the effect of memory access latency, an important consideration when allowing many cores to access the global memory via the SMs.

 Stellenbosch University http://scholar.sun.ac.za

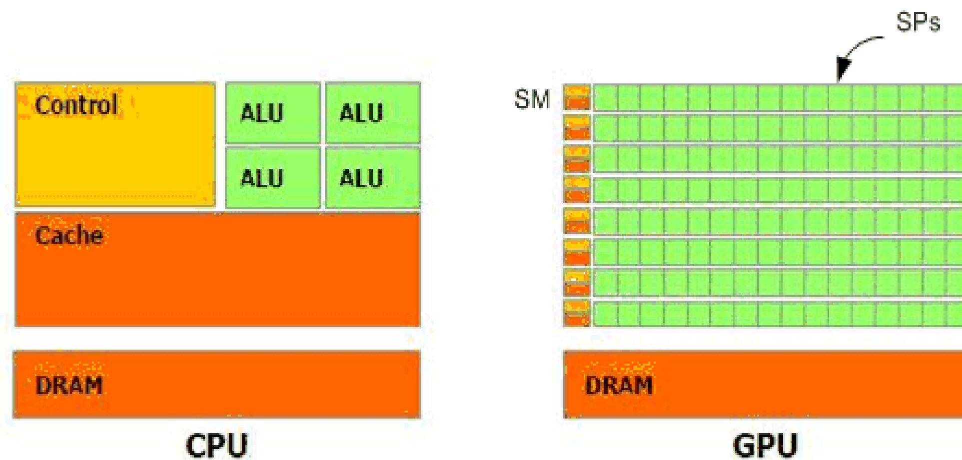

Figure 5.18: The cores and memory associated with a GPU (derived from Nvidia programming manual [82])

5.7.2 Parallel FDTD model on the GPU

Figure 5.19 shows the processing flow of the parallel FDTD on the CPU and the GPU. All of the data elements involved in the FDTD modelling have to be stored in the GPU's global memory. To take advantage of the SIMT arrangement afforded by the GPU cores, it is crucial to maintain the continuity of the array of data stored in the GPU's memory as it is presented to the threads. The relevance of this point becomes more obvious if one views the many cores of the GPU as forming a vector processing array. The communication overhead of the movement of data to the GPU's many processing threads has to be optimised in a way that is very similar to the arrangement and alignment of data consumed by the SSE and AVX vector registers, as described in sections 4.3.2 and 5.6.

The principle of achieving higher efficiency by processing continuous data elements and aligning data elements correctly in memory is not a new one, as is revealed by the inefficiency of incorrectly processing row or column dominant data on a CPU. Good memory alignment according to register length is necessary for all FDTD parallelisation methods using a massive vector processing scheme, not only the GPU.

The GPU is a case in point as the large number of processing cores (stream processors) do not have an exclusive channel to memory but are required to be supplied with data from the GPU's Streaming Multiprocessor (SM). The term used to describe the spatial ordering of data supplied to the Stream Processors (SP) in an optimised manner is termed coalescence [81]. If the data supplied to the SPs is out of order or not aligned with the SM correctly, the data is not coalesced and this will lead to long data access (latency) periods and hence low efficiency. When data is ordered in a manner that is optimally matched to the SPs, then the data is described as being coalesced and leads to low memory latency and good memory bandwidth. This in turn leads to efficient algorithmic performance on the GPU.

Stellenbosch University http://scholar.sun.ac.za

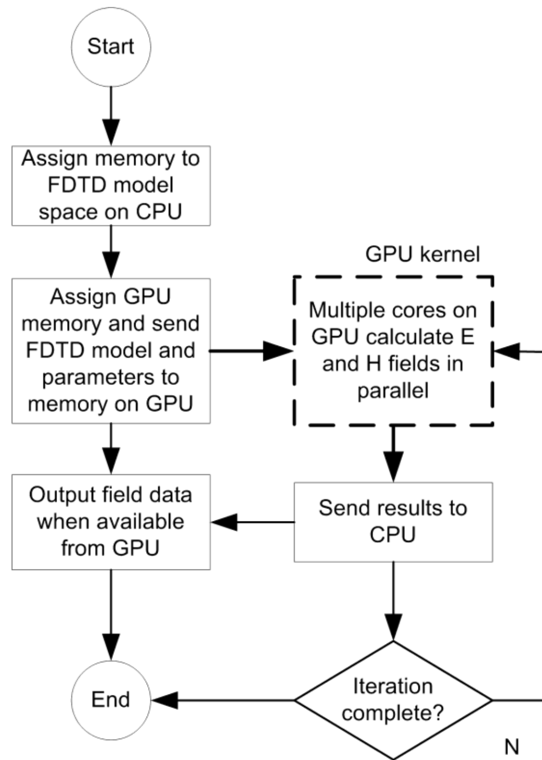

Figure 5.19: The FDTD process flow on the GPU

The CUDA code segment 5.7.1 serves as a framework to describe how the FDTD is parallelised on the GPU for the two dimensional FDTD. The code fragment is taken from a GPU kernel, which is the part of the FDTD processing flow for the GPU as is shown in figure 5.19. The GPU kernel is a segment of code built exclusively for execution on the GPU device itself. The same snippet of code is executed on every core in the GPU and will calculate the electric and magnetic field values required for one sweep of the FDTD grid, as is show in figure 5.20. As an example, a rectangular grid of cells is processed as a one dimensional array using all of the GPU cores as a quasi vector array processor. Ideally, the FDTD calculation for each cell in a column is performed simultaneously, one cell per GPU thread or core. In practical terms, the number of FDTD cells to be computed in one column will not be equal to the number of GPU cores (SP) available, and the number of threads used to compute the FDTD cells will need to be programmatically regulated such that most of the GPU cores (SP) are occupied by threads during processing. The dark rectangular area shown sweeping over the grid in Figure 5.21 represents the number of threads being processed simultaneously.

Stellenbosch University http://scholar.sun.ac.za

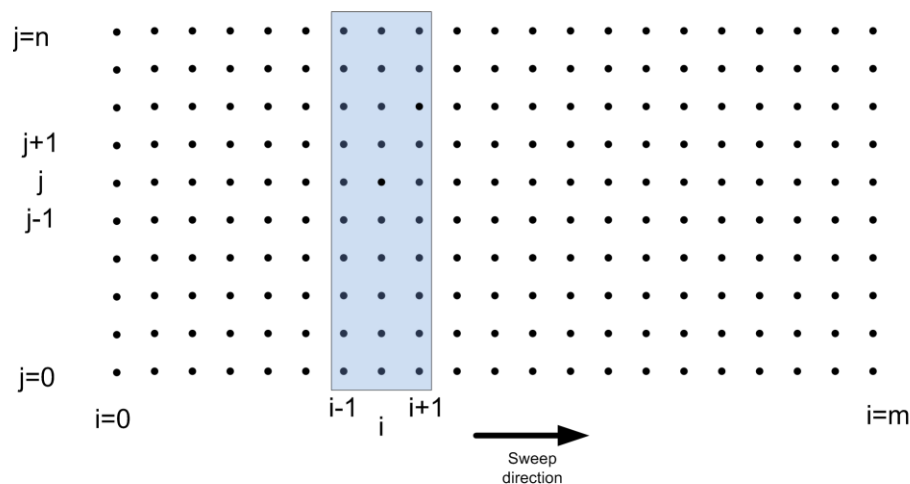

Figure 5.20: The processing sweep of threads for the 2D FDTD

The CUDA code segment 5.7.1 below represents the processing activity on the GPU kernel for one FDTD iteration cycle. The sweep width for this code is one column in width, with reference to figure 5.21. The FDTD variables and coefficients used in this code are all conventional two dimensional coordinate variables as defined in [3]. `idx` is a thread identifier and is a product of the block size (which is a thread virtualisation of the SM), the grid size specifying the number of blocks, and the thread identification per block. The values of the block and grid sizes have an effect on the performance of this kernel in that they determine the number of effective threads in an SM, also referred to as the occupancy. The conditional statements containing `blockparx` and `blockpar1` serve to define the computational limits. The variable `kdx` manages the assignment of thread and data, and is key to the performance of this code, as will be shown in the description on coalescence further below. The value `jb` is the size of a row or column, depending on the order of processing.

Code segment 5.7.1:

```
1) int idx = blockIdx.x*blockDim.x + threadIdx.x;

2) for(i=0;i<ncols;i++){

3) if (idx < (nrows-1)){
4) kdx=i*jb+idx;
5) if ((kdx < kx) && ((kdx%blockparx)<blockpar1)  && (kdx%blockparx>0)){
6)     ex[kdx]=caex[kdx]*ex[kdx]+cbex[kdx]*(hz[kdx]-hz[kdx-1]);
7)     }
8) }

9) if(idx<nrows){
10) kdx=i*jb+idx;
11)  if ((kdx%blockparx<blockpar1)&& (kdx>blockpar1)){
12)    ey[kdx]=caey[kdx]*ey[kdx]+cbey[kdx]*(hz[kdx-blockparx]-hz[kdx]);
13)  }
14)  }
15) }

16) __syncthreads();

17) for(i=1;i<ncols;i++){
18)   if(idx<nrows){
19)   kdx=i*jb+idx;
```

Stellenbosch University http://scholar.sun.ac.za

```
20)   hz[kdx] = dahz[kdx] * hz[kdx] + dbhz[kdx] * ( ex[kdx+1] - ex[kdx] +
          ey[kdx]- ey[kdx+jb] );
21)   hz[sourcepoint]=source[kloop];
22)
23)}
24)}
25)}
```

Three requirements for optimising the performance of this GPU code arrangement are:

1) Optimise the number of logical threads being processed in comparison to the physical cores so that the ratio of occupied cores is large.
2) Ensure that the memory being accessed is "coalesced", i.e. the memory accessed by threads must be arranged in a contiguous manner and aligned with the streaming multiprocessor processing it. This very issue has the greatest effect on the processing efficiency of the FDTD on the GPU, as is shown in figure 5.24.
3) Keep the number of conditionals to a minimum. The use of "if" statements leads to logical divergence in the warp and this can impact efficiency. The code kernel shown above was tested by commenting out the "if" statements and this was found to have little effect on the performance.

A simplistic schematic representation of coalescence is shown in the example below.

Consider a two dimensional matrix of data values:

0 1 2 3

4 5 6 7

8 9 a b

where 0 1 2 3 4 5 6 7 8 9 a b is stored in consecutive memory locations. If the GPU thread requires 3 data access cycles then a continuous memory allocation of data to the cores, as is shown in figure 5.23a, would be:

	Access 1	Access 2	Access 3
Core 1	0	4	8
Core 2	1	5	9
Core 3	2	6	a
Core 4	3	7	b

where Access 1, Access 2, and Access 3 are successive read cycles by the respective core.

The GPU threads are allocated a portion of data that is aligned with the size of the physical core structure in a streaming multiprocessor. The data can be supplied to the cores within a minimum number of read steps. In a simplistic sense, the data would be coalesced. Algorithmically, the GPU will try to coalesce data by using the following protocol [155]:

Stellenbosch University http://scholar.sun.ac.za

1) For each memory transaction, find the memory segment that contains the address requested by the lowest numbered thread.
2) Find all threads whose requested address is in this segment.
3) Reduce the segment size if possible.
4) Repeat the above process until all threads in a half-warp are served.

Consider that the FDTD data storage is aligned in a vertical sense on a grid pattern, so that a series of vectored processors would process as one continuous sequence of data points, as is shown in figure 5.21.

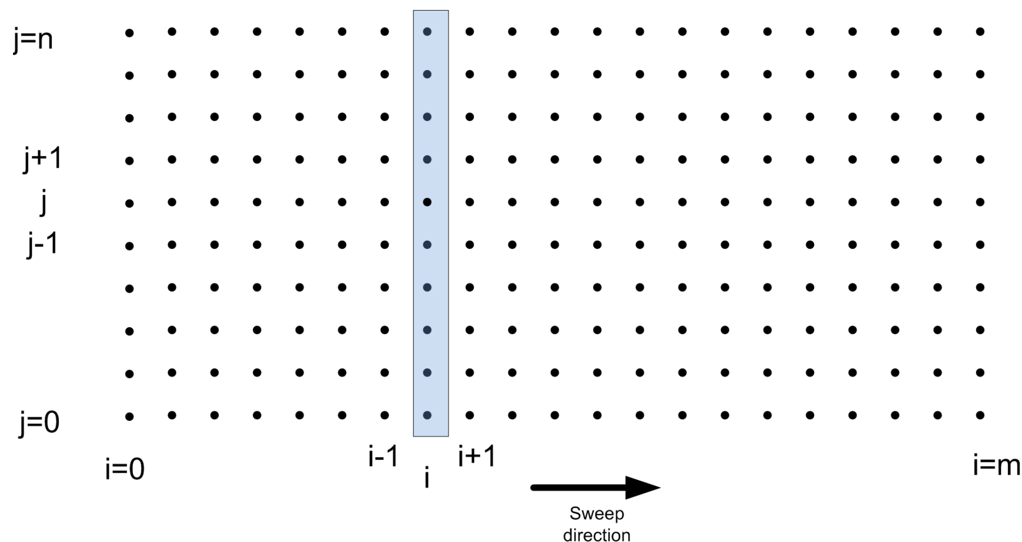

Figure 5.21: The processing sweep for the FDTD which is one column in width.

According to the extract of code shown below, consecutive GPU threads would read adjacent memory locations, assigned by the array identifier `kdx=i*jb+idx`, where `jb` is the size of the column in this instance.

Code segment 5.7.2:

```
1) for(i=0;i<ncols;i++){

2)  if (idx < (nrows-1)){
3)    kdx=i*jb+idx;
4)    ex[kdx]=caex[kdx]*ex[kdx]+cbex[kdx]*(hz[kdx]-hz[kdx-1]);
5)      }
6) }
```

If however the data is allocated to the cores in the following read order:

	Access 1	Access 2	Access 3
Core 1	0	1	2
Core 2	3	4	5
Core 3	6	7	8
Core 4	9	a	b

Stellenbosch University http://scholar.sun.ac.za

which would require several out of order read operations for the memory, as is shown in figure 5.23b. The memory access would not be coalesced and be very inefficient for the GPU or other SIMD based processors. This example is an over simplification as the addressing requirements are aligned to various memory boundaries in order to achieve proper efficiency [24, 81, 86]. A corresponding GPU kernel code snippet that illustrates the concept of uncoalesced data access is shown as code segment 5.7.3:

Code segment 5.7.3:

```
1) for(i=0;i<nrows;i++){

2)   if (idx < (ncols-1)){
3)   kdx=idx*(jb + i) ;
       ex[kdx]=caex[kdx]*ex[kdx]+cbex[kdx]*(hz[kdx]-hz[kdx-1]);
5)          }
6)   }
```

Although data is stored in the same column format as indicated in the previous example, the FDTD grid space is now processed with a row-by-row sweep, as is shown in figure 5.22. The threads allocated to a particular data element is assigned by the array identifier `kdx=idx*(jb + i)`. Using this identifier, consecutive threads are allocated data elements separated in memory by an address space of `jb*`(size of floating point value). This pattern of memory access by consecutive cores is described by figure 5.23b.

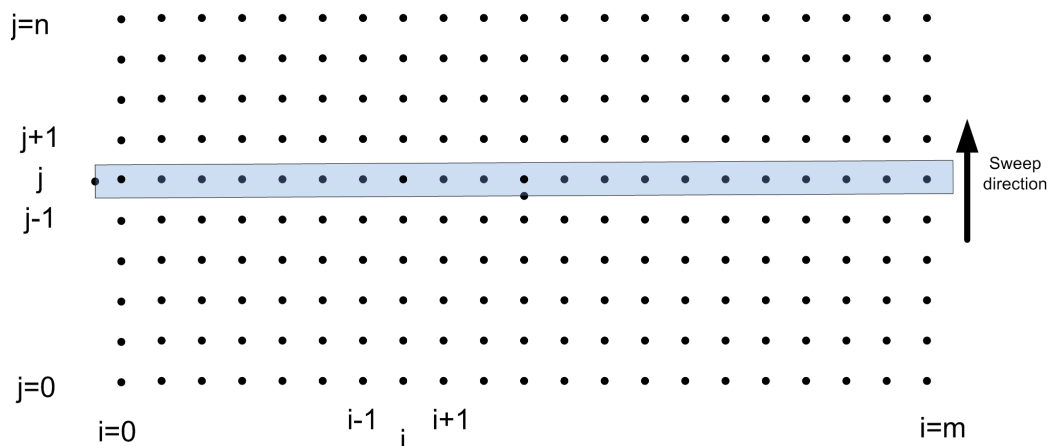

Figure 5.22

The result of deploying the FDTD on the GPU in an optimised state is shown in figure 5.24.

Stellenbosch University http://scholar.sun.ac.za

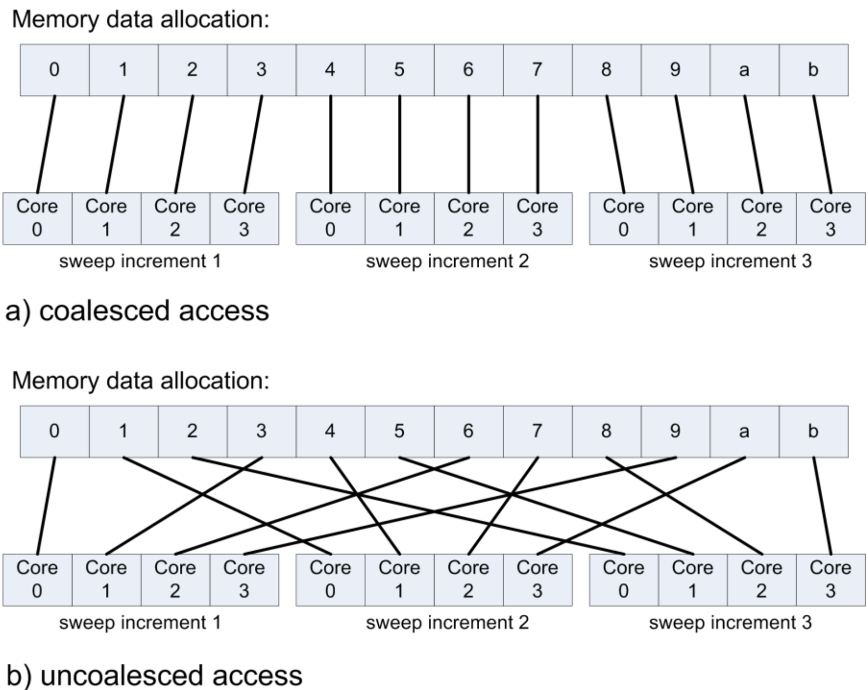

Figure 5.23: Simplified schematic of coalesced and un-coalesced memory access. Adapted from [81, 82].

A further improvement in GPU processing efficiency can be attained by reducing the memory access latency by making use of the shared memory variables available in every SM [27, 56]. The cache memory on a CPU has the same relationship to the CPUs cores as the shared memory provides to the Stream Processors (SP) belonging to one Streaming Multiprocessor (SM). The shared memory has very low latency and is limited in size when compared to the global memory. Another algorithmic consideration is that only cores within the same SM can access the shared memory belonging to that SM. The memory to thread alignment and memory continuity criteria required to achieve coalescence for the GPU using the global memory, now becomes even more stringent as the number of threads involved in one sweep is limited by the amount of shared memory available.

Figure 5.24 compares the FDTD throughput of Nvidia's GTX 480 gaming GPU against Nvidia's C2070 purpose built general purpose GPU numerical processor. The GTX 480 shows results for an FDTD grid of 20 million cells as it only has access to 1.5 GB of memory, opposed to the C2070 which has access to six GB of dedicated memory.

The computation of the split field boundary conditions are shown in CUDA kernel code segment 5.7.4, which represents the calculation of the left side PML of a two dimensional FDTD grid. The program variables in code section 5.7.4 are:

`idx,kdx,pml_idx:` GPU thread and array identifiers
`iebc:` width of the PML, in this case eight cells wide
`eybcl, hzybcl,hzxbcl:` split field electric and magnetic PML components
`ey,hz:` component of the electric and magnetic fields
`caeybcl,cbeybcl,caey,cbey:` coefficient values.

 Stellenbosch University http://scholar.sun.ac.za

The most complex programming task in converting this split field PML to the GPU is line 7 of the code segment 5.7.4, because it interleaves the main electromagnetic FDTD grid values `ey` and `hz` with the split field PML values `hzxbcl` and `hzybcl`. As all the data values have been stored as one dimensional arrays to facilitate optimised processing on the GPU, as described in the text above, the programmatic alignment of these arrays is quite delicate.

On reflection of this experience, it would have been simpler to code the FDTD for the GPU with a technique such as the CPML [3, 27,163], which does not require the computation of discrete boundary areas. According to [163] the CPML technique will also provide better absorption characteristics and simulation results.

Code segment 5.7.4:

```
// left PML
1)   if((idx < iebc) && (idx > 1)){
2)     for(j=0;j<je;j++){
3)     kdx=idx*jb+j;
4)            eybcl[kdx] = caeybcl[kdx] * eybcl[kdx] - cbeybcl[kdx] *
       (hzxbcl[kdx] + hzybcl[kdx] - hzxbcl[kdx-jb] - hzybcl[kdx-jb] );
       }
       }
// EY for i=0
5)    if(idx < je){
6)    pml_idx=(iebc-1)*jb+idx;
7)    ey[idx] = caey[idx] * ey[idx] - cbey[idx] * (hz[idx] - hzxbcl[pml_idx]
-      hzybcl[pml_idx]);
     }
```

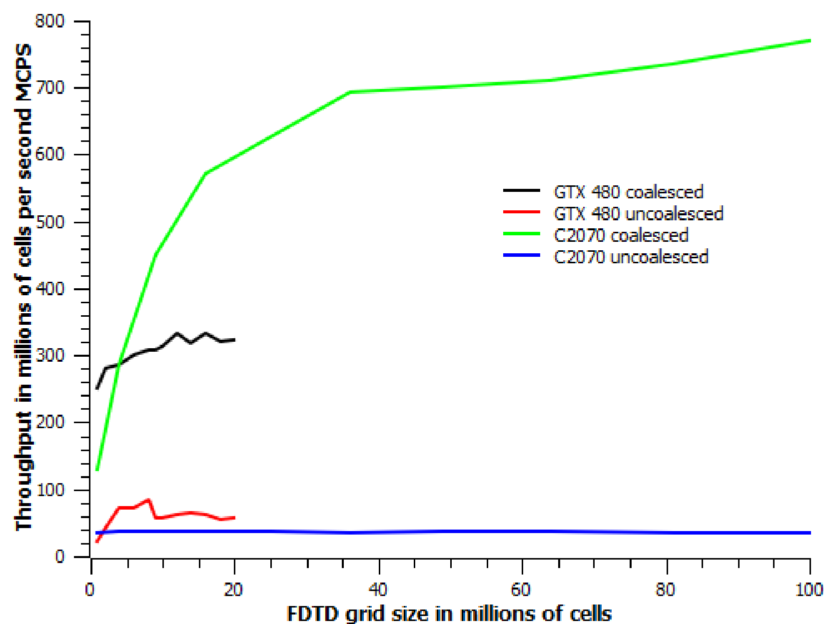

Figure 5.24: The effect of coalesced memory with the FDTD algorithm on two different GPUs

Stellenbosch University http://scholar.sun.ac.za

5.8 Parallel FDTD with Low Level threading

The basis of the threaded operation for both openMP and MPI is provided by generic threading functionality such as that provided by the POSIX threading [146] or generic Windows threading libraries. An early implementation of the FDTD on a cluster with MPI [92] discusses the use of POSIX threads to accelerate the dual core processors used in the cluster. In a contemporary scenario this FDTD hybrid deployment would very probably be implemented using a combination of the MPI and openMP threading libraries.

A parallel form of the FDTD, very similar in operation to either the openMP or MPI described above, can be created using the basic threading libraries by spawning threads to process segments of the electric and magnetic fields on a shared memory or NUMA architecture. The communication of data between threads and the synchronisation of thread events would need to be devised specifically for that program. The low level programming overhead for this exercise would be substantial and complex, and as it is already provided by application programming interfaces such as openMP and MPI, it will not be re-examined here.

Although the higher level threading methods with openMP and MPI are very popular, the existence and availability of basic threading needs to be mentioned as it will allow for the parallelisation of the FDTD on hardware platforms, such as the Android phone, where the MPI or openMP are not available.

5.9 Parallel CUDA on a cluster of multiple GPUs

Figure 5.25 is a system architecture [153] that illustrates the configuration of multiple GPUs attached to a worker node or host PC. The multiple GPUs are attached to the worker node via a dedicated Nvidia switch which uses a single PCIe form factor. The CHPC achieves a similar architectural configuration although individual GPUs are attached to the worker node by dedicated PCIe form factors, i.e. the worker node uses multiple PCIe channels to accommodate several GPUs directly.

Stellenbosch University http://scholar.sun.ac.za

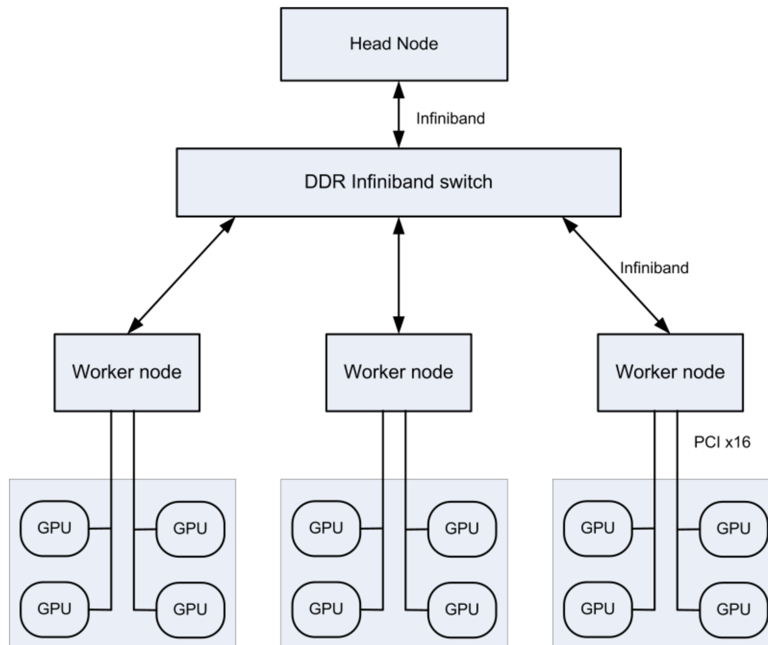

Figure 5.25: Hardware configuration using multiple GPUs, adapted from [153]

The method of distributing the FDTD on several GPUs to create a parallel FDTD system is based on a strategy that is very similar to processing the FDTD with MPI, as described in section 5.5. The FDTD modelling space is broken into several chunks and each chunk is processed on a GPU. The fringe values of the FDTD chunks are interchanged in the manner as shown in figure 5.26. On Nvidia GPUs, the GPUdirect technology allows GPUs attached to the same node to communicate data between their respective global memories directly. Nvidia's GPUdirect technology also allows the communication of fringe data between the dedicated memories of GPUs that are not on the same node.

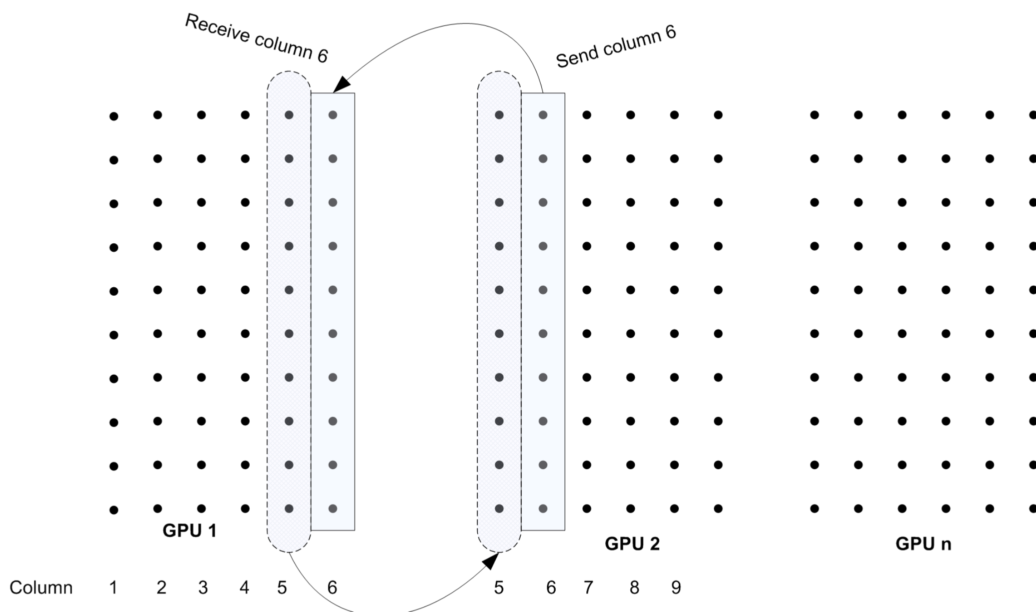

Figure 5.26: The interchange of fringe data from FDTD chunks on multiple GPUs

Stellenbosch University http://scholar.sun.ac.za

Processing results from a system of multiple GPUs in [153] has achieved a cell throughput of 7910 MCPS using eight worker nodes with a total of 32 GPUs. The system exhibits good performance scalability and is shown to process FDTD grid sizes of up to 3000 million FDTD cells using 64 GPUs. Interconnecting the worker nodes by using an infiniband switch as shown in figure 5.25 allows the easy addition of more worker nodes and GPUs and therefore makes the system highly scalable.

The process flow of the parallel FDTD on multiple GPUs is illustrated schematically in figure 5.27. It shows the basic structure of the algorithm is similar to an MPI implementation. The bulk of the FDTD computations are performed on the GPUs and MPI is used to control the process flow between worker nodes.

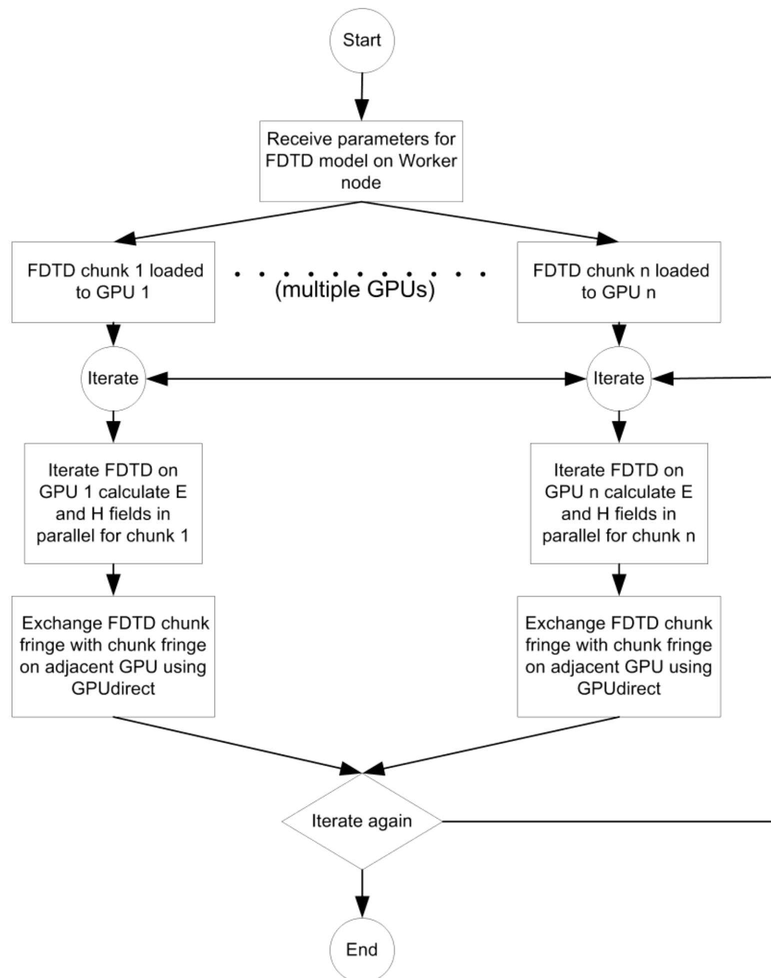

Figure 5.27: The parallel FDTD on multiple GPUs

Using the openCL interface [158] and MPI, a similar FDTD implementation on a GPU cluster [92] was found to achieve a throughput of up to 3000 MCPS using 16 GPUs. Stefanski et al [92] show that the communication overhead between GPUs is proportional to the area of the chunk fringe data that is exchanged. The hardware configuration used for this implementation uses a large number of GPU to CPU ratio and hence the increased ratio of computation to communication latency allows more efficient processing of the FDTD. The openCL technique also uses several GPUs in parallel, although still attached to a single host or worker node [129].

Stellenbosch University http://scholar.sun.ac.za

5.10 Parallel FDTD on the GPU using Direct Compute

The Direct Compute functionality is an interface for General Purpose GPU (GPGPU) computations that is incorporated as part of the suite of Microsoft's DirectX11 graphics APIs. GPU program threads are executed on physical cores referred to as compute shaders. Although the compute shaders have access to graphics resources, the execution of the compute shader is not attached to any stage of the graphics pipeline. The overall programming paradigm is very similar to that of CUDA in that the GPU is treated as a discrete processor and the FDTD grid data is transferred to the dedicated memory of the GPU for processing. An advantage of using Direct Compute is that applications derived from this API will function on GPUs created by manufacturers other than Nvidia, particularly Intel, whose GPU products are currently the most prolific in the global PC market. A combination of conventional C language and a special "C" like scripting language is used to program the FDTD on the GPU hardware. For Direct Compute the scripting language is called Higher Level Scripting Language (HLSL) [96]. The FDTD algorithm is functionally decomposed in the same way as for CUDA in that the multidimensional grid data is presented to the GPU for SIMD processing. In this manner many FDTD threads are processed concurrently by the Stream Processors (SP) or cores in the GPU. As with CUDA, the continuity of data is crucial for optimised performance of the FDTD on the GPU. A code snippet of a single two dimensional FDTD processing iteration is shown in code segment 5.10.

Code segment 5.10:

C control program code(Repeat for each FDTD iteration):

```
1) RunComputeShader( Parameter list including number of rows and columns of
   FDTD space );
```

HLSL code:

```
2) int kdx=idx*ROWS+jdx;
3) int kdx1=idx*ROWS;
4) static int kloop=0;
5) kloop==kloop+1;

6) if(kdx != kdx1){
    A[kdx].ex = B[kdx].a*A[kdx].ex + B[kdx].b*(A[kdx].hz - A[kdx-1].hz);}


7) if(kdx>ROWS){
    A[kdx].ey = B1[kdx].a*A[kdx].ey + B1[kdx].b*(A[kdx-ROWS].hz -
            A[kdx].hz); }


8) A[kdx].hz = B1[kdx].c*A[kdx].hz + B1[kdx].c*(A[kdx+1].ex - A[kdx].ex +
            A[kdx].ey - A[kdx+ROWS].ex  );

9) A[source].hz = B2[kloop].a;
```

```
where:

kdx is the thread identifier
.ex,.ey,and .hz are the FDTD electric and magnetic cell values .a,.b, and
.c are the FDTD electric and magnetic coefficients described in [3]
```

Stellenbosch University http://scholar.sun.ac.za

The code titled as "`C control program`" code is the statement that runs the graphics kernel described by the HLSL code. A thread is launched to process one FDTD cell per thread. All FDTD grid data and related coefficients have been loaded into the GPU's dedicated memory before initiation of this process. It should be noted that one of the differences to the implementation of the CUDA specific FDTD for this work is that the Direct Compute version is customised specifically for the HD3000 GPU found on Intel's i5-2500k processor. Once completed, the Direct Compute version of the FDTD functioned adequately on an Nvidia GPU with 96 CUDA cores. A comparison of the performance of the Direct Compute FDTD algorithm on Intel's integrated HD3000 GPU and on Nvidia's 96 core GT 240 GPU is shown in figure 5.28.

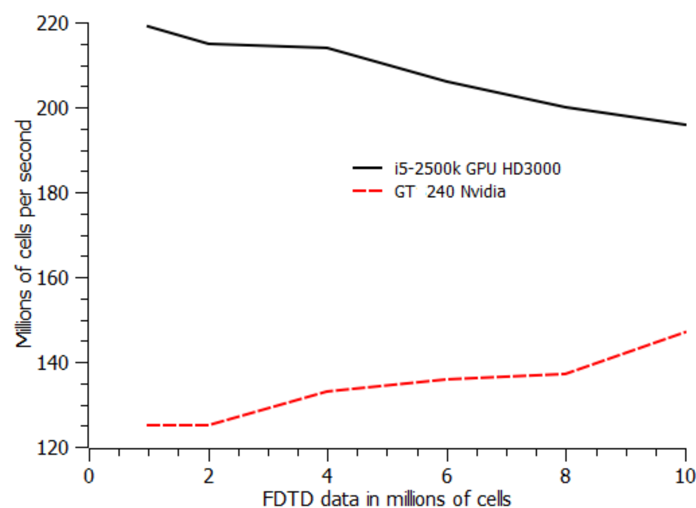

Figure 5.28: A comparison of the performance of the FDTD algorithm deployed using the Direct Compute on GPUs from two different manufacturers

The Direct Compute based FDTD performs well on the integrated HD3000 GPU, although the internal structure of the 12 Graphical Execution Units (EU) is enigmatic as Intel does not disclose what the internal hardware architecture of the EU is. It is known that the graphical EU is composed of a system of compute shaders (or Stream Processors (SP) in CUDA terminology) but it is not known precisely how many shaders there are and how these are physically configured.

Figure 5.24 shows the FDTD through using the Direct Compute API on GPUs from two different manufacturers. The HD 3000 GPU is integrated to the Intel i5 processor and shares the same physical memory as the four processing cores as is described in section 4.3.2. The maximum available memory to the HD 3000 GPU as used for these tests is 820 MB. The GT 240 is a discrete Nvidia GPU with 96 stream processors and has one GB of dedicated memory.

The HD3000 shows a degradation of performance with an increase in the size of the FDTD grid being processed, possibly owing to the influence of the Windows operating system. Although the general FDTD cell throughput is not as good on the discrete GT 240 GPU as on the integrated HD 3000 GPU, it does not experience the same performance degradation for larger FDTD grids sizes. This behaviour is possibly owing to more physical memory being available to the GT 240 than to the HD3000.

Stellenbosch University http://scholar.sun.ac.za

The objective of the FDTD implementation on the APU was to allow concurrent FDTD processing on both the internal GPU and the conventional processing cores, while accessing the FDTD grid data from a single addressable memory space. Processing the FDTD resident in one addressable memory will allow the concurrent FDTD threads on the internal GPU and cores to avoid the communication latency penalty experienced between a single discrete GPU and host computer connected via a PCIe link when interchanging FDTD data. At the time of this experiment, the hardware available was an Intel i5 four core processor of the second production generation. The only interface to GPGPU functionality on this series of Intel i5 processor was with the DirectCompute API.

It eventually was not possible to access memory space reserved for processing on the internal GPU available to the conventional processing cores directly. It was also too programmatically complex to exchange specific sections of the FDTD grid data between the GPU's memory address space and processing cores. For these reasons the experiment was abandoned on the second generation Intel i5 processor. Intel is now producing the third generation of its i5 processor which has the capability of being programmed with the openCL interface. The examination of whether the above objectives can now be achieved by using openCL would be the basis for future work.

5.11 Accelerated Processing Units and Heterogeneous System Architecture technology

The Accelerated Processing Unit (APU) is a hybrid processor consisting of conventional Complex Instruction Set Computing (CISC) processing cores and a GPU on the same processor die. The FDTD can be deployed on the CISC cores with the openMP threading library as described in section 5.4. The FDTD can also be deployed on the integrated GPU as described in 5.10 above.

This hardware arrangement has the potential to allow the concurrent processing of the FDTD on the GPU and the multiple CISC cores of the APU. By using the same physical memory to communicate FDTD grid data between the GPU and the processing cores, as is shown in figure 5.29 below, there is a reduction in the communication latency, and hence a corresponding reduction in the processing time of the FDTD when processing one FDTD grid shared between the GPU and the processing cores.

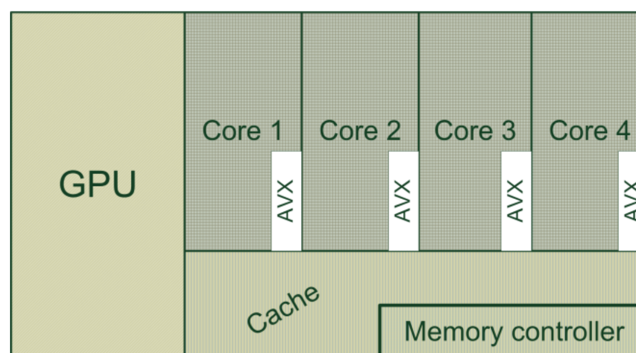

Figure 5.29: A schematic of the APU die showing both the GPU and the cores on the same die

Stellenbosch University http://scholar.sun.ac.za

Although the implementation of the FDTD on the different devices available on the APU has been described above, i.e. the integrated GPU and the four CISC cores, a summary of the performance of the FDTD is shown in figure 5.29 below. An extension of this work would entail the implementation on the integrated GPU and the four conventional cores to process the FDTD concurrently and using the same physical memory.

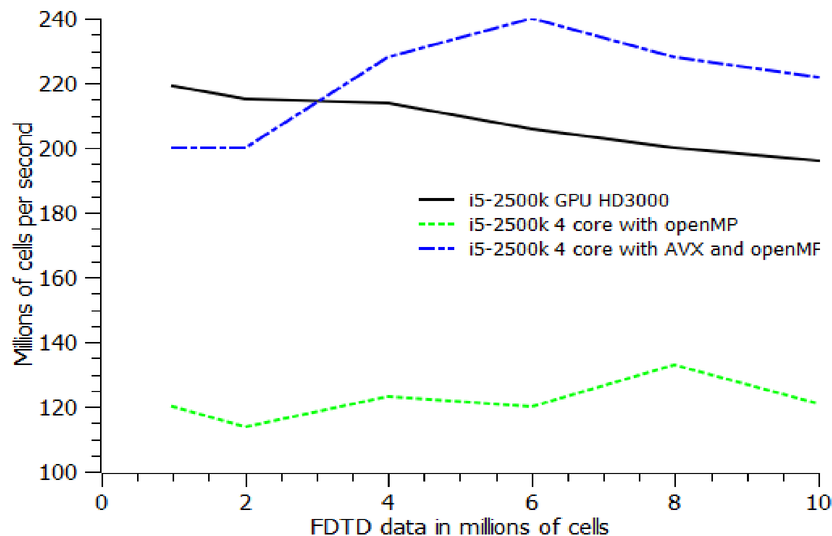

Figure 5.29: The performance of the parallel FDTD as implemented on the different APU

5.12 The parallel FDTD on the Intel Many In Core processor

Although no MIC was available for the FDTD experiments in this thesis, a short section on the MIC is included here owing to its state-of-the art features listed in the hardware description. These are in summary;

1) More than 50 cores on one processing die.
2) Extended vector processing registers to accommodate sixteen concurrent single precision floating point operations per core.
3) The novel ring like memory interconnect which uses up to 16 memory channels to supply data to the processing cores.

The MIC is a shared memory device and can the FDTD can be parallelised using a variety of threading API's such as MPI, openMP, and openCL. The MIC has the potential to process
50 cores x 16 vector floating point operations per core = 800 simultaneous floating point operations. A parallel implementation of the FDTD on the MIC [162] using the openMP threading achieves a speedup of nearly 60 times over that achieved on a single core. The FDTD cell throughput in MCPS on the Intel Phi has unfortunately not been published by Intel, although an extrapolation from the FDTD cell throughput on a single Nehalem Xeon processor, would place this at 60 x 28 MCPS = 1680 MCPS.

The code description from this article [162] also shows that use is made of the vector registers on each core by using the auto vectorisation capability of the Intel compilers, as described in section 5.6 of this thesis.

Stellenbosch University http://scholar.sun.ac.za

5.13 Summary

Several techniques that influence the implementation of the parallel FDTD method on HPC platforms have been examined. The fundamentals of how the FDTD can be deployed with the openMP and MPI threading libraries, and how these techniques apply to various architectures, has been described. It has also been shown how to implement the FDTD method as a data parallel algorithm on processors such as the General Purpose Graphical Processing Unit. The creation of more powerful hybrid HPC architectures, such as the cluster of GPUs, or the combination of AVX, GPU and multi core on the Accelerated Processing Unit (APU), require the use of several different parallelisation techniques to deploy the FDTD successfully.

Very much in the way that a modern HPC system is made up several different devices that can be used to accelerate a numerical algorithm, the associated parallel FDTD is a combination of a variety of parallelisation techniques. In order to understand the deployment effort of the parallel FDTD on a contemporary HPC platform, one needs to compare the combination of parallelisation techniques on these systems. This comparison and analysis is undertaken in the following chapter.

Stellenbosch University http://scholar.sun.ac.za

Chapter 6

6 A comparison of the FDTD method on different high performance computing platforms

6.1 Overview
6.2 Comparison of FDTD implementations on different HPC platforms
- 6.2.1 Discrete GPGPU and SMP
- 6.2.2 Discrete GPGPU and multicore SIMD
- 6.2.3 SMP and Bluegene/P
- 6.2.4 Cluster and SMP
- 6.2.5 Integrated GPGPU and discrete GPGPU
- 6.2.6 Other systems

6.3 General classification table
- 6.3.1 Scalability of process
- 6.3.2 Scalability of hardware
- 6.3.3 Memory access efficiency
- 6.3.4 Openness
- 6.3.5 Algorithm encoding
- 6.3.6 Performance in speed
- 6.3.7 Performance in terms of model size
- 6.3.8 Cost of purchase
- 6.3.9 Power consumption
- 6.3.10 Implementation time
- 6.3.11 Future proofing from aspect of coding

6.4 Summary

6.1 Overview

The main focus of this chapter is the comparison of factors contributing to the deployment of the FDTD on different High Performance Computing platforms. An evaluation is made of the implementation of specific FDTD deployments in order to identify deployment issues that were not apparent in chapters 4 and 5. The objective is to convert the subjective effects of some of the contributing factors into a quantifiable form.

A comparison of the parallel FDTD on different computing platforms needs to consider several attributes of varying scale, such as the size of the FDTD data sets being processed. A good example of this is shown in the graphic in table 6.1 below which illustrates the variation in the size of the data sets that can be processed by a single off-the-shelf GPU and the M9000 Shared Memory Processor (SMP). As this thesis is focused on the deployment of the parallel FDTD on these platforms, it is necessary to compare the effort involved in implementing the parallel algorithm on the different hardware platforms. These comparisons serve to describe the FDTD from an implementation aspect and are not meant to be only a performance comparison, although FDTD cell processing throughput is one of the categories.

Stellenbosch University http://scholar.sun.ac.za

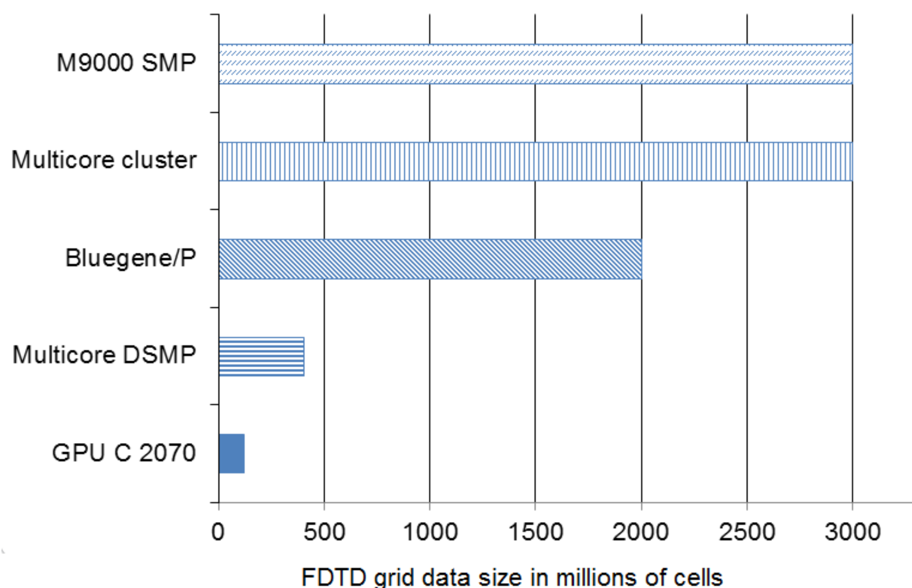

Table 6.1: A comparison of FDTD grid sizes on different HPC platforms deployed with the parallel FDTD in this thesis.

The comparison of the FDTD ranges from the traditional approach of the clusters, where parallelism is achieved by a massive task parallel approach, to the use of the refined microarchitecture of the GPU, where processing performance is achieved in a data parallel sense.

The general classification table at the end of this chapter is a summary of all the features contributing to the deployment of the FDTD on specific HPC platforms. The notes accompanying this classification are a brief description of the entries in the table and only serve as a summary of the argument presented in this thesis.

6.2 Comparison of FDTD implementations on different HPC platforms

6.2.1 GPGPU and SMP

The comparison of the parallel FDTD on the Graphical Processing Unit (GPU) and the parallel FDTD on the Shared Memory Processor (SMP) is a comparison of two very different hardware architectures, coding implementation methods and cost scales. The most obvious difference is that the FDTD for the GPU has been implemented in a data parallel sense whereas for the M9000 SMP this is done is a task parallel sense.

The discrete GPU is configured as an accelerator to the host computer and has its own dedicated memory, as is described in chapter 4. This allows the movement of FDTD grid data to the GPU accelerator for processing. The typical contemporary off-the-shelf GPU has a maximum memory of one to two GB which is a restriction to the size of the FDTD grid that can be processed on the GPU. GPUs specifically produced for the purpose of numerical processing, such as Nvidia’s C 1060 and the C 2070, are fitted with a dedicated global memory of up to six GB. This additional memory allows the processing of much larger data sets on these devices.

Stellenbosch University http://scholar.sun.ac.za

The long communication latency experienced when moving data to the host computer from the GPU, or from the GPU to the host computer, prevents the efficient renewal or update of FDTD grid data on the GPU while processing on both the GPU and host computer. The consequence of this is that all of the grid data needs to be resident in the GPU's memory if one is to optimise the FDTD algorithm for the discrete GPU. The largest model that was processed on a discrete GPU for this thesis is 121 million grid points, which is small when compared to the model sizes of over two billion FDTD grid points processed on the M9000 SMP. It must be noted that the M9000 SMP was able to compute model size of up to 20 billion FDTD grid points. When multiple GPUs are used in a cluster architecture it is possible to process data grids of the same magnitude as can be processed on the M9000, as is described in section 5.9.

The M9000 SMP architecture is a legacy architecture, where many processing cores have access to the same memory address space, and the ease of deployment of the parallel FDTD speaks of the continuous refinement this development environment has undergone over the previous two decades. The conversion of the serial three dimensional FDTD onto the SUN M9000 shared memory processing system took less than two hours and this fact in itself is a powerful differentiator when compared with the other development platforms.

The very short deployment time is a good basis of comparison for the conversion of the serial FDTD for the GPU. It took more than a week's worth of intense development work to produce basic performance results from the parallel FDTD on the GPU. The development effort was taken up by the requirement to reorganise the multidimensional FDTD grids into a one dimensional grid as a pre-requisite for processing on the GPU. From an aspect of coding, the Perfectly Matched Layer (PML) boundary conditions were difficult to deploy for the GPU as it required many hours of programming to align the boundary areas with the main FDTD grid correctly. The choice of boundary conditions in the coding process is quite crucial for the GPU implementation, as CPML boundaries are much simpler to integrate with the FDTD grid [27]. The GPU programming effort for the FDTD can be broken into 3 distinct stages:

1) Basic implementation of the FDTD on the GPU to achieve data parallelism.
2) Integration of boundary conditions with the main FDTD grid.
3) Optimisation of the FDTD on the GPU, such as memory coalescence and use of shared memory variables.

The optimisation of the FDTD on the GPU is a crucial exercise that is needed to produce good performance results. Figure 6.1 below illustrates the benefit of optimising the FDTD on the GPU by coalescing [55] the access of data with global GPU memory. This coalescence of the data has large implications for the performance achievable on the GPU and is somewhat analogous to the reading the correct data order on rows or columns used in matrix handling methods on conventional CPUs. The use of shared memory, a limited set of variables with extremely low access latency, can enhance the performance of the FDTD even more. The amount of shared memory variables available to a collection of threads on a GPU is however very limited and increases the complexity of the programming.

Figure 6.1 illustrates a cost implication of optimising the FDTD for the GPU by comparing the optimised performance of the parallel FDTD on a 120 core gaming GPU, the GTS 240, with the un-optimised performance of the 480 core GTX 480 GPU. The Nvidia GTS 240 or GTX 480 GPUs are

Stellenbosch University http://scholar.sun.ac.za

commonly purchased for enhancing the graphics of computer games on personal computers and are also commonly referred to as "gaming" GPUs. Figure 6.1 shows that the optimised form of the low cost 120 core GTS 240 GPU clearly outperforms the un-optimised and more expensive 480 core GTX 480 GPU. A detailed description of the coalescence is given in section 5.7. The Nvidia C 2070 and C 1060 GPUs can process larger FDTD data grids than the gaming GPUs as they have a larger dedicated memory. It is speculated that the artefacts in the throughput plots for the GTS 240 and GTX 480 gaming GPUs are as a result of interference by the Windows operating system, under which these FDTD implementations were developed. The FDTD implementation on the C 1060 and the C 2070 GPUs shows no such artefacts and were run in a linux environment.

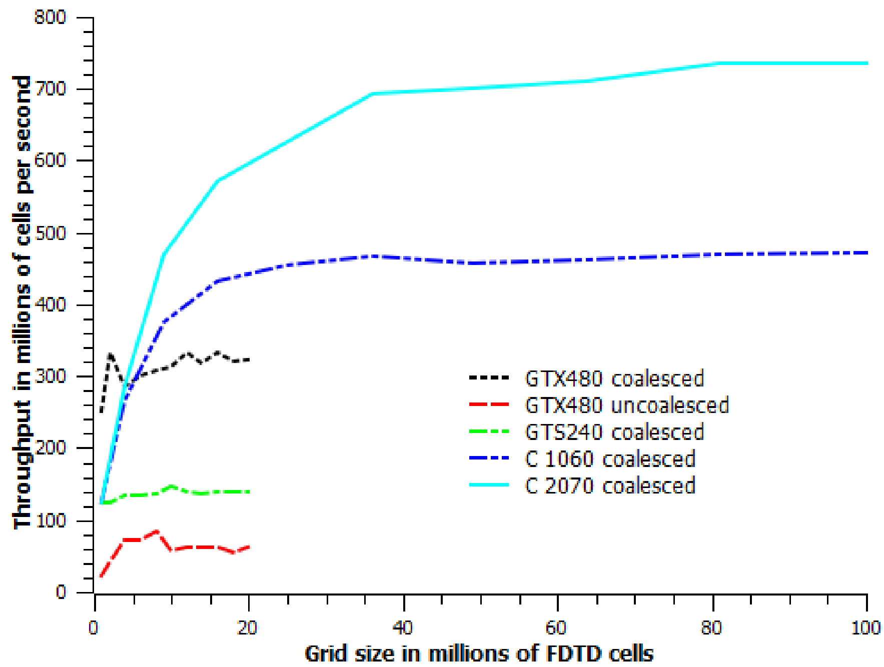

Figure 6.1: Performance of the FDTD on various GPUs for various stages of optimisation

The performance comparison of the parallel FDTD on the M9000 SMP and the GPU, as is shown in figure 6.2, was made for a range of FDTD grid sizes. The performance of the parallel FDTD using MPI on 16 cores of the M9000 is closely matched to that of Nvidia's C 1060 GPU. Similarly, the performance of the parallel FDTD with MPI on 32 cores of the M9000 is very close to that of the 448 core C 2070 GPU. The peak throughput of FDTD data by the M9000 in figure 6.2 is approximately 780 million cells per second. The same parallel FDTD deployment on the M9000 will only reach a maximum peak of 780 million cells per second when processing an FDTD grid size of two billion cells. It is possible that the cause for the parallel slowdown on the M9000, described in figure 5.3, is also the reason for this throughput ceiling in processing performance.

Stellenbosch University http://scholar.sun.ac.za

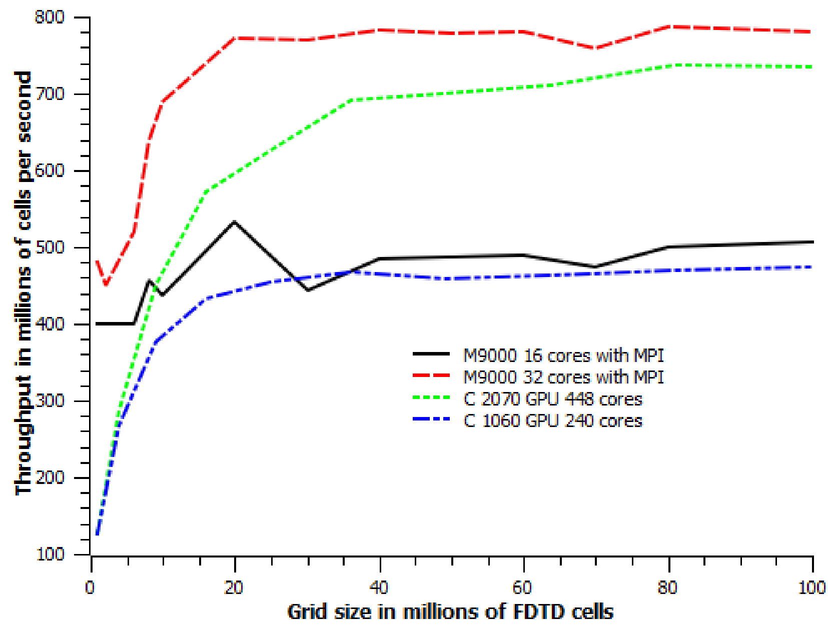

Figure 6.2: A comparison of the parallel FDTD on the M9000 SMP and the GTX480 GPU.

Figure 5.3 shows the reduction in processing time on the M9000 SMP as more cores are used for computational effort. Processing this small FDTD data set with more than 32 cores results in ”parallel slowdown”, a phenomenon caused by:

1) The increased algorithmic load of communication between the processing threads and memory.
2) Memory contention of too many threads attempting to access the same memory.
3) Processor affinity; the short period threads caused by loop parallelism force constant context changes and the reloading of cache memory in processors associated with other threads [51, 101].

The efficiency of the parallel FDTD algorithm implemented for the M9000 SMP is affected by the ratio of FDTD cell degrees of freedom to number of processing cores, also referred to as the granularity, the effect of which is described in figure 6.3. The efficiency is defined as;

$$Efficiency(\%) = \frac{Measured\ FDTD\ cell\ throughput}{Ideal\ FDTD\ cell\ throughput} \times 100$$

The efficiency of the parallel FDTD on the M9000 increases as the granularity increases, i.e. the ratio of FDTD cells computed per core increases in relation to the amount of data communicated between FDTD chunks (or threads).

Stellenbosch University http://scholar.sun.ac.za

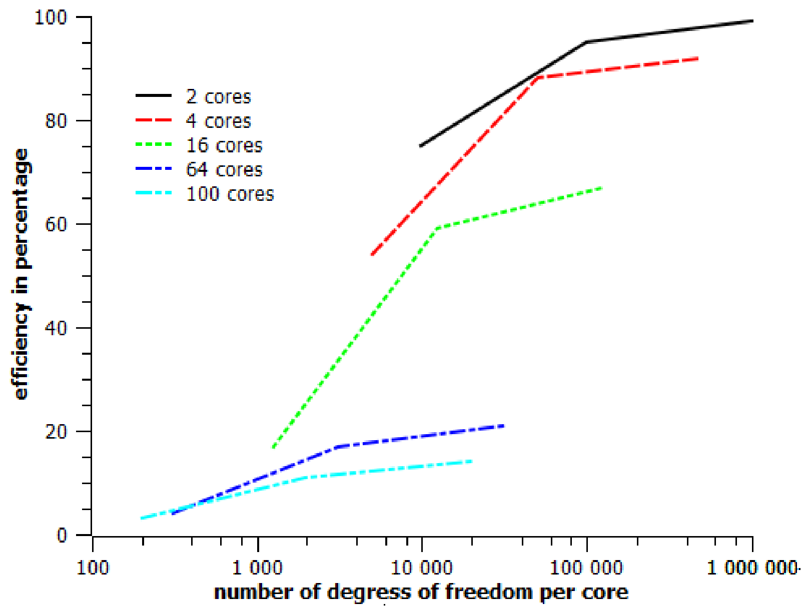

Figure 6.3: The efficiency of the parallel FDTD using the MPI threading libraries on the M9000 SMP.

The performance variation in the openMP and the MPI versions of the parallel FDTD on the M9000 platform is shown in the figure 6.4. The variation in performance can be attributed to processor affinity penalties incurred by these two different threading techniques on the M9000 [51, 101], i.e. the MPI FDTD thread has continuous access to the same physical core and memory during processing of the FDTD whereas the openMP algorithm is based on loop parallelism, which continuously creates new threads, with a related core affinity problem. This difference in parallel loop duration is schematically illustrated in the figures 5.7 and 5.10 which show the thread duration for the parallel FDTD implemented with either openMP or MPI.

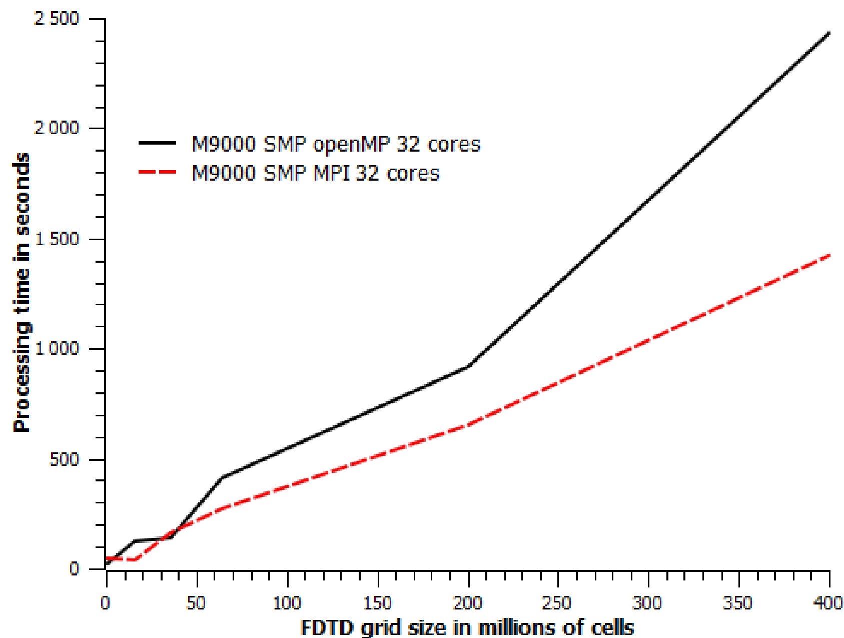

Figure 6.4: Processing time of the parallel FDTD with openMP or MPI on the M9000 SMP for varying FDTD model sizes

Stellenbosch University http://scholar.sun.ac.za

6.2.2 GPGPU and multicore vector registers

The comparison of the implementation of the FDTD on the GPU and the multicore vector registers is based on the algorithmic similarity of processing the FDTD on these platforms. The data parallel nature of both of these implementations suggests that there is also some similarity between these architectures. This similarity is a result of the Single Instruction MultipleThread (SIMT) design of the GPU Stream Processor and the use of a Single Instruction Multiple Data (SIMD) algorithm for the SSE or AVX. Both of these devices perform SIMD-like instructions to parallelise the FDTD.

Discrete GPUs in general consume more power than the multicore processors using vector extensions, such as the AVX. The Nvidia GTX 480 consumes up to 350 watt when processing intense numerical algorithms such as the FDTD. The four core Intel i5-2500k processor, including the contribution from the AVX registers and integrated GPU on the same die, consumes a maximum of 105 watt under load.

Figure 6.5 below is a comparison of the speedup achieved by implementations of the FDTD on a 240 core Nvidia C 1060 GPU and the FDTD running on an eight core Intel Xeon processor using the Streaming SIMD Extensions (SSE) registers. While the implementation of the optimised FDTD on the GPU is a relatively complex coding project, the optimisation of the parallel FDTD using the SSE capability of the multicore CPU is relatively simple, as described in section 5.6. The FDTD implemented using the SSE shows the same amount of speedup when using either the openMP or MPI threading techniques.

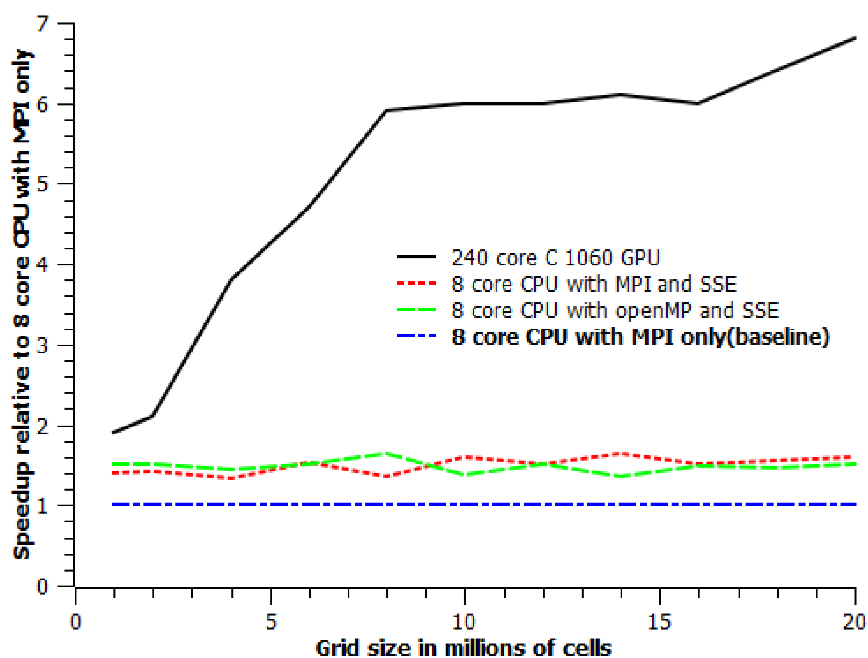

Figure 6.5: Speedup of the FDTD using SSE on an eight core CPU compared to FDTD on the C 1060 GPU.

The multicore SIMD architecture is an evolving platform in which the number of cores and the specialised circuitry such as the length of the vector registers is growing rapidly. At the time of writing of this thesis, at least one chip manufacturer is planning a 50 core chip with each core

Stellenbosch University http://scholar.sun.ac.za

incorporating the AVX functionality (Knight's Corner, from Intel). When one considers that the AVX specification allows an extension to 512 bits and that this will allow the simultaneous processing of eight single precision floating point numbers, there is a potential to process 50 x 16 = 800 floating point numbers concurrently. Table 6.2 is a table of contemporary processors and their SIMD/SIMT capability.

Device	Manufacturers identification	cores	Concurrent single floating point calculations
GPU	C 1060	240	240
4 cores with SSE - 128	intel i5-2500k	4	16
8 cores with SSE - 128	intel Xeon 7560	8	32
4 cores with AVX - 256	intel i5-2500k	4	32
4 cores with AVX - 1024	As per AVX specification only	4	128
50 cores with AVX - 512	Knights Corner (release in 2013)	50+	800

Table 6.2: A comparison of the SIMD capability of different devices.

When processing the FDTD in a data parallel sense with a vector register that is twice as long, i.e. the 256 bit AVX register as compared to the 128 bit SSE, it does not automatically follow that there will be a twofold improvement in performance. Figure 6.6 illustrates the speedup achieved on a four core processor with the FDTD using either the SSE or the AVX registers. The FDTD using AVX does show a marked improvement in performance for a low core count, but for a higher core count the performance of the FDTD using AVX is not much better than the FDTD using SSE.

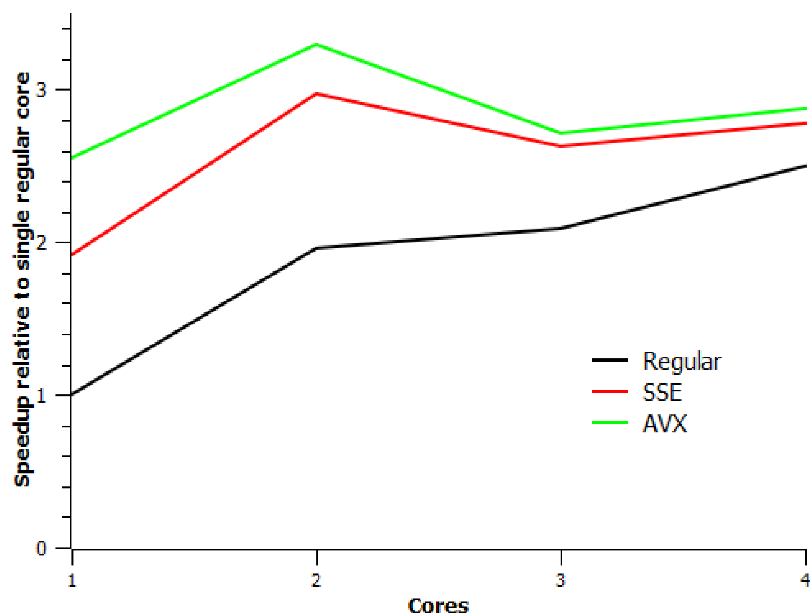

Figure 6.6: A comparison of the FDTD performance with SSE and AVX computing a grid of 20 million points on the four core Intel i5-2500k processor

This throttling effect for a higher core count could possibly be attributed to the limitation in data bandwidth for the data being supplied to the processing cores by the memory. This assumption is borne out by the improvement in the performance of the FDTD with AVX when using two channels supplying data from the memory to the cores, as is shown in figure 6.7.

Stellenbosch University http://scholar.sun.ac.za

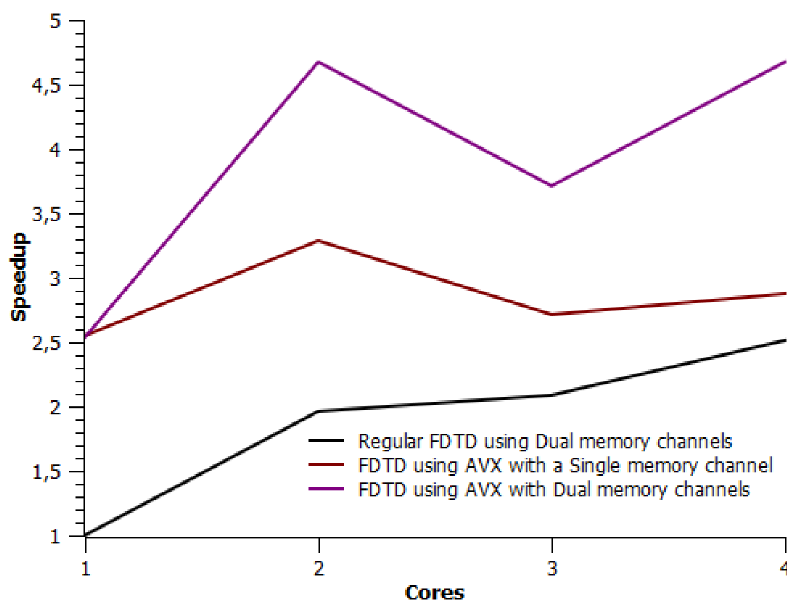

Figure 6.7: Improvement in the FDTD using dual memory channels . The regular MPI dual indicates processing with no vector extensions but using two memory channels. The results are derived from processing the FDTD on a four core i5-2500k processor computing 20 million cells.

The performance-limiting memory bottlenecking issue, as described in section 5.6, is very probably the reason why the 256 bit AVX based FDTD implementation does not achieve the expected speed-up over the 128 bit SSE based FDTD implementation graphed in figure 6.6.

Programming the FDTD with vector registers such as the AVX requires two computational stages:

Stage 1: A data fetch stage that will load the FDTD grid values into memory in preparation for processing.

Stage 2: A processing stage that will compute the FDTD values in a data parallel sense.

The benefit of the SIMD based implementation of the FDTD using AVX lies in the ability to perform simultaneous computations as described in Stage 2, its weakness is that new data cannot be supplied to the vector registers quickly enough for relatively sparse computations such as used for the FDTD update equations. If one could therefore increases the number of computations in the second stage while the overhead of the memory fetch requirement remains the same, this will improve the performance of the FDTD SIMD operation, as described in section 5.6.

This can be expressed in the following simple relationship as:

$$\frac{time\ taken\ for\ data\ fetch\ per\ FDTD\ iterative\ loop}{time\ taken\ for\ compute\ of\ FDTD\ on\ vector\ registers} = n$$

If n > 1 then memory bottlenecking will occur on multicore chips using SIMD with AVX.

If n < 1 then memory bottlenecking will not restrict computation of FDTD, or have less of an effect.

It is possible to increase the performance of this two stage cycle by improving the efficiency of the data fetch operations or by increasing the load of the vector register computation.

Stellenbosch University http://scholar.sun.ac.za

The data fetch operation can be made more efficient by using low latency local cache memory to fetch data [100], or to provide more physical channels to access the data. Attempts were made to program the data supply to the local cache but it was found that the implementations of the FDTD using these did not show any performance advantage over the conventional fetch from memory. Using more than one physical memory channel to access memory showed a marked improvement of the data fetching operation resulting in an overall improvement of the FDTD using AVX as is shown in figure 6.7.

Some compilers are achieving a performance improvement by automatically vectorising the FDTD using the AVX registers on multicore processors. An examination of the resultant assembly code provided by the Intel C compiler, reveals that the automatic vectorisation is achieved by unrolling the embedded loops that make up the FDTD computation, and arranging these to be processed by the AVX registers on the processor, a technique which is very similar to the manually coded method. Figure 6.8 shows that the Intel compiler successfully automates the vectorisation of the FDTD for a single core, but is not very effective for more than one core in the multi core environment. Three curves that show the effect of using the Intel auto-vectorising compiler are compared in figure 6.8. These curves are:

1) The Regular FDTD with no AVX speedup.
2) The FDTD using the Advanced Vector eXtensions (AVX) achieved by manually programming the registers.
3) The FDTD as accelerated automatically by the compiler with the AVX.

A consideration that also has to be made is that the building block of contemporary supercomputers currently being manufactured is a multicore processor. The implication of this is that any computational mechanism that will speed up the processing of the FDTD on the multicore processor, such as the AVX, will also speed up the processing of the FDTD on the supercomputer.

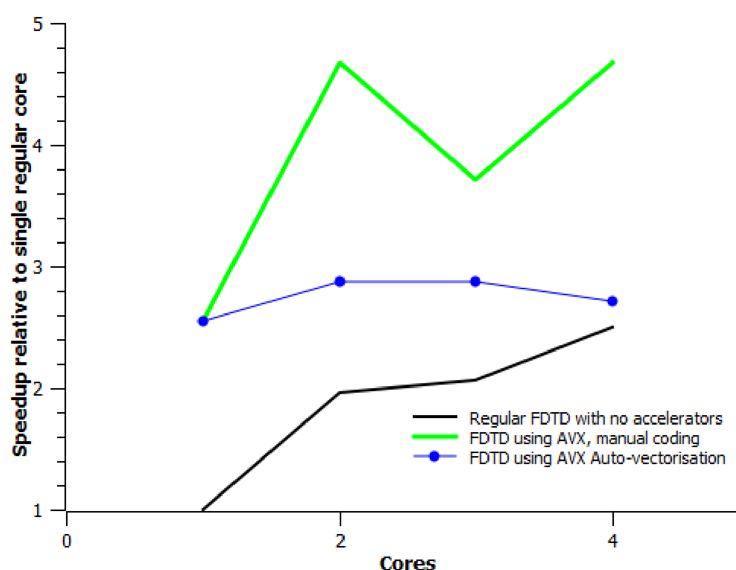

Figure 6.8 Auto-vectorisation of the FDTD with the Intel C compiler showing speedup of the four core Intel i5-2500k processor computing 20 million cells.

Stellenbosch University http://scholar.sun.ac.za

6.2.3 M9000 SMP and Bluegene/P

Architecturally the Bluegene/P and the M9000 SMP are similar in that they both use a multi-core processor as their basic computing work-horse and have a highly sophisticated proprietary processor interconnect to provide communication between the processors providing the actual computation. The primary difference in these systems is the way their respective operating systems view their memories. The Bluegene/P operates as a non-unified memory architecture (NUMA) and the M9000 sees one unified memory address space (UMA). Although the physical memory on both machines is distributed and associated with specific processors, the M9000 SMP has a memory access latency on average ten times faster than the Bluegene/P (see table 4.2). Although not every processor is directly attached to the same physical memory, this highly effective memory access speed (low access latency) allows it to be used as one addressable memory.

Another memory specific difference between the Bluegene/P and the M9000 SMP is that the Bluegene/P only allows 32 bit memory addressing and makes available two GB of memory per processing node (each with four cores). The M9000 on the other hand has 64 bit memory addressing, which allows for the processing of a much larger FDTD grid size per node. Experimentation has shown that the Bluegene/P will run the parallel FDTD without encountering memory restrictions with up to 20 million grid points per node. As there are 1024 nodes on the Bluegene/P used for this thesis this will theoretically allow the processing of an FDTD grid of 20 billion grid points. The maximum size of FDTD grid size processed on the Bluegene/P for this thesis is two billion cells. The performance of the parallel FDTD with MPI on the Bluegene/P and the FDTD with MPI on the M9000 SMP is shown in figure 6.9 below. Of note is the logarithmic scale of the throughput in millions cells per second which shows the extraordinary processing power of the Bluegene/P when compared to the performance of the M9000 SMP.

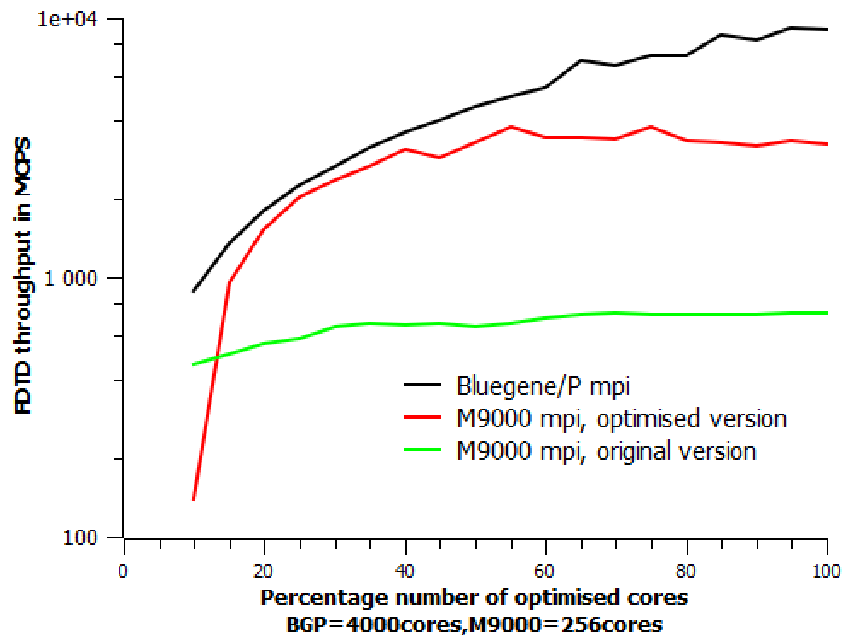

Figure 6.9: The performance of the parallel FDTD with MPI on the Bluegene/P.

The efficiency of the FDTD achieved on the Bluegene/P compares well to what is found by Yu et al. [53] although the results presented here do show more variability, very possibly arising from

Stellenbosch University http://scholar.sun.ac.za

different FDTD thread to physical core mappings between timing runs. The efficiency of the FDTD on the Bluegene/P and a comparison to the Yu-Mittra [53] results is shown in figure 6.10 below.

Despite its Bluegene/P's formidable reputation as a supercomputing engine, the programming of the parallel FDTD with MPI is a conventional programming process. The parallel FDTD code used on the Bluegene/P was developed on a Personal Computer using the C language and MPI.

The parallel FDTD built using the MPI library (section 5.5) requires absolutely no code change whatsoever between the Bluegene/P and the M9000 SMP platforms. Equal FDTD grid sizes were computed on each processor and there were no load-balancing issues on either the M9000 SMP or the Bluegene/P because of this. The shared memory architecture allows the development of the parallel FDTD with openMP on the M9000, which is not possible on the Bluegene/P. Although using the openMP threading library does not provide an advantage in processing speed and capacity, it does allow for much faster program development and prototyping. It is also possible to program the FDTD on four core IBM Power processors (such as used by the Bluegene/P) with openMP and using these as compute nodes in a larger MPI hybrid structure, as was deployed by Yu and Mittra [51]. This hybrid deployment could explain the more efficient processing of the FDTD on the Bluegene/P with a core count in the lower range as is shown in figure 6.10. The Yu and Mittra data has been extracted from work done by [53], as described further below.

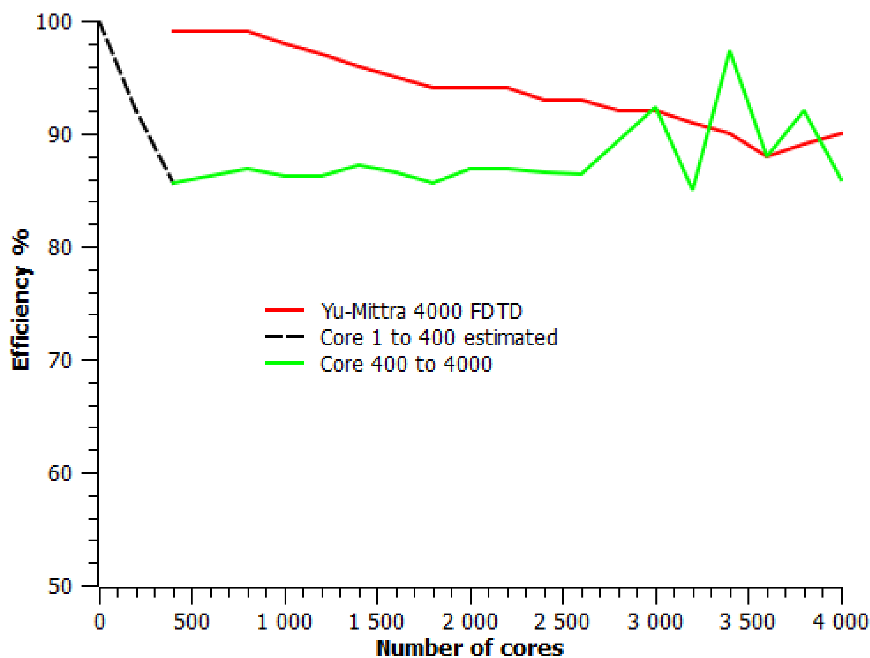

Figure 6.10: Efficiency of the FDTD on the Bluegene/P with a FDTD grid of two billion cells.

6.2.4 Cluster and SMP

As the M9000 SMP and the Westmere (Xeon processor based) cluster both use multicore processors as their computational workhorses, it is not unexpected that they all achieve a similar order of speedup for the parallel FDTD on a small number of cores when implemented with the MPI API, as is shown in figure 6.11. Although the clusters are as efficient as the SMP when only using a few cores, the M9000 SMP is more effective at processing the FDTD as the number of cores used increases, also

Stellenbosch University http://scholar.sun.ac.za

graphed in figure 6.11. This mirrors in some way the basic architecture used for the clusters, as the multicore processors can also be considered to be individual SMPs. The cores on the multi core processor constituting the M9000 SMP will access local on-chip memory, which has a lower memory access latency than the memory accessed via a memory interconnect. This is because the cores on the multicore chip all access the same physical memory arrangement, even though it is not the same computational address space.

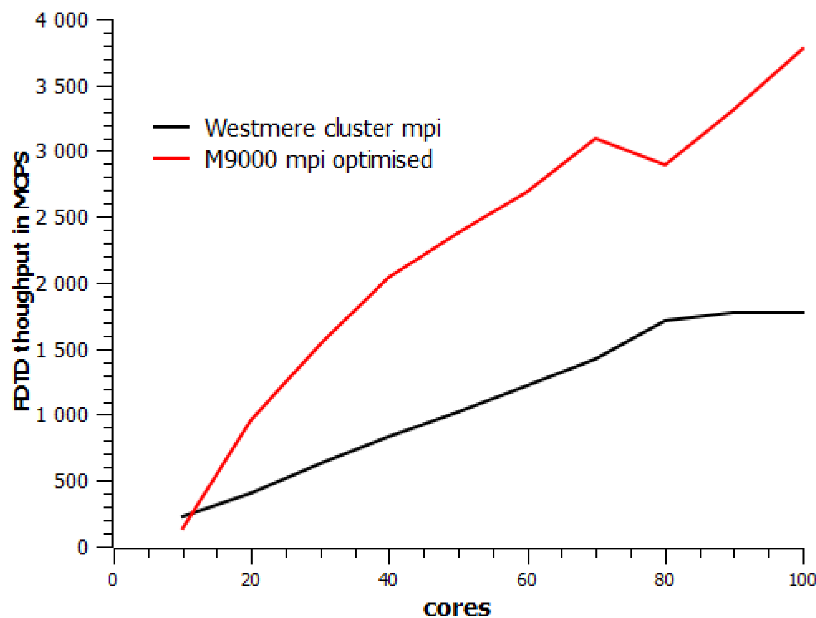

Figure 6.11: The FDTD performance comparison on the M9000 SMP and Westmere Cluster when processing a FDTD grid of 2000 million cells.

Although the cluster performance is limited in its ability to achieve good FDTD cell throughput as is shown in 6.12, this does not affect its ability to process large FDTD models, as is shown in figure 6.12 below.

The reader will notice that although the cluster does achieve a speed-up for a low core count, it is less efficient when using a larger number of cores as is shown in figure 6.14. An examination of the efficiency achieved by similar FDTD deployments on clusters [125, 126] shows some agreement with the FDTD efficiency achieved with a maximum processing core count of 16. An interesting effect of the type of communication channel on the efficiency of the FDTD on a cluster is shown by [144] in figure 6.13 below. The acceleration factor is the speedup achieved by a given number of nodes.

Stellenbosch University http://scholar.sun.ac.za

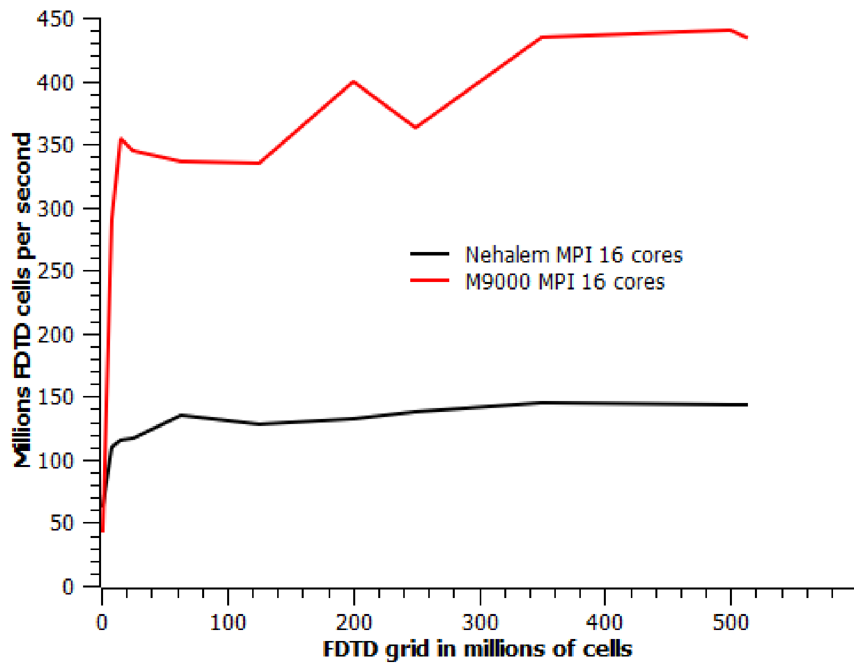

Figure 6.12: A comparison of the effect of size of the FDTD modelling space on a cluster and the M9000 SMP.

The efficiency of the FDTD is shown to be subject to a combination of the latency and bandwidth of the communication channel. In essence, figure 6.13 supports the reasoning of how the long latency, discussed in chapter 4, negatively affects the performance of the parallel FDTD.

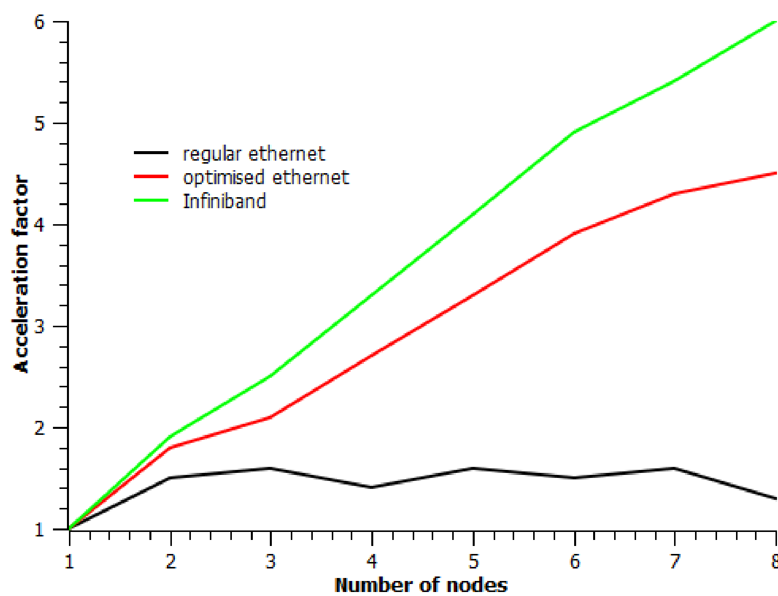

Figure 6.13: Graph taken from [144] showing the effect of inter-node communication efficiency on the acceleration of the parallel FDTD.

Stellenbosch University http://scholar.sun.ac.za

Other FDTD implementations [53] on clusters are able to maintain an efficiency of above 80% for a large number of cores [80 cores]. It is speculated that this may be as a result of implementing the parallel FDTD with a processor topology [54] that more closely matches that of the FDTD model dimensions. The matching of topology is something that cannot be easily achieved when processing a FDTD program on a system where the processors are abstracted from the topology by a scheduling process such as Moab.

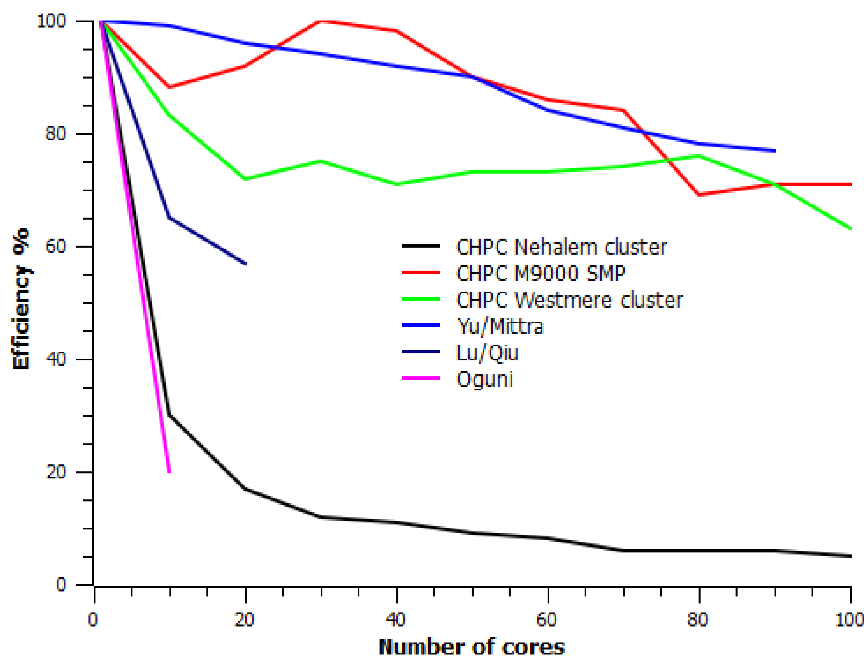

Figure 6.14: A comparison of the efficiency of the parallel FDTD with MPI on several clusters at the CHPC. This is compared to the efficiency of the parallel FDTD with openMP on the M9000, also a CHPC system. Also shown, although for varying model sizes and number of cores, are FDTD cluster efficiencies from Yu, Lu, and Oguni [100, 125, 126].

The comparison of the parallel FDTD on the Distributed Shared Memory Architecture (DSMP) architecture, in this instance two multi-core Nehalem processors connected via the Quick Path Interconnect (QPI) technology, is in effect a comparison of a system that is part SMP and part cluster. The reason it is being viewed as an SMP is because the operating system on this platform is configured to view the addressable memory as a Unified Memory Address (UMA) system. The DSMP configuration is similar to a cluster configuration as it is a collection of multi core processors that are more tightly coupled which permits them to achieve lower inter-processor communication latency than a conventional cluster configuration. A comparison of the performance of the parallel FDTD implemented with openMP on a conventional SMP (M9000) and the Intel DSMP is shown below in figure 6.15. The performance on both platforms is very similar despite the difference in the specification of the Chip Multi Processor (CMP) and interconnects on these platforms. The degradation in performance of the DSMP from 16 cores onwards is assumed to be because of the processing inefficiencies caused by the hyper-threading, as described in section 4.14.

Stellenbosch University http://scholar.sun.ac.za

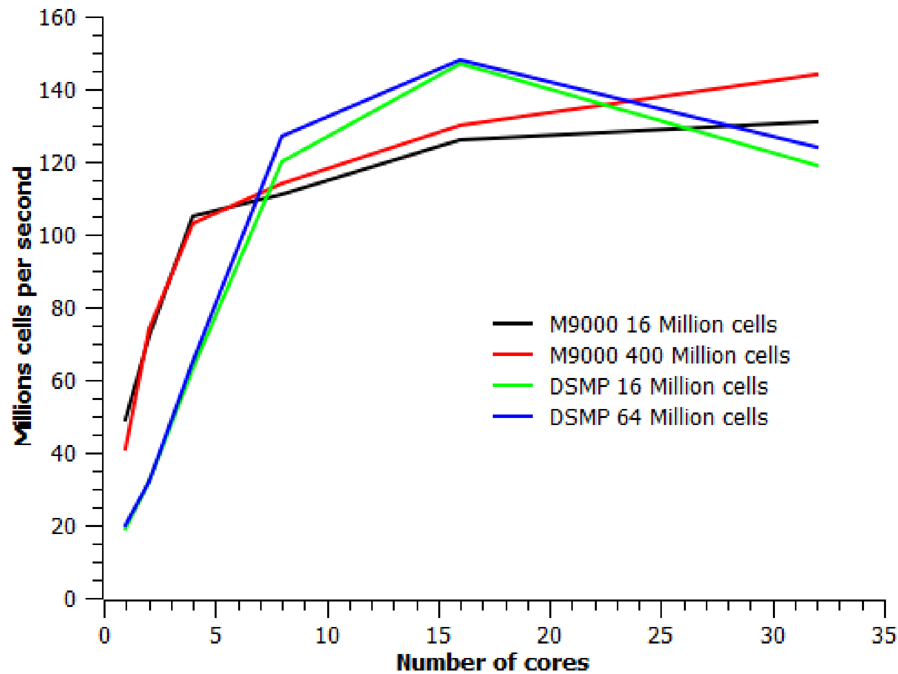

Figure 6.15: The performance of the parallel FDTD using openMP on the SMP and Distributed SMP platforms (large multicore processors).

6.2.5 Integrated GPGPU and discrete GPGPU

The merits of processing the FDTD with an SIMD device has been discussed in section 6.2.2 above. It is apparent that one of the major limitations in processing large FDTD grids simultaneously on both a host computer and discrete GPU is the data transfer latency between the host's memory and the GPU memory, as per the description in section 4.3.1. The Accelerated Processing Unit (APU) provides the GPU on the same die as the CPU cores, as is described in section 4.3.2, thereby providing access to the same physical memory and using the same memory caching system. This arrangement of CPU and GPU has the potential to reduce the data communication latency between the integrated CPU and GPU, and hence the concurrent computation of the FDTD on both the CPU and the GPU. Figure 6.16 below is a schematic showing a discrete GPU configured with an APU to process the FDTD. The FDTD grid data has to be moved from the APU's memory to the external GPU's memory for processing. If one is considering the concurrent processing of the FDTD grid with both a GPU and the multiple cores of the APU, is makes sense to minimise the effect of large communication latencies and process the FDTD with both the APU's cores and the APU's GPU.

Stellenbosch University http://scholar.sun.ac.za

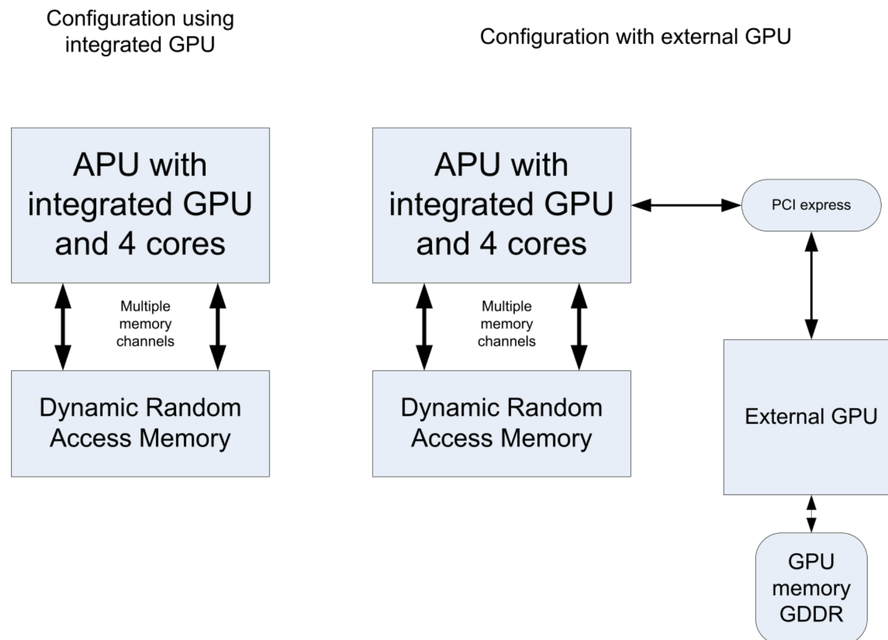

Figure 6.16: A description of the additional data communication overhead required for an APU to process the FDTD on an external GPU.

According to the independent performance values of the FDTD on the multiple cores and the internal GPU of the APU, as is shown in figure 6.17, the potential of a combined FDTD throughput provided by the multiple cores and the internal GPU, could be in the excess of 200 Million cells per second.

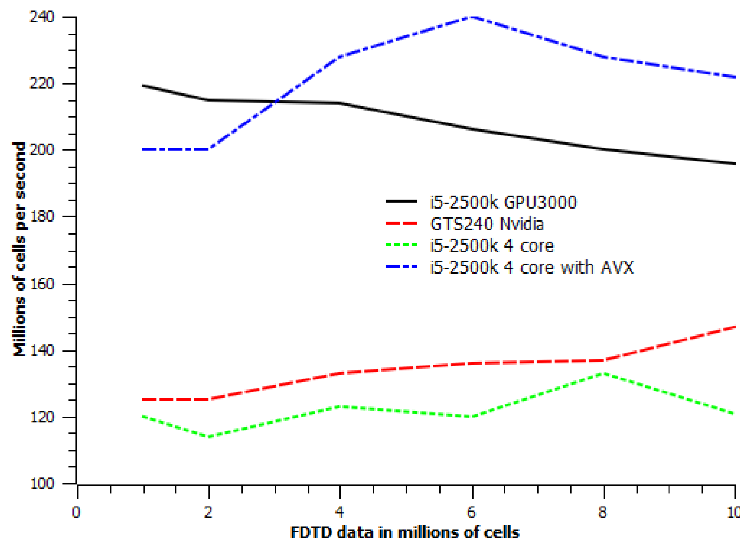

Figure 6.17: The computational throughput of the integrated GPU and a GPU configured via PCIe as a co-processor.

6.2.6 Other systems

The author is aware of other HPC parallel processing systems such as the Quantum or FPGA computers but has restricted the research performed here to the more commercially available platforms owing to time constraints.

Stellenbosch University http://scholar.sun.ac.za

6.3 General classification table

The deployment effort and performance of the FDTD algorithm for different High Performance platforms is dependent on several factors as listed in table 6.3 below. Each factor has been ranked on a scale of 1 to 10 against the respective platforms, the lower rank denoting a positive or beneficial aspect, and the higher rank indicating a negative or detrimental aspect. Each factor has been summarised below and related back to the discussion in this thesis. The classification is to serve as a comparison of the deployment effort of the FDTD on these platforms.

Category	Bluegene/P	GPU	Cluster	M9000	Multicore
Scalability of Process	1	2	1	3	4
Scalability of Hardware	1	2	1	3	5
Memory Access Efficiency	2	1	3	2	5
Openness	6	1	2	3	1
Algorithm coding effort	4	7	3	4	1
Performance (FDTD cell throughput)	1	2	4	3	6
Performance in Size, Grid Range	2	1	2	2	2
Cost of purchase	5	1	6	7	4
Power consumption	4	1	5	7	3
Implementation time	4	6	3	1	2
Future proofing from aspect of coding	3	6	3	3	3
Total Rating: 1 is ideal, 110 is difficult	33	30	33	38	36

Table 6.3: Classification table of the FDTD deployment effort on several High Performance platforms. Lower values are better.

Notes on the General classifications: Overall Ranking – 1 is beneficial, 10 is detrimental.

6.3.1 Scalability of process

The scalability of process has been ranked one (easy), to ten (hard). This is a measurement of the effort involved in scaling the FDTD on a particular computing platform.

Rank one – The FDTD as a distributed MPI process on the Bluegene/P or multicore cluster is highly scalable (section5.5), although consideration has to be made of the processor topology to achieve good performance. In practice, the use of Controlled Source Electro Magnetics [153, 154] by geophysical companies can model commercial scenarios on clusters using thousands of FDTD MPI processes.

Rank two - The FDTD can be readily adapted to a GPU cluster [153] as described in section 5.9. The FDTD experiences a good increase in capacity and capability with the corresponding addition of more GPU processing units. Communication between GPUs can take place using a variety of techniques including MPI or GPUdirect, a communication system used by Nvidia to directly exchange data between the dedicated memories of Nvidia GPUs.

Rank three – On the SUN M9000 SMP the FDTD process is easily scaled with no code change to the parallel FDTD programs created with MPI or openMP as described in sections 5.4 and 5.5. There is a

Stellenbosch University http://scholar.sun.ac.za

top limit in physical size and addressable memory that can be attached to any single SMP platform and this will limit the scalability. In order to make the FDTD process scale on the clustered SMP platform, it may be necessary to change the openMP architecture of the parallel FDTD to an openMP/MPI hybrid system to accommodate the physical cluster structure of several M9000s strung together.

Rank four – The FDTD with MPI or openMP on the Intel multicore processor scales well owing to the versatile interconnectivity offered by the Quick Path Interconnect (QPI) fabric, yet is limited by the physical size of the configuration, i.e. A motherboard with several effectively becomes a blade processor. Any further growth would define this architecture as a cluster.

6.3.2 Scalability of hardware

The scalability of hardware has been ranked from one to ten, one being highly scalable and ten being difficult to scale.

The Bluegene/P and the multicore cluster platforms have rank one as all that will be required to scale the system will be the addition of more computing hardware of similar architecture (section 4.3.5 and 4.3.6). Good examples of this are the cluster farm at Google inc. [128] and the Bluegene/P installation in the Jülich Supercomputing centre [127] in Germany.

A similar consideration has to be made for the GPU in that a GPU cluster can be created by attaching several GPUs to a single host controller (or worker node) [153], as is described in section 5.9. These GPU clusters can then be expanded quite readily by adding similar host controllers to the system. As the GPU cluster requires only a fraction of the power consumed by larger systems, such as the M9000 and Bluegene/P, no special consideration has to be made for additional power supplies and cooling. This advantage places the GPU cluster into a better scalability category and will receive a rank equal to two.

Rank three, as assigned to the SUN M9000 SMP, has the same attributes as one but the consideration has to be made that at some stage the processing capacity of the SMP will be reached (section 4.3.3). The SMP nodes can be linked together in a cluster configuration, but this will change the nature of the architecture of the platform to being a cluster.

Rank five is a rank assigned to the multicore processor as only a finite number of these will fit onto one motherboard (section 4.3.4). The architecture of a motherboard populated with multicore processors makes up the basis of a blade server. These blade servers are the building blocks of the Nehalem and Westmere clusters used for this thesis.

6.3.3 Memory access efficiency

Rank is from one to ten, one for being efficient and ten being inefficient.

The memory access efficiency is one of the key components contributing to the performance of the FDTD method on high performance computing platforms. The FDTD is computationally sparse and depends on whether sufficient data can be supplied to the processing cores quickly enough. The

Stellenbosch University http://scholar.sun.ac.za

data required by the FDTD process is supplied either from memory directly or from the communication of data with other FDTD threads. Two examples of how the memory access efficiency can influence the FDTD performance are illustrated by the optimisation of the FDTD for the GPU in section 5.7, and by the limitation in processing speed caused by the data supply bottlenecking for the vector processing on the multicore processor in section 5.11. The effect of system latency and bottlenecking on the FDTD on different high performance platforms has been described in various sections of chapters four and five.

The GPU as used in asymmetrical processing requires the data to be moved from the host memory to the GPU for processing. Once on the GPU and optimised for the GPU memory (section 4.3.1), the FDTD process achieves good performance (section 5.7). The size of the memory for the GPU is commonly limited to a few Giga Bytes (GB) and this limits the size of the array that can be processed on one GPU alone. The movement of data between a single GPU and host is hampered by the long latency experienced when communicating over a PCIe bus. This prevents the simple sharing of FDTD data grids between the GPU and host (sections 5.7, 5.9).

The processing of the FDTD using multiple GPUs, i.e. a GPU cluster (section 5.9) overcomes the memory limitation in processing large FDTD grids. In addition to this, the use of Nvidia's GPUdirect technology allows low latency communication between the memories of Nvidia GPUs in a cluster of GPUs. The ability of the GPU cluster to process extremely large FDTD arrays [153] at a throughput (7900 MCPS) comparable to that of the Bluegene/P (8000 MCPS) warrants a memory access efficiency rank of one for the GPU.

The Bluegene/P has been given a ranking of two. Despite the formidable performance achieved for the FDTD on the Bluegene/P in this thesis (section 6.2.3), it is limited in memory size by only addressing a 32 bit address space per node (section 4.3.6). Good FDTD performance was achieved on the Bluegene/P because of the efficiency of the sophisticated data interconnect. The effort required for memory access optimisation on the GPU was not required for the Bluegene/P.

The Westmere cluster computer has received a rank of three for memory access because the communication system between cluster blades is driven by an Infiniband or high speed Ethernet switch, and not a sophisticated interconnect dedicated to inter-process and memory communication, as on the Bluegene/P (section 4.3.5 and 6.2.4).

Rank value of five has been assigned to the multicore processor at the Intel Manycore Test Laboratory (MTL) as this uses the Quick Path Interconnect (QPI) technology to create a Distributed Shared Memory Processor (DSMP). The operating system distributes the memory usage across the available memory infrastructure and the user is presented with a unified memory as described in section 4.3.4 and 6.2.4. The memory access is good when processed on a small number of cores but appears to limit the performance of the FDTD as is shown in section 6.2.4.

A Rank of five has been assigned to the SUN M9000 SMP machine as inter process communication speed of FDTD threads is performed by the exchange of data in memory. Memory access performance for a low number of cores is very efficient as is shown by the good FDTD throughput achieved. FDTD performance rapidly deteriorates when processed on a larger number of cores on the M9000, presumably owing to excessive data management between the processing cores and the memory.

Stellenbosch University http://scholar.sun.ac.za

6.3.4 Openness

Rank 1 indicates good openness and rank 10 indicates difficult to access information.

In order to use any HPC system a certain learning curve is involved in getting to understand the nuances of the system and languages, compilers, etc. Although openness is a difficult concept to quantify, some have tried to formalise it [31].

Openness also refers to the amount and quality of information of self-help on the internet in the form of documentation, examples and manufacturer supported user forums. Information required for the deployment of a numeric algorithm such as the FDTD is readily available. Of the work performed for this thesis one particular example of this stands out. The Bluegene/P is a highly sophisticated supercomputer which is a culmination of extensive operational research [86]. The parallel FDTD implementation created at the Centre for High Performance Computing (CHPC) was done by extrapolating the methods from other platform's paradigms, such as the MPI threading on the Nehalem cluster, onto the Bluegene/P. Although the results provided by the implementation are adequate, better access to good documentation would have allowed the use of dual simultaneous floating point calculations per compute core. For these reasons, the Bluegene/P was afforded a high ranking for openness, i.e. some information could not be easily located in the Bluegene/P documentation.

A system will also score a high ranking for openness if the information required to deploy an algorithm depends on IT support. Voluminous information may be available on the web, but does not apply to the development situation in hand. Direct intervention may be required by the HPC centre's support personnel to allow the developer to compile, link, or run a program.

The good quality of the documentation provided by Nvidia's CUDA development has made possible the evolution of the GPU into a common place development platform for high performance computing. By providing good examples that are easy to understand and well laid out documentation that is readily available provides an easy entry point to HPC.

A low ranking is indicative of a system that has good openness. A high ranking refers to a system that is not very open. Although the openness value is not easy to quantify, it is critical to have a good openness rating if one is considering efficient software development on an HPC platform.

6.3.5 Algorithm coding effort

The coding effort required to produce the parallel FDTD on a specific platform is dependent on several factors, as described in section 3.10. These are in summary;

- User community size
- Maturity of technology
- Hardware or platform availability
- FDTD Coding precedent
- Documentation
- Orthodoxy

Stellenbosch University http://scholar.sun.ac.za

The multicore platform, essentially multicore processors coupled together with the Quick Path Interconnect fabric (QPI), has the best and lowest Rank (= one). The FDTD is rapidly converted to parallel form with a mature threading technology (either openMP or MPI) and a platform is always available. Documentation is excellent and the user groups are large and well established, able to furnish advice if one needs assistance. The process flow of the parallel FDTD on this platform is not unlike the process flow of the FDTD in serial form.

The GPGPU implementation has been listed on the other end of the ranking (= seven) scale owing to the unorthodox coding requirements and the lack of a specific coding template for the FDTD on the GPU. The CUDA documentation is good and there are several large and knowledgeable user groups that can assist with coding issues.

6.3.6 Performance in speed (capacity in FDTD cell throughput)

The peak performance of the FDTD was measured in Millions of Cells Processed per Second (MCPS). The peak measurement taken for each particular platform is entered in the table 6.4. The performance figures shown in table 6.4 are from FDTD implementations for this thesis and are described in some detail in chapters five and six. Apart from the GPGPU, all of these implementations took approximately the same coding effort and were measured with the same FDTD algorithm. The Bluegene/P dominates these results owing to the formidable performance of its interconnect (section 4.3.6) and also because this machine was available as a dedicated resource for this thesis. Although the FDTD performance achieved on the cluster for this thesis is low in terms of millions of cells per second, some larger cluster implementations by Yu and Mittra [100] are shown to achieve the performance of that shown for the Bluegene/P.

Platform	Peak MCPS	Rank
Bluegene/P	8900	1
GPGPU Nvidia Cluster* [153]	7910	2
M9000 SMP	3800	4
Intel Westmere Cluster	1536	5
MTL Nehalem multicore	148	6

Table 6.4: The FDTD processing throughput in Millions of Cells Per Second (MCPS) achieved by the parallel FDTD implementations for this thesis. * other sources.

6.3.7 Performance in terms of model size (capability)

Rank one is for a large FDTD model size, to ten being a small model size.

This evaluation is for which platforms can deploy the biggest FDTD grids. The Bluegene/P at the Centre for High performance Computing (CHPC) has the largest number of cores and required the least amount of effort to implement for massive grids. The largest FDTD grid implemented on the Bluegene/P for this thesis, as shown in figure 6.9, is two billion FDTD cells. Multicore cluster platforms are also capable of processing this size and larger FDTD model [131, 132].

Stellenbosch University http://scholar.sun.ac.za

The cluster and SUN M9000 SMP platforms required the same effort for large system deployment, but were not always available owing to the very high user demand of these shared resources. FDTD grids of two billion cells in size were processed on the SUN M9000 SMP and on the cluster computers.

The discrete GPU could only hold relatively small memory grids and this proved to be a serious liability for the deployment of large FDTD grids. The largest deployment of GPU based FDTD model used for this thesis is 121 million grid points, achieved on the C 2070 GPU at the CHPC. Deployments of the FDTD on a cluster of GPUs [153] has been shown to process up to three billion FDTD cells.

6.3.8 Cost of purchase

Rank is from one for the cheapest to ten being the most expensive.

Some typical purchase cost pricings of HPC systems are shown in table 6.5. These pricings were obtained from the Centre of High Performance Computing in Cape Town. The prices given in the table 6.5 are approximates for the year 2010 to 2011. Prices have been normalised to the peak FDTD throughput in MCPS shown in table 6.4.

The normalised prices show that best value for the processing using the FDTD is achieved for the GPU, which is in a category of its own in this regard and has been awarded a rank of one for the general classification in table 6.3. Although the Intel multicore system also achieves a good normalised value, it is an order of magnitude less than the GPU and therefore is assigned a rank of four. The normalised pricing of the cluster system has been adjusted for the proportion of cores available to run the FDTD method, and is very similar to the Bluegene/P. According to the normalised pricing, the M9000 has been awarded the lowest ranking.

Two pricing considerations that have not been factored into the study are the maintenance costs for the HPC systems, and for the system life cycle before it is overtaken by newer technology. According to Koomey's law, discussed in sections 4.4 and 5.2, the life cycle of a HPC system may currently be only one to two years, before it is overtaken by newer technology. This factor alone makes the processing of the FDTD by GPU, or GPU cluster, an unassailable cost proposition.

Platform	Cost in Rand 2010/2011	Price per core in Rand	Peak MCPS per million Rand	Rank
GPU GTX 480	6 000	67	57692	1
GPU Cluster (C2070 x 4)*	100 000	55	-	-
Intel multicore MTL	30 000	1875	4933	4
Bluegene/P	7 000 000	1708	1271	5
Intel cluster	9 900 000	1237	1090	6
SUN M9000	5 100 000	19921	760	7

Table 6.5: Purchase price of HPC platforms. * costing for price of purchase comparison only

Large computing systems require sophisticated servicing and sub systems, such as cooling racks and clean rooms. There is a cost attached to a maintenance plan for HPC system in terms of capital expenditure and in terms of dependence on a specific vendor. These maintenance costs are reliant on the relationship between the HPC system vendor and the computing centre hosting the HPC

Stellenbosch University http://scholar.sun.ac.za

system, and should be formulated in the form of a formal Service level Agreement (SLA). The absence of an SLA could result in the total failure of the system, i.e. it would be awarded a rank of ten+.

6.3.9 Power consumption

Rank one is desirable, ten is not.

The power consumption of the different hardware platforms was examined in section 4.4.1 and 5.2. It was concluded there that the power consumption was difficult to normalise and hence compare, although some definite trends can be established.

The power consumption is a limiting factor for the size of the computing system from the aspect of being able to dissipate enough heat without physically destroying the physical computing hardware, and from reducing the financial cost of the power consumption in itself.

Power per peak MCPS = peak MCPS/system power

HPC system	Power per core (W)	Power per peak MCPS (W/MCPS)	Rank
Nvidia C 2070 GPU*	0.5	0.3	1
S 870 GPU cluster	2.0	0.8	-
Intel multicore MTL*	5.3	5.3	3
Intel Phi MIC	6	-	-
IBM Bluegene/P*	9.7	4.5	4
Nehalem cluster*	10	5.3	5
Sun M9000-64*	151	10.2	7

Table 6.6: Power features of different HPC systems (- not included in main classification)

6.3.10 Implementation time

The Implementation time is taken from the time to verify the correct operation of the parallelised FDTD on a specific platform, i.e. the serial FDTD was reformulated into the parallel FDTD as described in chapter five. The coding environment was a mixture of Microsoft Windows and a variety of Linux operating systems. The coding language was C or C++. Technologies were restricted to MPI, openMP, and data parallelism for the GPUs.

The deployment time listed in table 6.7 below is an estimate of the programming time it took for one person (the author of this thesis) to deploy the basic parallel FDTD on a particular platform:

Platform	Time to deploy(days)	Rank
SUN M9000	Less than 1	1
Intel multicore MTL	1	2
Cluster	3	3
Bluegene/P	5	4
GPU	30+	6

Table 6.7: Time to implement the parallel FDTD

Stellenbosch University http://scholar.sun.ac.za

According to this table, the SUN M9000 received the best rank (=one) and the GPU the worst ranking (=six). The implementation time of the parallel FDTD shown for the GPU is conservative. Although the learning curve for the CUDA programming did take considerable time, most of the GPU coding effort was taken up by coding the PML as split field components. The objective of coding the same basic serial FDTD algorithm into parallel form for every high performance platform was to normalise the effort taken to deploy the FDTD.

6.3.11 Future proofing from aspect of coding

Future proofing of code is a serious commercial consideration when building any high performance system owing to the constantly evolving price/performance of new hardware and the new uses that are being found for the FDTD. A hardware platform will receive a low ranking for this category if the legacy FDTD code is able to be easily ported from one generation of the computing platform to the next, as the cost for the re-deployment of the code will be small. The cluster or IBM Bluegene would be good examples of this as the same parallel FDTD code base written with MPI will run on any of these platforms, and has therefore received a ranking of three.

The rapidly changing architecture of the contemporary GPU defines this hardware as immature and the FDTD written for this platform requires constant optimisation from one generation of hardware to the next. For this reason the FDTD for the GPU is considered to be less "future proof" and will receive a higher, less "future proof", ranking.

Customised hardware platforms such as the FPGA require coding specific to that platform. Upgrading from one device to another, even from the same manufacture, will probably require some recoding, even if only for optimisation.

6.4 Summary

The parallel FDTD implementations on several different HPC systems have been compared in order to expose the characteristics that contribute to their deployment effort and performance. These parallel FDTD systems are deemed to be a good representation of most of the parallel systems currently in commercial or academic use. The deployment effort has been summarised in a tabular form and the contributing characteristics have been listed on a ranking scale of one to ten, one denoting a rank that is conducive to a good or easy deployment effort and ten as a detriment to the deployment effort. Each ranking is described in some detail and related back to the sections in this thesis that addresses these characteristics.

A total value of all the rankings shows that the GPU systems are the easiest to deploy. Although initially considered to be too small to manage large FDTD modelling scenarios, the GPU cluster is an entry point to very large scale FDTD processing at a low cost. The results of the comparison do not imply that these systems can all be viewed as competing HPC systems, instead they are all components of the hybrid computing solutions that will be used to process the parallel in the future.

Stellenbosch University http://scholar.sun.ac.za

Chapter 7

7 Conclusions and Future Work

7.1 Contributions
7.2 Publications
7.3 Trends in FDTD and HPC
7.4 Possible directions for future work
7.5 Conclusions

7.1 Contributions

This thesis has made the following contributions:

1) Provided researchers and software application developers with a comprehensive comparison, quantified where possible, of the deployment effort required to implement algorithms using the parallel FDTD method on a cross section of contemporary high performance computing platforms.
2) A quantified determination that shows the GPU cluster to be the most suitable current high performance platform for the deployment of the parallel FDTD.
3) Identified programming techniques required for the deployment of the FDTD on different high performance computing platforms. The programmer can choose techniques from a variety of implementation methods and does not need to spend time in creating a parallel FDTD solution from first principles. The use of these techniques radically reduces the programming time needed to create a working parallel FDTD program. Modern high performance computing platforms combine a variety of architectures that require of more than one programming technique to achieve good FDTD performance.
4) Provided a comparison of the performance of the currently popular high performance computing architectures from the aspect of a parallel FDTD implementation. Although the FDTD method is not difficult to parallelise, it does make full use of all the facets of a high performance computing system, such as the inter-processor communication channels, which the algorithms such as those used by the LINPACK standard [84] does not.
5) Plotted the historical relationship between the parallel FDTD and the emergence of high performance computing. The operation of the parallel FDTD is not practical without high performance computing and the advanced techniques used to implement the parallel FDTD, owing to the large demands on computing capacity and capability required by this method.
6) Computational mechanisms have been identified that will be relevant to the deployment of the parallel FDTD on new multicore architectures in the near future. An example of this is the acceleration of the FDTD using vector register technology, such as SSE or AVX, on up-scaled processors such as the Intel Phi (a 50 core processor slated for release in 2013). The implementation analysis performed in this thesis indicated that although programming the parallel FDTD for this type of device may be quite conventional, the FDTD implementation will probably be subject to data bottle-necking on Many In Core (MIC) processors. The data bottle-necking issue is in itself an important characteristic as although some processes will

Stellenbosch University http://scholar.sun.ac.za

perform well on the 50 core processor, the FDTD may not, owing to the limited data supply from the memory to the processing cores.

7) The creation of a leading edge deployment of the FDTD on multicore processors with AVX registers. At the time of recording the research with these vector registers, there was no publication available that had achieved this. At the time of completion of this thesis there are now several publications showing the result of accelerating the FDTD with AVX registers, but these all discuss one specific method of implementation. The work in this thesis has outlined several ways in which to implement the parallel FDTD using acceleration from the AVX functionality.
8) The creation of a common parallel FDTD code with MPI that can be run on many different supercomputers with different architectures. The parallel FDTD code is operational on personal computers, large computer clusters, IBM's Bluegene/P and the SUN M9000 and is based on the MPI threading technique. Using the same FDTD coding method is very useful when benchmarking the FDTD deployment on different computing platforms.
9) The creation of a parallel FDTD code for GPUs using Microsoft's Direct Compute interface. This FDTD implementation for GPUs can be used on GPUs from different manufacturers. For this thesis the Direct Compute based FDTD implementation was tested on Nvidia and Intel GPUs.

7.2 Publications

The following contributions have resulted from the research work into the deployment of the parallel FDTD:

Conference Proceedings and Publications:

1) R G Ilgner, D B Davidson, "A comparison of the FDTD algorithm implemented on an integrated GPU versus a GPU configured as a co-processor", *IEEE International Conference on Electromagnetics in Advanced Applications (ICEAA),* pp. 1046-1049, 2012.
2) R G Ilgner, D B Davidson, "A comparison of the parallel implementations of the FDTD method on a Shared Memory Processor and a GPU", *AfriCOMP11 - Proceedings of AfriCOMP 2nd African Conference on Computational Mechanics,* pp. 1-6, 2011.
3) R G Ilgner, D B Davidson, "The acceleration of the FDTD method using extended length registers in contemporary CPUs", *South African Centre for HPC National Meeting,* pp. 1- 4, 2011.
4) R G Ilgner, D B Davidson, "A comparison of the performance of the parallel FDTD method using openMP versus the parallel FDTD method using MPI on the Sun Sparc M9000", *South African Centre for HPC National Meeting*, pp. 1- 4, 2010.

7.3 Trends in FDTD and HPC

The comparison of the FDTD deployment on various high performance system undertaken for this thesis shows that there is a trend in achieving greater processor performance with a corresponding reduction in power consumption. In addition to achieving good performance for the processing of the FDTD, GPU clusters also provide a platform with very low power consumption.

Stellenbosch University http://scholar.sun.ac.za

A trend can be identified with regard to the architecture of the high performance computers on which the FDTD is deployed. We are seeing an ever-increasing number of cores being placed on processors dies, and increase in the vector processing capacity of these cores. The parallel FDTD techniques deployed in this thesis describe the positive and negative aspects of these implementations in some detail and the considerations that need to made for the optimisation of the FDTD on these platforms.

Another trend is the move towards hybrid architectures, such as the Accelerated Processing Unit (APU) and the Tianhe Supercomputer [131] which uses a hybrid combination of blade computers and GPUs to achieve processing performance. This thesis allows a comparison of the different parallelisation and performance optimisation techniques that can be used to implement the FDTD on these hybrid platforms.

7.4 Possible directions for future work

To conclude this thesis, here follows a list of further FDTD related items that are not in the scope of this thesis but may be interesting to investigate in the future:

- An FDTD algorithm that can efficiently compute a data set that is larger than the memory of the system can contain. Data would be continuously drawn into the processing engine and saved back to disk. The key behind this is to produce an asynchronous process that can read and write the FDTD cell structure fast enough so that the processors are not underutilised. This technology would overcome the limitation of only processing FDTD models that fit into a single GPU's memory.
- A very large scale processing system for the FDTD that will use a synchronised clock to exchange data between the FDTD threads. It is envisaged that the synchronisation will be achieved using a GPS clock and have processing and storage nodes somewhere in the internet cloud. The communication latency would be hidden by the sheer scale of performing the inter process communication for a large number of devices synchronously. The benefit of such a system would be extremely large processing capacity for the FDTD systems.
- Adjusting the sequence of the FDTD iteration to include both the electric and magnetic field computation in one processing step, as opposed to the existing two. This would increase the computational density of the FDTD iteration and possibly improve its efficiency.
- Using the computational power of large processor systems to adjust the cell sizing of the FDTD dynamically so as to allow more efficient computation of mixed resolution meshes. Stability of the FDTD would need to be controlled on a per cell calculation basis so as to limit the effects of dispersion and inappropriate Courant conditions.
- Combining several HPC systems of different architecture at the CHPC in order to process extra large FDTD fields. An example of this would be the combination of either two independent clusters or the Bluegene/P and the SUN M9000 to process the FDTD on one combined super system.

Stellenbosch University http://scholar.sun.ac.za

- The creation of a parallel FDTD algorithm based on openMP to overcome the processing performance issues related to data contention and processor affinity.

7.5 Conclusions

The classification of the factors contributing to the FDTD deployment effort and performance in the previous chapters is as a result of the analysis and comparison of parallel FDTD implementations on various computing platforms. These classification factors are drawn directly from the experience base of having deployed the parallel FDTD systems, and having analysed the hardware and software mechanisms that contribute to the performance of the parallel FDTD process. Although the deployment effort of the parallel FDTD has been rated by using different aspects of the implementation, it must be emphasised that the ranking indicates the suitability for one type of implementation scenario or another, and not that one HPC platform is better than the other.

The deployment effort was evaluated for the following platforms:

1) IBM's Bluegene/P
2) Multicore clusters
3) SUN M9000 shared memory processor
4) General purpose graphics processing units
5) Multicore processors
6) Accelerated processing units

The following parallelisation techniques were used for the FDTD:

1) Message Passing Interface threading
2) openMP threading interface
3) CUDA and Direct Compute processing libraries for the GPU
4) Compiler intrinsics for vector registers such as AVX and SSE

A comparison of the deployment effort of the parallel FDTD on different high performance computing systems is a challenging task because there is no single quantifiable factor that makes one deployment superior to another. In addition to this, many of the features that define the deployment effort are subjective ones and can only be quantified by the comparison of different FDTD implementations.

Analysing the characteristics of the FDTD deployment effort is analogous to comparing the characteristics of different vehicles. Most vehicles have a common set of characteristics such as load carrying capacity, number of seats, fuel consumption, etc., although the type of vehicle could range from a small passenger vehicle to a large truck, i.e. The designated use of the vehicle determines the relevance of the characteristics being used for the comparison. In the context of the FDTD, as used for electromagnetic modelling, a HPC platform would be chosen according to a given purpose as well. A large commercial processing venture wishing to deploy the FDTD method to process large volumes of data would choose a large system such as a conventional or GPU cluster system, whereas

Stellenbosch University http://scholar.sun.ac.za

a smaller contractor wishing to process smaller FDTD data sets may choose to deploy the FDTD on a shared memory system such as a multicore or a discrete GPU based system. An examination of the characteristics of table 6.3 indicates that it will be easier to find the computing skills to deploy the FDTD on a computer cluster or a Bluegene/P. Conversely, a small organisation that does not have the correct GPGPU software implementation skills should avoid trying to develop the FDTD method on the GPU.

The deployment effort characteristics shown in table 6.3 compare several features that these different FDTD implementations have in common. In summary, this shows that although the contributing factors vary quite widely, the deployment effort on computer clusters and supercomputers such as the Bluegene/P all have a similar deployment effort and that the GPU implementations are more difficult to deploy the FDTD on.

Parallel FDTD implementations can defy the tenets of high performance processing, such as “bigger is better”. An interesting FDTD deployment that illustrates this is the phenomenon of “parallel slowdown”, experienced on the SUN M9000 SMP. Some processing architectures reach a maximum processing potential with an optimum number of cores and the processing performance degrades if more processing cores are used. It is desirable to deploy the parallel FDTD with the least amount of effort as a singular assessment based on the number of processing cores alone does not always make it apparent which systems are better deployment targets.

Despite the Bluegene/P being a sizeable investment, it does offer good FDTD performance results for very little deployment effort. Although some negative comments are made in this thesis about the availability of the development information for this system, the contemporary version of the Bluegene series is a production-ready supercomputer and should be considered on a par with any large cluster when considering an investment for a large FDTD processing project. An old computing industry adage claims “No one ever got fired for buying IBM!”, and the Bluegene series is one of the reasons that gives substance to this.

The GPU and GPU cluster provide extraordinary performance in terms of capacity and capability. Given the low cost advantage of the GPU cluster purchase over other larger HPC system, the GPU cluster has to be the considered as the most obvious choice for the processing of numerical formulations such as the FDTD. In addition to providing the processing power for numerical formulations, the GPU also has the ability to process the graphical results from the formulations, which are an important stage of the electromagnetic modelling process. Achieving good processing performance from the GPU requires an intimate knowledge of how to optimise numerical algorithms for these devices.

Stellenbosch University http://scholar.sun.ac.za

Chapter 8

8 Bibliography

[1] K S Yee, “Numerical Solution of Initial Boundary Value Problems Involving Maxwell’s Equations in Isotropic Media,” *IEEE Trans. Antennas Propagation*, vol. 14, No 8, pp. 302-307, 1966.

[2] D B Davidson, *Computational Electromagnetics for RF and Microwave Engineering, 2nd edn,* Cambridge University Press, Cambridge, England, 2011.

[3] A Taflove, S C Hagness, *Computational Electrodynamics, The Finite-Difference Time-Domain Method. Third Edition,* Artech House, Chapters 3 to 7.

[4] P Micikevicius, *3D Finite Difference Computation on GPUs using CUDA*, *NVIDIA*, 2701 San Tomas Expressway, Santa Clara, California, 95050.
Available at: http://www.nvidia.com

[5] J de S Araujo, R Santos, C da Sobrinho, J Rocha, L Guedes, R Kawasaki, “Analysis of Monopole Antenna by FDTD Method Using Beowulf Cluster,” *17th International Conference on Applied Electromagnetics and Communications*, Dubrovnik, Croatia, October 2003.

[6] D Sullivan, “Electromagnetic Simulation Using the FDTD Method” *IEEE Series on RF and Microwave Technology*, 2000.

[7] J Maxwell, *A Treatise on Electricity and Magnetism*, Clarendon Press Series, Oxford.

[8] M Hill, M Marty, ”Amdahl’s Law in the Multicore Era”, *IEEE Computer Society Publication*, July 2008.

[9] C Guiffaut, K Mahdjoubi, “A Parallel FDTD Algorithm using the MPI Library”, *IEEE Antennas and Propagation Magazine*, vol. 43, No 2, pp. 94-103, April 2001.

[10] D Buntinas, G Mercier, W Gropp, “Implementation and Shared-Memory Evaluation of MPICH2 over the Nemesis Communication subsystem”, *Mathematics and Computer Science Division, Argonne National Laboratory*. gropp@msc.anl.gov.
Available at: http://www.anl.gov

[11] P M Dickens, P Heidelberger, D M Nicol, “Parallelized Direct Execution Simulation of Message Passing Parallel Programs”, *IEEE transactions on Parallel and Distributed systems*, pp.1090-1105, 1996.

[12] K Donovan, “Performance of Shared Memory in a Parallel Computer”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, no 2, pp. 253-256, 1991.

[13] A Taflove, S C Hagness, *Computational Electrodynamics, The Finite-Difference Time-Domain Method. Third Edition,* Artech House, Chapters 8, Near Field to Far Field.

[14] L Chai, *High Performance and Scalable MPI Intra-Node Communication Middleware for Multi-Core Clusters*, PhD Thesis, Ohio State University, 2009.

Stellenbosch University http://scholar.sun.ac.za

[15] W Yu, X Yang, Y Liu, R Mittra, D Chang, C Liao, M Akira, W Li, L Zhao, *New Development of Parallel Conformal FDTD Method in Computational ElectroMagnetic Engineering*, Penn State University, 2010.

[16] J Kim, E Park, X Cui, H Kim, "A Fast Feature Extraction in Object Recognition using Parallel Processing on CPU and GPU", *IEEE Int. Conf. on Systems, Man and Cyberneti,* San Antonia, Texas, 2009.

[17] W Yu, M Hashemi, R Mittra, D de Araujo, M Cases, N Pham, E Matoglu, P Patel, B Herrman, "Massively Parallel Conformal FDTD on a BlueGene Supercomputer", *IEEE Int. Conf. on Systems, Man and Cybernetic,* San Antonia, Texas, 2009.

[18] Intel White paper, *Intel DDR Dual Memory Dual Channel White Paper.* Available at: http://www.intel.com.

[19] W Yu, X Yang, Y Liu, L Ma, T Su, N Huang, R Mittra, R Maaskant ,Y Lu, Q Che, R Lu, Z Su, 'A New Direction in Computational Electromagnetics: Solving Large Problems Using the Parallel FDTD on the Bluegene/L Supercomputer Providing Teraflop-Level Performance", *IEEE Antennas and Propagation Magazine*, vol. 50, No 2, pp.26-41, April 2008.

[20] C Brench, O Ramahi, "Source Selection Criteria for FDTD Models", *IEEE Int Symposium on Electromagnetic Compatibility*, Denver, Colorado, 1998.

[21] R Makinen, J Juntunen, M Kivikoski, "An Accurate 2-D Hard-Source Model for FDTD", *IEEE Microwave and Wireless Components Letters*, vol. 11, no 2, 2001.

[22] D Rodohan, S Saunders, "Parallel Implementations of the Finite Difference Time Domain (FDTD) method", *Second International Conference on Computation in Electromagnetics*, Nottingham, England, April 1994.

[23] W Buchanan, N Gupta, J Arnold, "Simulation of Radiation from a Microstrip Antenna using Three Dimensional Finite-Difference Time-Domain (FDTD) Method", *IEEE Eighth International Conference on Antennas and Propagation*, Heriot-Watt University, Scotland March 1993.

[24] N Takada, T Shimobab, N Masuda, T Ito, "High-Speed FDTD Simulation Algorithm for GPU with Compute Unified Device Architecture", *IEEE Antennas and Propagation Society International Symposium*, North Charleston, South Carolina, USA, June 2009.

[25] S Adams, J Payne, R Boppana, "Finite Difference Time Domain (FDTD) Simulations Using Graphics Processors", *IEEE HPCMP Users Group Conference*, Pittsburgh, PA, USA, June 2007.

[26] D Liuge, L Kang, K Fanmin, "Parallel 3D Finite Difference Time Domain Simulations on Graphics Processors with CUDA", *International Conference on Computational Intelligence and Software Engineering*, Wuhan, China, December 2009.

[27] M Inman, A Elsherbeni, J Maloney, B Baker, "GPU Based FDTD Solver with CPML Boundaries", *IEEE Antennas and Propagation Society International Symposium*, Hawaii, USA, June 2007.

[28] R Abdelkhalek, H Calandra, O Coulaud, G Latu, J Roman, "FDTD Based Seismic Modeling and Reverse Time Migration on a GPU Cluster", *9th International Conference on Mathematical and Numerical Aspects of Waves Propagation – Waves*, Pau, France, June 2009

Stellenbosch University http://scholar.sun.ac.za

[29] T Anderson, S Ketcham, M Moran, R Greenfield, "FDTD Seismic Simulation of Moving Battlefield Targets", *Proceedings of Unattended Ground Sensor Technologies and Applications III*, SPIE vol. 4393, 2001.

[30] T Ciamulski, M Hjelm, and M Sypniewski, "Parallel FDTD Calculation Efficiency on Computers with Shared Memory Architecture", *Workshop on Computational Electromagnetics in Time-Domain*, Perugia, Italy, Oct 2007.

[31] K Lakhani, L Jeppesen, P Lohse, J Panetta, "The Value of Openness in Scientific Problem Solving", *President and Fellows of Harvard College*, 2007.
Available at: http://www.techrepublic.com/whitepapers/the-value-of-openness-in-scientific-problem-solving/314795

[32] R Maddox, G Singh, R Safranek, *Weaving High Performance Multiprocessor Fabric*, Intel press, Chapters 1-2, 2009.

[33] U Drepper, *What every programmer should know about memory*. Red Hat Incorporated, 2007.
Available at: http://lwn.net/Articles/250967

[34] Intel, White paper, *Intel® Advanced Vector Extensions Programming Reference*, June 2011. Available at: http://www.intel.com

[35] T Willhalm, Y Boshmaf, H Plattner, A Zeier, J Schaffner, "SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units", *VLDB conf*, Lyon, France, Aug 2009.

[36] Fujitsu, White paper, *Sparc Enterprise Architecture* M4000/M5000/M6000/M7000/M8000/M9000, October 2009.
Available at: http://www.fujitsu.com/global/services/computing/server/sparcenterprise/products/m9000/spec

[37] D Luebke, *White Paper, Nvidia® GPU Architecture and Implications,* 2007.
Available: http://www.nvidia.com

[38] K Leung, R Chen, D Wang, "FDTD Analysis of a Plasma Whip Antenna", *IEEE Antennas and Propagation Society International Symposium*, Washington, DC, USA, July 2005.

[39] G Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities", *AFIPS Conference proceedings*, vol. 30, Atlantic City, New Jersey, USA, April 1967.

[40] C Niramarnsakul, P Chongstitvatana, M Curtis, "Parallelization of European Monte-Carlo Options Pricing on Graphics Processing Units". *Eight International Joint Conference on Computer Science and Software Engineering*, Nakhon Pathom, Thailand, May 2011.

[41] M Su, I El-Kady, D Bader, Y Lin, "A novel FDTD Application Featuring openMP-MPI hybrid parallelization", *IEEE International Conference on Parallel Processing*, Montreal, QC, Canada, 2004. (same as 63)

[42] D Rodohan, S Saunders, "A CAD Tool for Electromagnetic Simulation on the Associative String Processor", *International Conference on Application-Specific Array Processors*, Venice, Italy, October 1993.

Stellenbosch University http://scholar.sun.ac.za

[43] T Nagaoka, S Watanabe, “Multi-GPU accelerated three-dimensional FDTD method for electromagnetic simulation”, *IEEE Annual International Conference of the Engineering in Medicine and Biology Society*, Boston, MA, USA, Aug/Sept 2011.

[44] D Mich´ea, D Komatitsch. “Accelerating a Three-Dimensional Finite-Difference Wave Propagation Code using GPU Graphics Cards”, *International Journal of Geophysics*. No 182, pp. 389-402, 2010.

[45] D Sheen, S Ali, M Abouzahra, J Kong, “Application of the Three-Dimensional Finite-Difference Time-Domain Method to the Analysis of Planar Microstrip Circuits”, *IEEE Transactions on Microwave Theory and Techniques*, vol. 38, pp. 849-857, 1990.

[48] Parallel Virtual Machine (PVM) Website.
Available at: http://www.csm.ornl.gov/pvm/

[49] L Tierney, *SNOW Reference Manual*, Department of Statistics & Actuarial Science University of Iowa, Sept 2007.

[50] Z Li, L Luo, J Wang. S Watanabe, “A Study of Parallel FDTD Method on High-Performance Computer Clusters”, *International Joint Conference on Computational Sciences and Optimization*, vol. 2, Sanya, Hainan, China, April 2009.

[51] C Zhang, X Yuan, A Srinivasan, “Processor Affinity and MPI Performance on SMP-CMP Clusters”, *International Conference on Scalable Computing and Communications; Eighth International Conference on Embedded Computing*, Dalian, China, September 2009.

[52] J Lorenzo, J Pichel, D LaFrance-Linden, F Rivera, D Singh. “Lessons Learnt Porting Parallelisation Techniques for Irregular Codes to NUMA Systems”, *18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, Pisa, Italy, Feb 2010.

[53] W Yu, R Mittra, X Yang, Y Liu “Performance Analysis of Parallel FDTD Algorithm on Different Hardware Platforms”, *IEEE Antennas and Propagation Society International Symposium*, Charleston, SC, USA, June 2009.

[54] Z Yu, R D Wei, L Changhog, “Analysis of Parallel Performance of MPI based Parallel FDTD on PC Clusters”, *IEEE Microwave Conference Proceedings*, Suzhou, China, December 2005.

[55] P Konczak, M Sypniewski, “The method of improving performance of the GPU-accelerated 2D FDTD simulator”, *18th International Conference on Microwave Radar and Wireless Communications*, Vilnius, Lithuania, June 2010.

[56] V Demir, A Elsherbeni, “Programming Finite-Difference Time-Domain for Graphics Processor Units Using Compute Unified Device Architecture”, *IEEE Antennas and Propagation Society International Symposium*, Toronto, Ontario, Canada, July 2010

[57] K Sano, Y Hatsuda, L Wang, S Yamamoto, “Performance Evaluation of Finite-Difference Time-Domain (FDTD) Computation Accelerated by FPGA-based Custom Computing Machine”, *Graduate School of Information Sciences, Tohoku University*, 2010

[58] H Suzuki, Y Takagi, R Yamaguchi, S Uebayashi, “FPGA Implementation of FDTD Algorithm”, *IEEE Microwave Conference Proceedings*, Suzhou, China, December 2005.

Stellenbosch University http://scholar.sun.ac.za

[59] C He, W Zhao, M Lu, "Time Domain Numerical Simulation for Transient Waves on Reconfigurable Co-processor Platform", *Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa, CA, USA, April 2005 .

[60] R Mittra, W Yu, *Advanced FDTD Method: Parallelization, Acceleration, and Engineering application,* Artech House, 2011.

[61] S Hagness, A Taflove, J Bridges, "Two-Dimensional FDTD Analysis of a Pulsed Microwave Confocal System for Breast Cancer Detection: Fixed-Focus and Antenna-Array Sensors", *IEEE Transactions on Biomedical Engineering*, vol. 45, pp.1470-1479, 1998.

[62] H Lee, F Teixeira, "Cylindrical FDTD Analysis of LWD Tools Through Anisotropic Dipping-Layered Earth Media", *IEEE Transactions on Geoscience and Remote Sensing*, vol. 45, pp. 383-388, 2007.

[63] I El-Kady, M Su, D Bader, "A novel FDTD Application Featuring openMP-MPI Hybrid Parallelization", *International Conference on Parallel Processing*, Montreal, Quebec, Canada, August 2004.

[64] W Walendziuk, J Forenc, "Verification of Computations of a Parallel FDTD Algorithm", *Proceedings of the International Conference on Parallel Computing in Electrical Engineering*, Warsaw, Poland, September 2002.

[65] IEEE Standards, *IEEE Recommended Practice for Validation of Computational Electromagnetics Computer Modeling and Simulations*, IEEE Electromagnetic Compatibility Society, 2011.

[66] A Elsherbeni, C Riley, C E Smith, "A Graphical User Interface for a Finite Difference Time Domain Simulation Tool", *Proceedings of the IEEE SoutheastCon 2000*, pp. 293-296, Nashville, TN, USA, April 2000.

[67] A Taflove, M J Piket, R M Joseph, "Parallel Finite Difference-Time Domain Calculations", *International Conference on Computation in Electromagnetics,* London, England, Nov 1991.

[68] R Lytle, "The Numeric Python EM project", *IEEE Antennas and Propagation Magazine,* vol. 44, 2002.

[69] V Andrew, C Balanis, C Birtcher, P Tirkaa, "Finite-Difference Time-Domain of HF Antennas", *Antennas and Propagation Society International Symposium*, Seattle, WA, USA, June 1994.

[70] T Namiki, "Numerical Simulation of Antennas using Three-Dimensional Finite-Difference Time-Domain Method", *High Performance Computing on the Information Superhighway, HPC Asia '97*, pp. 437-443, Seoul, Korea, May 1997.

[71] N Homsup, T Jariyanorawiss, "An Improved FDTD Model for the Feeding Gap of a Dipole Antenna", *Proceedings of IEEE SoutheastCon*, Charlotte-Concord, NC, USA, March 2010.

[72] S Watanabe, *Member,* M Taki, "An Improved FDTD Model for the Feeding Gap of a Thin-Wire Antenna", *IEEE Microwave and Guided Wave Letters*, April 1998.

Stellenbosch University http://scholar.sun.ac.za

[73] T P Stefanski, S Benkler, N Chavannes, N Kuster, "Parallel Implementation of the Finite-Difference Time-Domain Method in Open Computing Language", *International Conference on Electromagnetics in Advanced Applications (ICEAA),* Sydney, Australia, Sept 2010.

[74] R Alverson, D Roweth, L Kaplan, "The Gemini System Interconnect", *IEEE 18th Annual High Performance Interconnects (HOTI), Santa Clara, USA*, pp. 83-87, 2010.

[75] H Subramoni, K Kandalla1, J Vienne, S Sur, B Barth, K Tomko, R McLay, K Schulz, D K Panda, "Design and Evaluation of Network Topology-/Speed- Aware Broadcast Algorithms for InfiniBand Clusters", *IEEE International Conference on Cluster Computing (CLUSTER), Austin, USA*, pp. 317-325, 2011.

[76] C Rojas, P Morell, "Guidelines for Industrial Ethernet Infrastructure Implementation: A Control Engineer's Guide", *IEEE-IAS/PCA 52nd Cement Industry Technical Conference,* Colorado Springs, USA, pp. 1-18, 2010.

[77] White paper, *Q-logic High performance network, Introduction to Ethernet Latency*.
Available at: http://www.qlogic.com/Resources/Documents/TechnologyBriefs/Adapters/Tech_Brief_Introduction_to_Ethernet_Latency.pdf

[78] S Alam, R Barrett, M Bast, M R Fahey, J Kuehn, C McCurdy, J Rogers, P Roth, R Sankaran, J S Vetter, P Worley, W Yu, "Early Evaluation of IBM Bluegene/P", *International Conference High Performance Computing, Networking, Storage, and Analysis.* Austin: ACM/IEEE, 2008.

[79] Y Yang, P Xiang, M Mantor, H Zhou, "CPU-Assisted GPGPU on Fused CPU-GPU Architectures", *IEEE 18th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 1-12, New Orleans, USA, 2012.

[80] M Daga, A M Aji, W Feng, "On the Efficacy of a Fused CPU+GPU Processor (or APU) for Parallel Computing", *Symposium on Application Accelerators in High-Performance Computing (SAAHPC),* pp. 141-149, Knoxvill, USA, 2011.

[81] CUDA C Best Practices Guide NVIDIA manual.
Available at: http://www/nvidia.com

[82] CUDA Programming manual.
Available at: http://www/nvidia.com

[83] D Hackenberg, R Schöne, D Molka, M S Müller, A Knüpfer, "Quantifying power consumption variations of HPC systems using SPEC MPI benchmarks", *Journal of Computer Science - Research and Development* , vol. 25, no. 3, pp. 155-163, 2010.

[84] A Petitet, R C Whaley, J Dongarra, A Cleary, "HPL – A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers", *Innovative Computing Laboratory,* University of Tennessee.

[85] V Morozov, "Bluegene/P Architecture, Application Performance and Data Analytics" *Argonne Leadership Computing Facility, Getting Started Workshop,* Argonne National Laboratory, 2011.

[86] X Meilian, P Thulasiraman, "Parallel Algorithm Design and Performance Evaluation of FDTD on 3 Different Architectures: Cluster, Homogeneous Multicore and Cell/B.E.", *The 10th IEEE*

Stellenbosch University http://scholar.sun.ac.za

International Conference on High Performance Computing and Communications, pp. 174-181, Dalian, China , Sept 2008.

[87] P H Ha, P Tsigas, O J Anshus, "The Synchronization Power of Coalesced Memory Accesses", *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, pp. 939-953, 2010.

[88] Y Tian, C Lin, K Hu, "The Performance Model of Hyper-Threading Technology in Intel Nehalem Microarchitecture", *3rd International Conference on Advanced Computer Theory and Engineering (ICACTE)*, vol. 3, pp. 379-383, Chengdu, China, Aug 2010.

[89] H Inoue, T Nakatani, *Performance of Multi-Process and Multi-Thread Processing on Multi-core SMT Processors*, IBM Corp.

[90] B K Cabralg, W Lagunar, R Mcleod, S L Ray, S T Pennockr, L Bergerm, F Bland, "Interactive Pre and Post-Processing Tools", *Antennas and Propagation Society International Symposium*, San Jose, CA, USA, June 1989.

[91] S Krakiwsky, L Tumer, M Okoniewski, "Acceleration of Finite-Difference Time-Domain (FDTD) Using Graphics Processor Units (GPU)", *IEEE MTT-S International Microwave Symposium Digest*, vol. 2, pp. 1033-1036, June 2004.

[92] T Stefanski, N Chavannes, N Kuster, "Hybrid OpenCL-MPI Parallelization of the FDTD Method", *International Conference on Electromagnetics in Advanced Applications (ICEAA)*, pp. 1201-1204, Torino, Italy, Sept 2011.

[93] A G Schiavone, I Codreanu, R Palaniappan, P Wahid, "FDTD speedups obtained in distributed computing on a Linux workstation cluster", *IEEE International Symposium Antennas and Propagation Society*, vol. 3, pp. 1336-1339, Salt Lake City, UT, USA, July 2000.

[94] J R Yost, "IEEE Annals of Computing", *IEEE Journals and Magazines*, vol. 3, pp. 2-4, 2011.

[95] J G Koomey, S Berard, M Sanchez, H Wong, "Implications of Historical Trends in the Electrical Efficiency of Computing", *IEEE Annals of the History of Computing*, vol. 33, pp. 46-54, 2011.

[96] Microsoft HLSL Programming manual.
Available at: http://msdn.microsoft.com/en-us/library/windows/desktop/bb509635(v=vs.85).aspx

[97] White paper: A Osseyran , P Michielse, *Mapping applications to architectures – the strategy at SARA*, May 2012.

[98] R Bolla, R Bruschi, "RFC 2544 performance evaluation and internal measurements for a Linux based open router", *Workshop on High Performance Switching and Routing*, Poznan, Poland, June 2006.

[99] L Zhang, X Yang, W Yu, "Acceleration study for the FDTD method using SSE and AVX Instructions", *2nd International Conference on Consumer Electronics, Communications and Networks (CECNet),* pp. 2342-2344, Hubei, China, April 2012.

Stellenbosch University http://scholar.sun.ac.za

[100] W Yu, X Yang, Y Liu, R Mittra, J Wang, W Yin, "Advanced Features to Enhance the FDTD Method in GEMS Simulation Software Package", *IEEE International Symposium on Antennas and Propagation (APSURSI)*, pp. 2728-2731, Washington, USA, July 2011.

[101] A Foong, J Fung, D Newel, "An In-Depth Analysis of the Impact of Processor Affinity on Network Performance", *12th IEEE International Conference on Networks, (ICON 2004), Proceedings,* pp. 244-250, Singapore, Nov 2004.

[102] M Lee, Y Ryu, S Hong, C Lee, "Performance Impact of Resource Conflicts on Chip Multi-processor Servers", *PARA'06 Proceedings of the 8th international conference on Applied parallel computing: state of the art in scientific computing*, pp. 1168-1177, Umea, Sweden, June 2006.

[103] Moore's law website.
Available at: http://www.mooreslaw.org

[104] IBM Bluegene/P Appplication Programming Manual.
Available at: http://www.redbooks.ibm.com/redbooks/pdfs/sg247287.pdf

[105] D M Sullivan, "Exceeding the Courant condition with the FDTD method", *IEEE Microwave and Guided Wave Letters*, vol. 6, pp. 289-291, 1996.

[106] G Mur, "Absorbing Boundary Conditions for the Finite-Difference Approximation of the Time-Domain Electromagnetic-Field Equations", *IEEE Transactions on Electromagnetic Compatibility*, vol. EMC-23, pp. 377-382, 1981.

[107] J P Berenger, "Perfectly Matched Layer for the FDTD Solution of Wave-Structure Interaction Problems", *IEEE Transactions on Antennas and Propagation,* vol. 44, pp. 110-117, 1996.

[108] D B Davidson, "A review of important recent developments in full-wave CEM for RF and Microwave Engineering [Computational Electromagnetics]", *2004 3rd International Conference on Computational Electromagnetics and Its Applications, 2004, Proceedings ICCEA 2004,* pp. 1-4, Beijing, China, Nov 2004.

[109] C A Balanis, *Advanced Engineering Electromagnetic,* New York: Wiley, Chapter 6, 1989.

[110] R J Luebbers, K S Kunz, M Schneider, F Hunsberger, "A Finite-Difference Time-Domain Near Zone to Far Zone Transformation", *IEEE Transactions on Antennas and Propagation*, vol. 39, 1991.

[111] P Konczak, M Sypniewski, "GPU accelerated multiplatform FDTD simulator", *IET 8th International Conference on Computation in Electromagnetics (CEM 2011),* pp. 1-2, Wroclaw, Poland, April 2011.

[112] T Stefanski, N Chavannes, N Kuster, "Performance Evaluation of the Multi-Device OpenCL FDTD solver", *Proceedings of the 5th European Conference on Antennas and Propagation (EUCAP)*, pp. 3995 – 3998, Rome, Italy, April 2011.

[113] P Goorts, S Rogmans, S Vanden, Eynde, P Bekaert, "Practical Examples of GPU Computing Optimization Principles", *Proceedings of the 2010 International Conference on Signal Processing and Multimedia Applications (SIGMAP)*, pp. 46-49, Athens, Greece, July 2010.

 Stellenbosch University http://scholar.sun.ac.za

[114] R Schneider, M M Okoniewski, L Turner, "Custom Hardware Implementation of the Finite-Difference Time-Domain (FDTD) Method", *IEEE MTT-S International Microwave Symposium Digest*, pp. 875-878, Seattle, WA, USA, June 2002.

[115] M Douglas, M Okoniewski, M A Stuchly, "Accurate Modeling of Thin Wires in the FDTD Method", *IEEE Antennas and Propagation Society International Symposium*, pp. 1246-1249, Atlanta, GA, USA, June 1998.

[116] D Gorodetsky, A Wilsey, "A Signal Processing Approach to Finite-Difference Time-Domain Computations", *48th Midwest Symposium on Circuits and Systems*, pp. 750-753, Cincinnati, Ohio, USA, August 2005.

[117] S E Krakiwsky, L E Turner, M Okoniewski, "Graphics Processor Unit (GPU) Acceleration of Finite-Difference Time-Domain (FDTD) Algorithm", *Proceedings of the 2004 International Symposium on Circuits and Systems*, V-265 - V-268, Vancouver, Canada, May 2004.

[118] O Cen Yen, M Weldon, S Quiring, L Maxwell, M Hughes, C Whelan, M Okoniewski, "Speed It Up", *IEEE Microwave Magazine*, pp. 70-78, 2010.

[119] V Varadarajan, R Mittra, "Finite-Difference Time-Domain (FDTD) Analysis Using Distributed Computing", *IEEE Microwave and Guided Wave Letters*, pp. 144-145, 1994.

[120] D Cabrera, X Martorell, G Gaydadjiev, E Ayguade, D Jimenez-Gonzalez, "OpenMP Extensions for FPGA Accelerators", *International Symposium on Systems, Architectures, Modelling, and Simulation*, pp. 17-24, Samos, Greece, July 2009.

[121] A Sheppard, *Programming GPUs*, O'Reilly Media, 2004.

[122] J Durbano, F E Ortiz, J R Humphrey, P F Curt, D W Prather, **"**FPGA-Based Acceleration of the 3D Finite-Difference Time-Domain Method", *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa, CA, USA, April 2004.

[123] J Serre, P Lewis, K Rosenfeld, "Implementing OSI-based Interfaces for Network Management", *IEEE Communications Magazine*, pp.76-81, 1993.

[124] J A Herdman, W P Gaudin, D Turland, S D Hammond, "Benchmarking and Modelling of POWER7, Westmere, BG/P, and GPUs: An Industry Case Study", *SIGMETRICS Performance Evaluation Review* 38(4), pp. 16-22 ,2011.

[125] Y Lu, C Qiu, R Lu, Z Su, "Parallelization Efficiency of an FDTD Code on a High Performance Cluster under Different Arrangements of Memory Allocation for Material Distribution", *IEEE Antennas and Propagation Society International Symposium,* pp. 4901-4904, Honolulu, Hawaii, USA, June 2007.

[126] N Oguni, H Aasai, "Estimation of Parallel FDTD-based Electromagnetic Field Solver on PC Cluster with Multi-Core CPUs", *Electrical Design of Advanced Packaging and Systems Symposium*, pp. 159-162, Seoul, Korea, Dec 2008.

[127] Jülich Supercomputing Centre.
Available at: http://www2.fz-juelich.de/jsc/general

Stellenbosch University http://scholar.sun.ac.za

[128] Google Platform.
Available at: http://en.wikipedia.org/wiki/Google_platform

[129] K Kima, Q Park, "Overlapping Computation and Communication of Three-Dimensional FDTD on a GPU Cluster", *Computer Physics Communications*, vol. 183, Issue 11, pp. 2364-2369, 2012.

[130] W Pala, A Taflove, M Piket, R Joseph," Parallel Finite Difference-Time Domain Calculations", *International Conference on Computation in Electromagnetics*, pp. 83-85, London, England, Nov 1991.

[131] L Liguo, M Jinjun, Z Guangfu, Y Naichang, "A Parallel FDTD Simulator based on the Tianhe-1A supercomputer", *IEEE International Conference on Microwave Technology & Computational Electromagnetics (ICMTCE),* pp. 406-409, Beijing, China, May 2011.

[132] W Yu, "A Novel Hardware Acceleration Technique for High Performance Parallel FDTD Method", *IEEE International Conference on Microwave Technology & Computational Electromagnetics (ICMTCE)*, pp. 441-444, Beijing, China, May 2011.

[133] G Schiavone, I Codreanu, P Palaniappan, P Wahid, "FDTD Speedups Obtained in Distributed Computing on a Linux Workstation Cluster", *IEEE Antennas and Propagation Society International Symposium,* pp. 1336-1339, Salt Lake City, UT, USA, July 2000.

[134] T Martin, "An Improved Near to Far Zone Transformation for the Finite Difference Time Domain Method", *IEEE Transactions on Antennas and Propagation,* pp. 1263-1271,1998.

[135] W Yu, R Mittra, "Development of a Graphical User interface for the Conformal FDTD Software Package", *IEEE Antennas and Propagation Magazine*, vol. 45, NO. 1, 2003.

[136] Y Jiang, **"**The Use of CAD Data to Create a System of Two-Dimensional Finite Element Analysis of Data", *International Conference on E-Product E-Service and E-Entertainment (ICEEE),* pp. 1-3, Henan, China, Nov 2010 .

[137] FEKO.
Available at: http://www.feko.info

[138] Hadoop High Performance data storage system.
Available at: http://hadoop.apache.org/

[139] The Green 500 List.
Available at: http://www.green500.org/

[140] openMP Manuals.
Available at: http://www.openMP.org/

[141] E Quinnell, E Swartzlander, C Lemons, "Floating-Point Fused Multiply-Add Architectures, Signals", *Conference Record of the Forty-First Asilomar Conference on Systems and Computers*, pp. 331-337, Pacific Grove, California, USA, Nov 2007.

[142] Wikipedia Torus Interconnects.
Available at: http://en.wikipedia.org/wiki/Torus_interconnect

Stellenbosch University http://scholar.sun.ac.za

[143] B Subramaniam, W Feng, "The Green Index: A Metric for Evaluating System-Wide Energy Efficiency in HPC Systems", *IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW),* pp. 1007-1013, Shanghai, China, May 2012.

[144] W Yu, X Yang, Y Liu, R Mittra, A Muto, *Advanced FDTD Method*. Artech House, Boston, pp. 248, 2011.

[145] C A Balanis, *Advanced Engineering Electromagnetics*, New York: Wiley, Chapter 1, 1989.

[146] Information Technology - Portable Operating System Interface (POSIX) Operating System Interface (POSIX), *ISO/IEC/IEEE 9945 (First edition 2009-09-15),* 2009.
Available at: http://en.wikipedia.org/wiki/POSIX

[147] D B Davidson, "High-Fidelity FDTD Modelling of Far-Field TE Scattering from a PEC Cylinder", *Propagation IEEE Antennas and Propagation International Symposium*, Chicago, Illinois, USA, July 2012.

[148] MOAB Job scheduler.
Available at: http://www.adaptivecomputing.com/products/hpc-products/moab-hpc-basic-edition

[149] R G Ilgner, D B Davidson, "A comparison of the FDTD algorithm implemented on an integrated GPU versus a GPU configured as a co-processor", *IEEE International Conference on Electromagnetics in Advanced Applications (ICEAA),* pp. 1046-1049, Cape Town, South Africa, Sept 2012.

[150] R G Ilgner, D B Davidson, "A comparison of the parallel implementations of the FDTD method on a Shared Memory Processor and a GPU", *AfriCOMP11 - Proceedings of AfriCOMP 2nd African Conference on Computational Mechanics,* pp. 1-6, Cape Town, South Africa, Jan 2011.

[151] R G Ilgner, D B Davidson, "The acceleration of the FDTD method using extended length registers in contemporary CPUs", *South African Centre for HPC National Meeting,* pp. 1- 4, Pretoria, South Africa, Dec 2011.

[152] R G Ilgner, D B Davidson, "A comparison of the performance of the parallel FDTD method using openMP versus the parallel FDTD method using MPI on the Sun Sparc M9000", *South African Centre for HPC National Meeting*, pp. 1- 4, Cape Town, South Africa, Dec 2010.

[153] D Pasalic, M Okoniewski, "Accelerated FDTD Technique for Marine Controlled Source Electromagnetic Imaging", *International Conference on Electromagnetics in Advanced Applications (ICEAA)*, pp. 1198 – 1200, Cape Town, WP, South Africa, Sept 2012.

[154] Controlled source electro magnetics.
Available at: http://en.wikipedia.org/wiki/Controlled_source_electro-magnetic

[155] Y Zhang, J Owens, "A Qualitative Performance Analysis Model for GPU Architectures", *International Conference on Electromagnetics in Advanced Applications (ICEAA)*, pp. 382 – 1393, Torino, Italy, Sept 2011.

Stellenbosch University http://scholar.sun.ac.za

[156] R Luebbers, "Lossy dielectrics in FDTD", *IEEE Transactions on Antennas and Propagation*, pp. 1586 – 1588, 1993.

[157] D Thiel, R Mittra, "Surface Impedance Modeling Using the Finite-Difference Time-Domain Method", *IEEE Transactions on Geoscience and Remote Sensing*, pp. 1350 – 1357, 1997.

[158] OpenCL - The open standard for parallel programming of heterogeneous systems. Available at: http://www.khronos.org/opencl.

[159] D. Goldberg, What every computer scientist should know about floating-point arithmetic, ACM Computing Surveys, vol 23, pp. 5-48, 1991.

[160] Intel White paper, T. Mattson, K. Strandberg, Parallelization and Floating Point Numbers. Available at: http://software.intel.com/en-us/articles/parallelization-and-floating-point-numbers.

[161] D. Orozco, G. Gao, "Mapping the FDTD Application to Many-Core Chip Architectures", *IEEE International Conference on Parallel Processing (ICPP)*, pp. 309-316, Vienna, Austria, Sept 2009.

[162] Intel White paper, S.Zhou,G.Tan, FDTD Algorithm Optimization onIntel® Xeon Phi TM coprocessor. Available at: http://software.intel.com/en-us/articles/fdtd-algorithm-optimization-on-intel-xeon-phi-coprocessor.

[163] J. Toivanen,T.Stefanski,N. Kuster,N. Chavannes, "Comparison of CPML Implementations for the GPU-Accelerated GPU solver", *Progress In Electromagnetics Research*, vol 19,pp. 61 – 75, 2011.

[164] Floating Point Units. Available at: http://en.wikipedia.org/wiki/Floating-point_unit

Stellenbosch University http://scholar.sun.ac.za

Chapter 9

9 Appendix - Verification of the FDTD implementations

9.1 Objectives of the verification
9.2 Far Field modelling with the FDTD
9.3 The FDTD model
9.4 Modelling with FEKO
9.5 A comparison of the FDTD and FEKO derived MOM results

9.1 Objectives of the verification

To verify that the FDTD algorithm used for the work described in this document produces correct computational results, the output of a classic computational modelling scenario computed with the FDTD was compared to an equivalent scenario modelled using FEKO, a commercial EM modelling software using the MOM technique as a solver.

9.2 Far Field modelling with the FDTD

This is based on a scattered field formulation which avoids the complication of separating the total and incident fields from the far field formulation. The following formulations and discussions are therefore based on the scattered field. The frequency domain near to far field formulation is well known and is based on the theoretical framework described by Luebbers [110], Martin [134], and Taflove [13]. As the determination of scattering from an arbitrarily shaped body can be quite complex, the principle of equivalence [13] is used and is drawn for the two dimensional case in Figure 9.1. The basic premise is that the electromagnetic radiation from an arbitrarily shaped body can be calculated from a fictitious surface known as the equivalent surface. The equivalence theorem is described in precise detail in texts such as Balanis [109] and Taflove [13] and the Figure 9.1 serves to introduce the surface electric and magnetic currents J and M, which are central to the calculations that follow. For the surface enclosing the scattering object, as shown in figure 9.1, the scattered and magnetic electric fields give rise to $\boldsymbol{J_s} = \boldsymbol{n} \times \boldsymbol{H}$, and $\boldsymbol{M_s} = \boldsymbol{E} \times \boldsymbol{n}$ where n is a normal to the surface and J, E, M and H are all vector quantities.

Stellenbosch University http://scholar.sun.ac.za

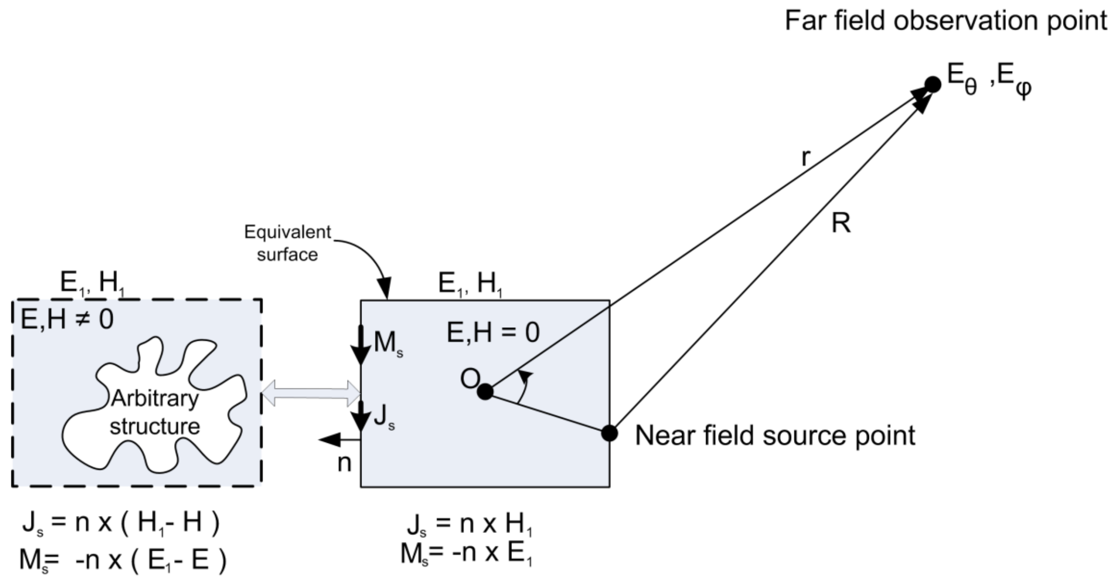

Figure 9.1: The far field calculated from the equivalent surface.

The far field is defined as being in the region where $r > {}^{2D^2}/_{\lambda}$ where r is the distance to the point of observation from the scattering object, D is the largest dimension of the scattering object, and λ is the wavelength of the source signal. As the antenna is a vertical structure in figure 9.1, the length of the dipole antenna, D, is not labelled.

The far field values E and H are derived from the integrated vector potential contributions from an equivalent surface as is shown in figure 9.1. The vector potentials from the equivalent surface are calculated from the electric and magnetic currents as is shown in the equations 9.1 and 9.2. The electric and magnetic surface currents are calculated from their respective contributing equivalent surface components, as described by Martin [134]. In the computational sense, the equivalent surface is made up of electric and magnetic field points that are spatially displaced [110, 134]. The electric surface currents are calculated from the magnetic field points making up the equivalent surface. The equivalent magnetic surface currents are derived from the electric field points making up the equivalent surface.

$$A = \frac{\mu_0}{4\pi} \iint_S \mathbf{J_S} \frac{e^{-jkR}}{R} dS = \frac{\mu_0 e^{-jkr}}{4\pi r} N \qquad 9.1$$

$$F = \frac{\varepsilon_0}{4\pi} \iint_S \mathbf{M_S} \frac{e^{-jkR}}{R} dS = \frac{\varepsilon_0 e^{-jkr}}{4\pi r} L \qquad 9.2$$

where radiation vectors N and L are defined as equations 9.3 and 9.4. R is the distance to the point of observation, r is the distance from the source to the point of observation and r’ is the distance from the source to a point on the equivalent surface, k is the wave number defined as $k = \frac{2\pi f}{c}$.

Stellenbosch University http://scholar.sun.ac.za

Python formulations of the equations 9.1 and 9.2 are shown in the code segment 9.1 below. The code segment shows the calculation of J and M for one of the equivalent surfaces, bearing in mind that a 3D structure would have six of these. The procedural context for this code segment is given in a FDTD far field process flowchart shown as figure 9.2. Functionality from a specialist numerical functionality library called numPy, was used to perform some of the numerical operations.

```
Python code segment 9.1:

Fourier= dt * exp(-1j * omega * iteration * dt / MAX_ITER)

Summation of J and M, the Fourier transformed Magnetic and Electric
surface currents for every E and H FDTD iteration

Jz[sx-1,0:ny,0:nz] += Hy[sx,0:ny,0:nz] * Fourier   #  J= Surface
Normal x H

Mz[s,0:ny,0:nz] += Ey[s+1,0:ny,0:nz]  * Fourier
```

When the FDTD has completed all its iterations, it will have summed (integrated) all of the contributions of J and M from all of the faces of the equivalent surface. It is then required to calculate spherical coordinates outside the equivalent surface and perform a summation and a reverse transform of the J and M potential contributions.

$$N = \iint_S \mathbf{J_s}\, e^{jkr'\cos\psi} ds \qquad 9.3$$

$$L = \iint_S \mathbf{M_s}\, e^{jkr'\cos\psi} ds \qquad 9.4$$

The Python code fragments in segment 9.2 express the functionality of equations 9.3 and 9.4 in programmatic terms. Although some of the language elements of the code can be related to 9.3 and 9.4 intuitively, the “add.reduce_ravel” function performs the summation(integration).

Python code segment 9.2:

```
1)Ny1= dy * dz * add.reduce( ravel ( Jy[sx-1,0:ny,0:nz] *
exp(1j*beta*rcosphi[sx-1,0:ny,0:nz]) ) )

2)Nz1= dy * dz * add.reduce( ravel ( Jz[sx-1,0:ny,0:nz] *
exp(1j*beta*rcosphi[sx-1,0:ny,0:nz]) ) )

3)Ly1= dy * dz * add.reduce( ravel ( My[sx,0:ny,0:nz] *
exp(1j*beta*rcosphi[sx,0:ny,0:nz]) ) )

 4)Lz1= dy * dz * add.reduce( ravel ( Mz[sx,0:ny,0:nz] *
exp(1j*beta*rcosphi[sx,0:ny,0:nz]) ) )
```

Stellenbosch University http://scholar.sun.ac.za

As shown by Balanis [109], using the coordinates transform to spherical coordinates Theta and Phi, the amplitudes of the Electric and Magnetic field components can be expressed as in equations 9.5 to 9.8.

$$E_\theta = -j\omega\left(A_\theta + \eta_0 F_\phi\right) = -jk\frac{e^{-jkr}}{4\pi r}(L_\phi + \eta_0 N_\theta) \quad 9.5$$

$$E_\phi = -j\omega\left(A_\phi - \eta_0 F_\theta\right) = +jk\frac{e^{-jkr}}{4\pi r}(L_\theta + \eta_0 N_\phi) \quad 9.6$$

$$H_\theta = -\frac{E_\phi}{\eta_0} \quad 9.7$$

$$H_\phi = \frac{E_\theta}{\eta_0} \quad 9.8$$

Where $\eta_0 = \sqrt{\frac{\mu_0}{\varepsilon_0}}$ is the impedance of free space. 9.9

The electric and magnetic fields as functions of theta and phi can be expressed in Python code as in the code segment 9.3 below:

```
Python code segment 9.3:

1)Etheta =  -1j * beta * exp(-1j * beta * R) / 4.0 / Pi / R *
(Lphi + No * Ntheta)

2)Ephi=        1j * beta * exp(-1j * beta * R) / 4.0 / Pi / R *
(Ltheta - No * Nphi)

3)Htheta=    1j * beta * exp(-1j * beta * R) / 4.0 / Pi / R *
(Nphi - Ltheta / No)

4)Hphi=      -1j * beta * exp(-1j * beta * R) / 4.0 / Pi / R *
(Ntheta + Lphi / No)
```

Stellenbosch University http://scholar.sun.ac.za

The process flowchart in figure 9.2 is a functional summary of the far field calculation that incorporates the electromagnetic and programmatic aspects described above.

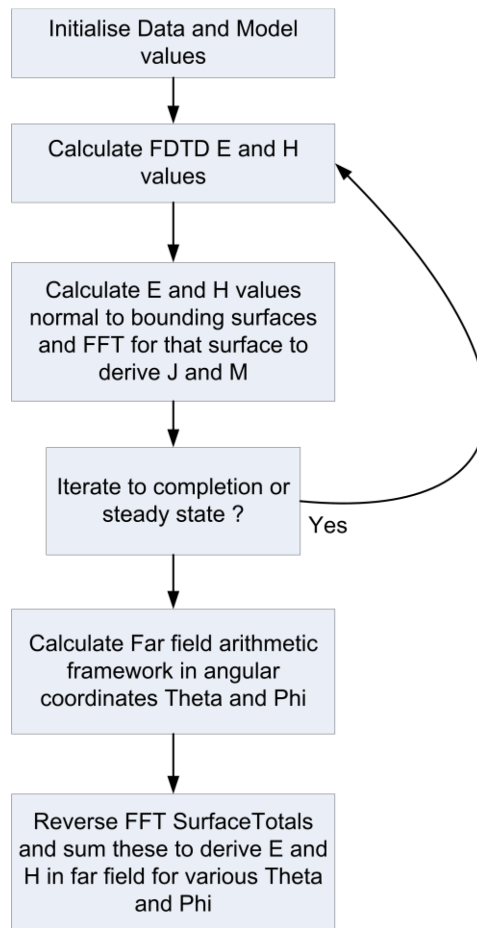

Figure 9.2: The FDTD far field calculation program flow.

9.3 The FDTD model

The model used for the FDTD far field calculation is shown below in planar view of the z, x axis and is mirrored in the z, y plane. The FDTD cell sizing was set to one cm in the x, y and z dimensions. The feed gap was excited at a frequency of 75 Mhz and was set to be the size of one cell, i.e. one cm. The far field was calculated at 100m from the dipole antenna.

Stellenbosch University http://scholar.sun.ac.za

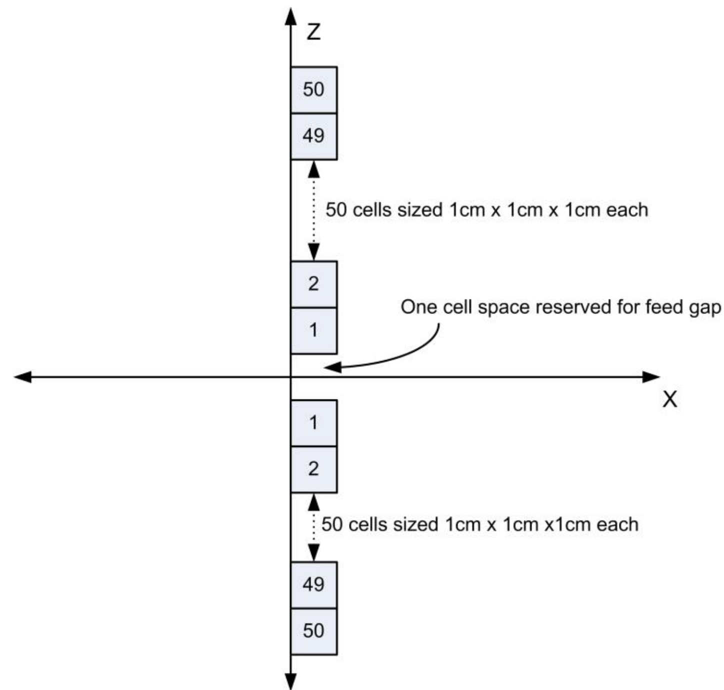

Figure 9.6: The FDTD vertical dipole model in the z, x plane

9.4 Modelling with FEKO

FEKO is a commercial software product [137] used for the simulation of electromagnetic fields. The software is based on the Method of Moments Integral formulation. The FEKO model consists of an infinitely thin vertical wire of the same length dimensions as the FDTD model, i.e. the dipole antenna was one m in length. Unlike the FDTD model, the thickness of the wire was very thin. The feed gap was excited with a continuous sine wave of 75 Mhz.

9.5 A comparison of the FDTD and FEKO derived MOM results

A comparison of the electric far field as caclulated by the FDTD and the FEKO MOM are show in the polar plot figure 9.3 below. The electric far field using either the FDTD or the MOM techniques looks very similar although there are some minor variations which could possibly be attributed to the difference in thickness of the source model.

Stellenbosch University http://scholar.sun.ac.za

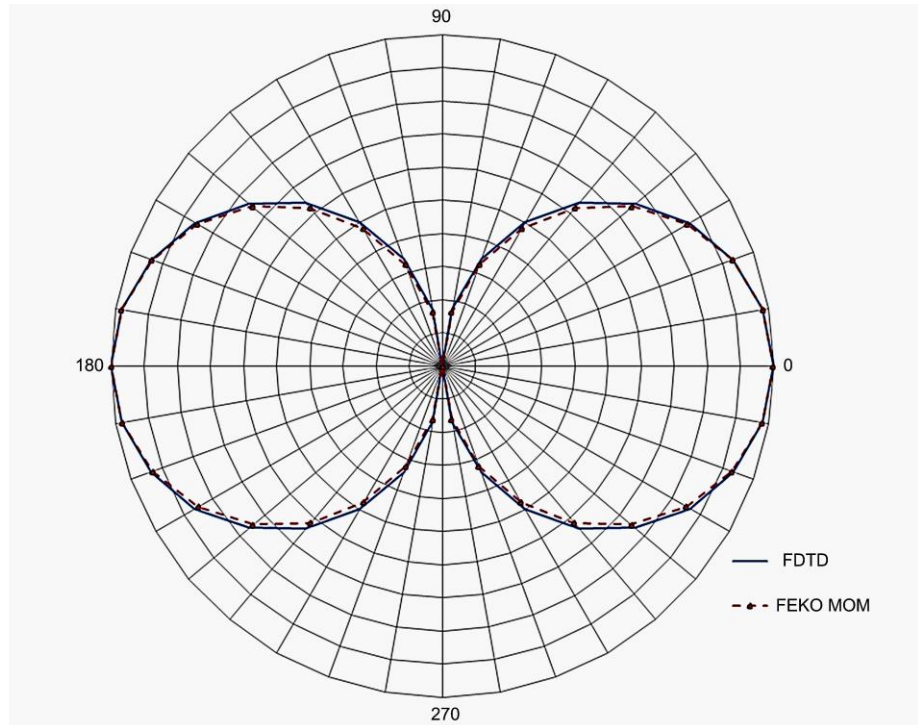

Figure 9.3: A comparison of the normalised electric field amplitudes using the FDTD and FEKO's Method of Moments.